Advanced Signal Processing Programming Assignment #3: Digital Communication

1 Introduction

In this exercise we shall use the theoretical principles taught in class in order to build a simple, yet fully operational communication system. The system will communicate using sound waves, where the transmitter will be a speaker connected to the computer, and the receiver will be a microphone. You can watch the short video we uploaded to moodle to see where we are headed at.

The channel model we shall use is the band-limited, continuous time AWGN and will use the principles appearing in Sections 2 and 3 of the lecture notes. As you shall see, making these principles work in real-life channel requires additional modules and elements. For this purpose, you will be given a step-by-step guidance in the process. In addition, in order to facilitate your implementation, some of the required parameters and functions are given to you.

Let's go!

1.1 Calculating the System Parameters

The audio channel is commonly sampled at a frequency of $F_s = 44.1$KHz. This number is derived from the human hearing range which is commonly assumed to span from 20Hz to 20KHz (though it varies between different human beings and changes with age). This sampling frequency theoretically gives us a bandwidth of 22.05KHz. However, the audio channel in this range is *frequency-selective*, and does not correspond to the simple AWGN model taught in class. This means that the channel's output is

$$Y(t) = h(t) \star x_w(t) + Z(t), \tag{1}$$

for some filter $h(t)$, whereas the model we have studied in class assumed $h(t) = \delta(t)$.

While the problem of dealing with frequency selective channels is well-understood, and most practical systems must address this issue in order to attain good performance, frequency selectivity adds a substantial complication to the communication system. In order to avoid it, we shall build a narrow band system, with a bandwidth of $B = 100$Hz. For an input signal with such a small bandwidth, the filter $h(t)$ behaves very similar to $\delta(t)$, so it may be viewed as an AWGN channel. As seen in the lecture, this implies a symbol spacing of

$$T_s = \frac{1}{B}. \tag{2}$$

It was also shown in class that Nyquist pulses can be used, facilitating the analysis of the continuous time system as a vector AWGN system. The pulse we shall use is a sinc pulse:

$$\psi(t) = \sqrt{B}\text{sinc}\,(Bt) = \frac{1}{\sqrt{T_s}}\text{sinc}\left(\frac{t}{T_s}\right) = \frac{1}{\sqrt{T_s}}\frac{\sin(\pi t/Ts)}{\pi t/Ts} \tag{3}$$

Since the value of $\frac{1}{\sqrt{T_s}}\frac{\sin(\pi t/Ts)}{\pi t/Ts}$ in not defined at $t = 0$, we shall standardly define $\psi(0)$ as the corresponding limit, i.e. $\psi(0) = 1/\sqrt{T_s}$. We will sometimes refer to this pulse as *the transmission pulse* and sometimes as *the Nyquist pulse.* The two names mean the same thing.

Another constraint discussed in class is the transmission time limit T. It implies that the entire transmission takes place <u>only</u> in the interval $[0, T]$.

The number of degrees of freedom K, indicates the dimension of the equivalent vector AWGN channel. It is also the number of orthogonal pulses that can be used in continuous time. It is clear that since the pulse spacing is T_s then $K \leq T/T_s = TB$. However, as shown in class, we should take:

$$K = \alpha TB \tag{4}$$

where $0 < \alpha < 1$ is the penalty paid for the fact that the transmission pulses are infinite in time, and practically undergo a finite truncation. In addition, the transmission pulse should be shifted in time so that:

$$\psi(t) = \sqrt{B}\text{sinc}\left(B\left(t - \frac{1-\alpha}{2}T\right)\right) \tag{5}$$

We assume that $(1-\alpha)T$ is sufficiently large, such that $\psi(t) \approx 0$ for all $t \notin [0, (1-\alpha)T]$. This implies that the receiver may only "listen" within the time interval $[0, T]$, where

$$T = KT_s + (1-\alpha)T, \tag{6}$$

and then start decoding.

In our system we modulate a single bit per transmission symbol. Therefore, the number of bits N is equal to the number of transmission pulses, which is equal to the number of degrees of freedom K . In the current exercise, the constraints are the number of degrees of freedom (i.e. the number of bits to transmit), and the delay of the sinc pulse in (5) (which is equal to $\frac{1-\alpha}{2}T$). In the following question we shall connect these values to T and α.

Question 1

Recall that $B = 100$Hz, denote the number of degrees of freedom (and the number of bits) by K. Setting $(1-\alpha)T = 32T_s$, and completely nulling $\psi(t)$ outside of the interval $[0, (1-\alpha)T_s]$, we obtain the transmission pulse:

$$\psi(t) = \begin{cases} \sqrt{B}\mathrm{sinc}\left(B\left(t-16T_s\right)\right) & \text{if } t \in [0, 32T_s] \\ 0 & \text{otherwise} \end{cases}. \tag{7}$$

Find the required transmission time T in order to obtain K degrees of freedom.

1.2 System High-Level

In the rest of the exercise, you will be asked to write a Matlab simulation whose high level architecture is depicted in Figure 1.
`TX`, `ChannelTXRX` and `RX` are functions representing the transmitter, channel and receiver respectively.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% create the information bits : infobits
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The Transmitter
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ChannelInVec = TX(config,infobits);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The Channel
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ChannelOutVec = ChannelTXRX(config,ChannelInVec);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The Receiver
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
rxbits = RX(config,ChannelOutVec);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process the output bits : rxbits
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 1: The high level of the communication system

We will update the functions `TX`, and `RX` from question to question, adding more sophistication to them, until in the end they will be sophisticated enough to deal with the audio channel.

1.3 The Configuration Structure

The file `config.mat` contains the Matlab `config` structure whose fields are elaborated in Table 1. To obtain this structure, type:

```
load('config.mat')
```

Then `config` will be a dictionary with the respective files, `config['Fs']`, `config['B']` etc.

Field Name	**Content**
General System Configuration	
`Fs`	=44100, the audio sampling frequency in Hz
`B`	=100, the system bandwidth in Hz
`Fc`	= `Fs`/32
`Ts`	=`Fs`/`B`, the symbol length in samples
`K`	Number of bits/degrees of freedom
	`config.K` **is configured by the user!**
`snrdB`	the SNR of the AWGN Matlab channel in dB
Pulse Shapes	
`tpulserect`	a vector with the values of the rectangular pulse shape
`tpulsesinc`	a vector with the values of the sinc pulse shape
Configuration of the Pass-Band system	
`RxLPFpulse`	a vector with the values of the RX LPF
`RxLPFFreq`	the cutoff frequency of the RX LPF
Synchronization Setting	
`nsynchbits`	The number of synchronization bits
`synchbits`	a vector with the values of the synchronization bits
`synchsymbol`	the synchronization symbol - after modulation
Audio Configuration	
`audiorecordinglength`	number of seconds to be recorded
`nChannels`	=1, internal audio interface configuration
`ID`	=-1, internal audio interface configuration
`nBitsAudio`	=16, internal audio interface configuration

Table 1: The entries of the config structure

1.4 Important Notes on the Implementation

The following notes are crucial to the success of your implementation.
Please read them carefully before you start implementing.

a. **Using the configuration structure to pass arguments between the functions.** Observe that in the system high level appearing in Figure 1, each of the three main functions have two inputs, and one of them is always the `config` structure. For example, the transmission function is activated by the line:

```
ChannelInVec = TX(config,infobits);
```

Naturally, the function `TX` may require additional parameters. There might also be parameter you would like to pass between the three functions (`TX`, `ChannelTXRX`, `RX`). This can be done, by adding the parameter to the configuration structure.

For example, if you want to set the number of degrees of freedom (and bits) to 10^4, you can do this by adding the following code-line to the beginning of the script:

```
config.K = 10^4;
```

By so doing, the value of `config.K` will be know to all three functions.

b. All the vectors included in the `config` structure are row vectors.

c. **Manipulating vector dimensions.** Note that in some cases, you might like a vector to be either a row or a column, but it would not be clear from the code whether it is a row or a column vector. In these cases, it is good practice to make the vector either row or column according to your preference. The Following line:

```
x = x(:);
```

changes any vector `x` to a column vector with the same elements.

```
x = x(:)';
```

changes any vector `x` to a row vector with the same elements.

d. Regarding the use of the sinc and rect pulses in questions 2-4, you are allow to either:

   a. Indicate the pulse (rect or sinc) to be used using a "flag" that you control **outside** of the TX and RX functions. For example, you can create a variable in 'config' named 'pulsetype', where config.pulsetype = 0 or 1 corresponds to a rect or sinc pulse.
   b. Indicate the pulse (rect or sinc) to be used from **inside** the TX and RX functions. In this case, you can write two separate lines that set the pulse to be used to be the sinc and rect, and leave one of them as a comment.

e. Regarding the modification of the TX and RX functions throughout the assignment, you are allow to either use:

a. The same TX and RX functions throughout the assignment, where you update them according to the instructions in each question. In this case, you can create two "flags" in 'config', that are controlled **outside** of the TX and RX functions: One flag for Q5, which enables the use of the synchronization in the RX function (namley, when extracting the optimal sample point), and another flag for Q6, which enables the use of the frequency modulation in the TX and RX functions, and the application of the LPF in the RX function.

b. Different TX and RX functions throughout the different questions of the assignment. In this case, please submit all the functions used, with titles indicating the respective question number, for questions 2-6.

   In any case, for Q7, you should submit two MATLAB files whose titles are 'TX' and 'RX', which contain only the function 'ChannelInVec = TX(config,infobits)' and 'rxbits = RX(config,ChannelOutVec)', respectively. These functions assume only sinc pulses, and have to be able to perform under **both synchronization and frequency modulation!**

2 A Matlab System with an AWGN Channel

As you shall see, using the audio media adds complexity to the system. We shall start with a Matlab based AWGN channel, which simulates the channel model we have seen in class. We will defer dealing with the audio interface to a later stage.

2.1 The Information Bits

In the final system, the information bits are derived from a text message. However, at this point we shall use 10^4 random bits drawn independently Ber(0.5). To draw a Ber(0.5) bit, you can use the `rand` command (i.e. uniform distribution between 0 and 1) and set the bit to one if the random variable is greater than 0.5, and to zero otherwise.

2.2 The Transmitter

2.2.1 2-PAM (BPSK) constellation

We use the following standard mapping from the input bit to the modulated symbol

$$0 \Rightarrow +1 \tag{8}$$

$$1 \Rightarrow -1 \tag{9}$$

In terms of energy, for any choice of information bits sequence, we will have that $\|\mathbf{x}_w\|^2 = K$, as $x_{w,i}^2 = 1$ for all $w \in [2^K]$ and all $i \in [K]$. This is precisely the 2-PAM constellation we have seen in the lectures, with $\Delta = 1$. This constellation is often also referred to as Binary Phase-Shift Keying (BPSK).

2.2.2 Moving to the "Continuous" Time Domain

As mentioned in class, most practical communication media can be regarded as channels in the continuous (rather than discrete) time domain. In this exercise the communication medium is the acoustic channel, whose bandwidth is several KHz. While practical communication systems usually apply both continuous time modules, and discrete time modules, in this exercise the entire processing will be in discrete time. We shall use the sampled time domain, and the sampling frequency will be the standard $F_s = 44.1$KHz. As the signals in the system will be in much lower frequencies, it is implied that the processing is done in a rate much higher than the Nyquist rate. The high sampling rate will enable us to explore some of the aspects of going from the continuous time AWGN channel to the vector AWGN channel.

As stated in (2) the time interval between two consecutive symbols is $T_s = \frac{1}{B}$, where T_s is in units of seconds and B is in units of Hz. With slight abuse of notation, we regard T_s in the simulation in units of discrete time samples (sampled at F_s) and therefore

$$T_s = \frac{F_s}{B}. \tag{10}$$

The construction of symbol-by-symbol on a pulse train (modulation of bits over the pulses) should be implemented as follows:

- Construct a *pulse train* from the vector containing the BPSK symbols. To do that, between every two consecutive symbols (± 1) in the BPSK vector, insert $T_s - 1$ zeros. On possible implementation is Matlab `upsample` command.

- Pass the resulting pulse train through a filter whose impulse response corresponds to the transmission pulse $\psi(t)$. Namely, if `PulseTrain` is the Matlab vector representing the pulse train, `txpulse` is the vector representing $\psi(t)$ and `TXFiltout` is the vector representing "continuous time" signal, then use:

  ```
  TXFiltout = conv(txpulse,PulseTrain);
  ```

  The Matlab `conv(a,b)` implements a convolution, which is equivalent to passing the signal `b` through a finite impulse response filter `a`.

  Note that you are given two different Nyquist pulses ("pulse shapes") in the config structure (see Table 1):

  – `config.tpulserect`, a vector with the values of the rectangular pulse shape.

  – `config.tpulsesinc`, a vector with the values of the sinc pulse shape.

  For transmission over the audio channel we will take `txpulse` as `config.tpulsesinc`, but throughout the Matlab simulations leading to the final communication system, we will often also take `txpulse` as `config.tpulserect`. Explicit instructions for which Nyquist pulse to use for each question are given.

- Concatenate $\mathbf{F_s}$ zeros at the **beginning** and $\mathbf{4F_s}$ zeros at the **end** of the output vector. This will correspond to one second of silence at the beginning and 4 seconds of silence at the end of the transmission.

The following question summarizes the implementation of the transmitter:

Question 2

a. Use `config.tpulserect` as the transmission pulse. Plot `config.tpulserect` and attach it to your solution. What is the L2 norm of this vector (i.e. the sum of the squares of its elements)?

b. Plot a figure with two subplots one on top of the other (using `subpplot` command in Matlab). The top figure should represent the pulse train respective to the first 32 input bits. The bottom plot should represent the corresponding output of the transmission filter (`TXFiltout = conv(txpulse,PulseTrain)`).

c. Repeat (a) using the sinc pulse in `config.tpulsesinc`.

d. Repeat (b) using the sinc pulse in `config.tpulsesinc`.

2.3 The Channel

Write the function `ChannelTXRX` which copies the input vector `ChannelInVec` to the output vector `ChannelOutVec` and adds Gaussian noise. Recall from the lectures that we define $\mathsf{SNR} = \frac{P}{\sigma^2}$. The average power of our 2-PAM constellation is $P = 1$. Thus, $\mathsf{SNR} = \frac{1}{\sigma^2}$ where σ^2 is the variance of each sample of the AWGN. The variance of the noise in our simulation therefore corresponds to the value of `config.snrdB`, which represents SNR of the system in dB scale. The Gaussian noise can therefore be drawn using Matlab `randn` command and multiplying the generated vector by `randn` by the factor `10^(-config.snrdB/20)`.

Question 3

In this question, use the sinc transmission filter.

a. Draw three figures, each containing two subplots. The top subplot representing the input of the channel (only first four seconds, namely, only the first $4F_s$ samples) and the bottom subplot representing the output of the channel (only first four seconds). The figures should correspond to three different channel SNR values: 100dB, 40dB, 12dB.

b. Bonus: Can you explain why in the 12dB plot, the signal is totally hidden in the noise at the channel output? Wouldn't you expect the signal to be larger than the noise at this point?

2.4 The Receiver

We start with a simple implementation of the receiver using the following modules:

- A matched filter, whose impulse response correspond to a time reversed version of the transmission pulse. The input of this filter is the channel output.
- Sampling the output of the matched filter in a spacing of F_s/B, i.e. 441 samples. Knowing from which sample to start is the tricky part. In the current system, the optimal first sampling point is F_s samples (corresponding to the one second delay) plus the length of the matched filter (corresponding to the fact that the matched filter is non-causal, so to circumvent this we actually convolve the channel output by a delayed version of the matched filter - namely by the the vector obtained by "flipping" the Nyquist pulse. The delay length is the length of the matched filter, which is also the length of the Nyquist pulse.).
- Reconstructing the BPSK symbols by taking the sign of the output of the previous stage.
- Mapping from BPSK (i.e. $+1$ or -1) back to bits (i.e. 0 or 1).

After this is done, calculate the average bit error rate. Namely, in our system

```
BER = mean(infobits~=rxbits);
```

The following question summarizes the implementation of the receiver.

Question 4

a. **Use the rectangular transmission filter.** Set SNR=100dB. Plot the output of the matched filter, in the time interval between the optimal first sampling point (as explained above, it should be F_s plus the length of the Nyquist pulse) minus 4000, to the same point plus 4000. Plot a red x in the optimal sampling point. Verify that it is indeed the optimal first sampling point.

b.* See that the system in the previous item gives BER=0.

In this section you will work with a reference input as follows:

load the file `RefInputRect.mat` which contains the following vectors: `ChannelOutVec`, `infobits`. Use `ChannelOutVec` as the input of your receiver (namely, run only your receiver function `RX`). Verify that the decoded bits are exactly the ones in `infobits` (namely, that BER=0).

c. Plot a *BER vs. SNR curve* as follows. Run a fully functional simulation, including transmitter, AWGN channel and receiver as depicted in Figure 1. Set `config.K=10^4`, and draw the information bits are independently Ber(0.5). Note that setting `config.K=10^4` implies that you only need to sample the first 10,000 bits and not all bits up to the 4 seconds of silence at the end of the transmission. Work in different SNRs points, from -12dB to $+12$dB in 3dB steps. Plot a figure whose X axis is the SNR in dB and whose Y axis is the measured BER in logarithmic scale (use `semilogy` plotting command).

d.* **Now change to the sinc transmission filter.** Repeat b with the reference signal in `RefInputSinc.mat` (the vector names are unchanged).

e. Repeat c with the sinc pulse shape.

<u>Note:</u> sections marked by an asterisk '*' (as section b.* above) are used only for your debugging and as a sanity check. They will not be checked nor graded, but you must nevertheless complete them to verify you are on the right track.

In the rest of the exercise, use only the sinc pulse!

2.5 Adding Synchronization

In the previous subsection we assumed that all the time elements and delays of the system are known, and that the receiver can exactly calculate the optimal sampling point of the matched filter output. However, in practical systems, there are many unknown delay elements and it is almost impossible to know the optimal sampling point in the required accuracy. This issue can be solved by adding a synchronization mechanism, which is based on sending a known signal at the beginning of the communication signal, and finding its location using a matched filter, just like we have studied under the topic of "Signal in Noise".

- We add synchronization bits before the information bits. The number of synchronization bits is `config.nsynchbits` and their values are stored in `config.synchbits` (known to both transmitter and receiver).
- At the transmitter, the synchronization bits are prepended to (i.e. put before) the information bits, and passed through the transmission filter. Namely, instead of generating the communication signal using `TX(config,infobits)`, we generate it using `TX(config,[config.synchbits,infobits])` .
- The receiver uses its knowledge of the synchronization bits, in order to identify the optimal sampling points. How is this done? Let $[a_0, \ldots, a_{N_{\text{sync}}-1}] \in \{-1, 1\}^{N_{\text{sync}}}$ be the vector of BPSK symbols corresponding to the synchronization bits `config.synchbits`, where $N_{\text{sync}} =$ `config.nsynchbits`. Let[1]

$$s(t) = \sum_{i=0}^{N_{\text{sync}}} a_i \psi \left(t - \frac{i}{B} \right), \tag{11}$$

  be the symbol-by-symbol on a pulse train signal corresponding to $[a_0, \ldots, a_{N_{\text{sync}}-1}]$. Note that we have added the synchronization bits before the information bits fed to the transmitter. Thus, any communication signal $x_w(t)$ generated by the transmitter,

[1]We use the notation $1/B$ instead of T_s seconds, as in (10) we have agreed to use T_s for the number of *samples* between any two shifts of the Nyquist pulse.

begins with the signal $s(t)$. This signal is known to the receiver, as the synchronization bits, and consequently also $[a_0, \ldots, a_{N_{\text{sync}}-1}]$, are known to it.

Now, if the received signal is $Y(t) = x_w(t-\tau) + Z(t)$, we know that $x_w(t-\tau)$ will begin with $s(t-\tau)$. For the task of synchronization, we treat the unknown part of $x_w(t)$ (that depends on `infobits` which are unknown to the receiver) as random noise. This results in the model $Y(t) = s(t-\tau) + \tilde{Z}(t)$. Under our topic "Signal in Noise", we have learned how to detect τ in (more or less) this model.[2] This is done by passing $Y(t)$ by the matched filter for the synchronization signal, namely $s(-t)$, and taking our estimate for τ as the time where the result is maximized.

We can therefore estimate τ, which is the time at which the synchronization signal first reached the receiver. The actual communication signal corresponding to the information bits begins N_{sync}/B seconds later.

For this exercise, the discrete-time synchronization signal s_n (corresponding to sampling $s(t)$ at sampling frequency F_s) was computed for you, and stored as `config.synchsymbol`. Do not confuse `config.synchsymbol` with `config.nsynchbits`!. Now, the length of the synchronization symbol is $N_{\text{sync}}T_s + L - 1$, as it is a convolution of the sinc pulse (of length L) with the N_{sync} bits upsampled by T_s. The point $n_{\max}$, where the matched filter using the flipped signal `config.synchsymbol` in MATLAB is maximal, should correspond to its end (if no error occurred due to the channel noise), as learned in the "Signal in Noise" topic. Therefore:

$$n_{\max} = n_{\text{start synch}} + N_{\text{sync}}T_s + L - 1 \tag{12}$$

So:

$$n_{\text{start synch}} = n_{\max} - N_{\text{sync}}T_s - L + 1. \tag{13}$$

Therefore, the point related the beginning of the first information bit is:

$$n_{\text{start synch}} + N_{\text{sync}}T_s \tag{14}$$

Now, n_{opt}, defined as the optimal sampling point related to the first information bit, should be L samples later. So, all in all:

$$\begin{aligned} n_{\text{opt}} =& n_{\text{start synch}} + N_{\text{sync}}T_s + L && (15)\\ =& n_{\max} - N_{\text{sync}}T_s - L + 1 + N_{\text{sync}}T_s + L && (16)\\ =& n_{\max} + 1 && (17) \end{aligned}$$

<u>Conclusion:</u> The optimal location for the first sampling of the first information bit at the output of the transmission pulse matched filter is

$$n_{\text{opt}} = n_{\max} + 1, \tag{18}$$

which is simply the location of the next sample after the maximum at the output of the synchronization matched filter as implemented in MATLAB!. The optimal sampling points related to the following bits will be in jumps of $T_s = 441$ samples from n_{opt}. In order to find the optimal sampling time, do the following:

[2]The noise $\tilde{Z}(t)$ is composed of both $Z(t)$ and the unknown part of $x_w(t)$. Thus, $\tilde{Z}(t)$ is not AWGN. Nevertheless, using the same solution derived for signal in additive white Gaussian noise works very well.

1. Compute the convolution between the "flipped version" of `config.synchsymbol` and the channel output. Find the time index (sample) at which the **maximum absolute value** is attained.[3]
2. Subtract the length of `config.synchsymbol` from this time, and add to it $N_{\text{sync}}T_s$ samples.
3. Add the length of `config.tpulsesinc`.

The time you get is the optimal sampling time of the convolution between the channel's output and the matched filter *for the Nyquist pulse* (i.e., flipped `config.tpulsesinc`), corresponding to the first information bit.

Warning: There are two different matched filters involved now, do not confuse them. One is the matched filter for the Nyquist pulse, i.e., the flipped version of `config.tpulsesinc`. We convolve the channel output with it in order to decode the transmitted bits. To that end, we sample its output in spacing of $T_s = 441$ symbols, as we have done in the previous question. If there is no delay, we know exactly at what time we should take the first sample, and this time was calculated and used in the previous question. Now, in the presence of unknown delay, we need to figure out what is the optimal time for the first sample. This is done using the matched filter for the synchronization signal, i.e., the flipped version on `config.synchsymbol`. We convolve the channel output with it and follow steps (1), (2) and (3) above to calculate the time at which the first sample should be taken.

- After the optimal first sampling point for the first information bit, sample the output of the matched filter (for `config.tpulsesinc`) every $F_s/B = 441$ samples, as before.

Question 5

a. Explain why each of the steps (1), (2) and (3) above is needed for the calculation/estimation of the optimal sampling time for the first information bit.

b. Note that after following the recipe in steps (1), (2) and (3), the obtained optimal sampling time is actually the time index (sample) at which the convolution between the "flipped version" of `config.synchsymbol` and the channel output is maximized. Namely, steps (2) and (3) cancel each other out. Explain why this happens.

c.* Add the synchronization mechanism to the transmitter and receiver as explained above. Namely, at the transmitter, add the known synchronization bits `config.synchbits` before the information bits, and pass both through the transmission filter. At the receiver, find the optimal first sampling point as explained above. You may and should limit the search to the first sampling point to the first 4 seconds ($4F_s$ samples). This will reduce complexity and prevent bugs whenever synchronization is not successful.

Run a fully functional simulation, including the new transmitter, AWGN channel and the new receiver as depicted in Figure 1. Work with SNR=100dB and $K = 10^4$. See you

[3] Taking absolute value is not important at this point, but will become important later when we assume the channel may also multiply the input by some unknown, possibly negative, coefficient β.

obtain BER=0. Note that when you calculate BER, check only the information bits and not the synchronization bits.

d.* Work with the reference signal stored in `RefInputSynch.mat` as you did in Question 4b. Note now a synchronization symbol was added at the transmitter. Note that an unknown delay was also added, so using the "hard coded" synchronization point (i.e. F_s plus the length of the transmission pulse) should not work. Verify you obtain BER=0.

e. Plot an SNR vs. BER curve as in Question 4c. Did the performance change with respect to Question 4c?

2.6 Adding Frequency Conversion and Channel Estimation

The communication system is supposed to work in a bandwidth of 100Hz using audio signals. However, typical audio systems (speakers, microphones etc.) are not designed to give high fidelity in very low frequencies, which are also not heard by the human ear. Therefore rather than transmitting a signal whose Fourier transform is supported on $[-B/2, B/2]$, it is useful, to convert the audio signal to the bands $[F_c - B/2, F_c + B/2] \cup [-F_c - B/2, -F_c + B/2]$. The frequency we use for the conversion is $F_c = F_s/32 \approx 1.37$KHz. We will first describe how this works in continuous time, and then describe how to incorporate the frequency conversion into our simulation.

Recall that for a signal $x(t)$ with Fourier transform $\mathcal{F}\{x(t)\} = X(f)$ we have that if

$$x^{\mathrm{UP}}(t) = x(t)\cos(2\pi F_C t),$$

is the *up-conversion* of $x(t)$, then

$$\mathcal{F}\left\{x^{\mathrm{UP}}(t)\right\} = \frac{1}{2}X(f - F_c) + \frac{1}{2}X(f + F_c).$$

Thus, if $X(f) = 0$ for all $f \notin [-B/2, B/2]$, then $\mathcal{F}\left\{x^{\mathrm{UP}}(t)\right\} = 0$ for all $f \notin [F_c - B/2, F_c + B/2] \cup [-F_c - B/2, -F_c + B/2]$. In particular, even if the channel is $Y(t) = X(t) \star h(t) + Z(t)$, for some filter $h(t)$ that is not a scaled delta function, but satisfies $H(f) = \beta$ for all $f \in [F_c - B/2, F_c + B/2] \cup [-F_c - B/2, -F_c + B/2]$, if we transmit the signal $x^{\mathrm{UP}}(t) = x(t)\cos(2\pi F_C)$ through it, we will get

$$Y(t) = \beta x(t)\cos(2\pi F_c t) + Z(t).$$

As mentioned above, a typical audio channel (consisting of a speaker and a microphone) is more accurate at higher frequencies than in low frequencies (any bass guitar player knows that large speakers are required in order to generate an accurate response at low frequencies. Generating a reasonable response at higher frequencies is doable even with cheap and small speakers). Thus, multiplying $x(t)$ by $\cos(2\pi F_c t)$ prior to transmission guarantees that it will be less distorted by the channel.

Next, at the receiver, we would like to cancel out the multiplication by $\cos(2\pi F_c t)$. To that end, we multiply the channel output $Y(t)$ by a cosine with the same frequency, namely with $\cos(2\pi F_c t)$.[4] Thus, we compute

$$\begin{aligned}\tilde{Y}(t) &= Y(t)\cos(2\pi F_c t) \\ &= \beta x(t)\cos^2(2\pi F_c t) + Z(t)\cos(2\pi F_c t) \\ &= \beta x(t)\cdot\frac{1}{2}(1+\cos(2\pi 2F_c t)) + Z(t)\cos(2\pi F_c t) \\ &= \frac{\beta}{2}x(t) + \frac{\beta}{2}x(t)\cos(2\pi 2F_c t) + Z(t)\cos(2\pi F_c t)\end{aligned}$$

Now, we would like to filter out the signal $\frac{\beta}{2}x(t)\cos(2\pi 2F_c t)$ whose Fourier transform $\frac{\beta}{4}X(f-2F_c)+\frac{\beta}{4}X(f+2F_c)$ is supported in $[2F_c-B/2, 2F_c+B/2]\cup[-2F_c-B/2, -2F_c+B/2]$. To that end, we use a low-pass filter $h^{\text{LPF}}(t)$ with $H^{\text{LPF}}(f)=1$ for $|f|\le B/2$ and $H^{\text{LPF}}(f)=0$ for $|f|>F_{\text{cut-off}}$, where $B/2\le F_{\text{cut-off}}\le 2F_c-B/2$, provided that $B/2<2F_c-B/2$. Thus, provided that $B<2F_c$ we get that

$$\begin{aligned}Y^{\text{down}}(t) &= h^{\text{LPF}}(t)\star\tilde{Y}(t) \\ &= \frac{\beta}{2}x(t) + h^{\text{LPF}}(t)\star(Z(t)\cos(2\pi F_c t)). \qquad (19)\end{aligned}$$

We omit the analysis of the statistics of $h^{\text{LPF}}(t)\star(Z(t)\cos(2\pi F_c t))$. We will assume however that after applying the matched filter and sampling every $1/B$ seconds, the obtained noise in discrete-time is AWGN. Thus, using up-conversion and down conversion we are able to use the channel at the desired frequency band (where it behaves more favourably, or where there are less interfering signals) and still reduce the problem to the AWGN channel we considered in class.

Recall that we simulate the continuous time signals by the discrete-time signals obtained by sampling them $1/F_s$ seconds. Therefore, in our simulation the conversion is done us follows:

- In the transmitter function `TX` we generate the channel input signal exactly as before, but in the end we multiply it by the sampled cosine function $\cos\left(\frac{2\pi F_c}{F_s}n\right)$.
- At the receiver, we first multiply the channel output by the sampled cosine function $\cos\left(\frac{2\pi F_c}{F_s}n\right)$, and then filter it using the low-pass filter `config.RxLPFpulse`. The resulting signal should be

$$\frac{\beta}{2}x_n + Z_n, \qquad (20)$$

[4] In practice, it is impossible to match the phase of the cosine used at the transmitter and at the receiver. Thus, it is more accurate to say that we multiply $Y(t)$ by $\cos(2\pi F_C t+\phi)$, for some $\phi\in[0,2\pi)$, representing the mismatch between the transmitter's and receiver's "clock". There are ways in which the receiver can estimate the phase ϕ, and adapt the cosine used for multiplication accordingly. Here, to keep things simple we ignore this aspect, and assume $\phi=0$. When you perform the audio simulations, this phase mismatch is present and may occasionally degrade performance significantly. Since this phase is a random variable, its effect will only occasionally degrade performance.

This is exactly the model we have dealt with in the previous sections, with the only difference is that now we have some unknown gain β (if β were equal to 2, this would have been precisely the model from before).

Thus, we will estimate β, and divide the resulting signal by $\beta/2$, to obtain the channel model $x_n + \frac{2}{\beta}Z_n$, which we have already dealt with.

2.7 Channel Estimation

To estimate β, we use the synchronization mechanism we have already incorporated in the transmitter `TX` and the receiver `RX` in Question 5. Recall that for synchronization we are passing the channel output through the matched filter corresponding to `config.synchsymbol`. You may verify that if there were no noise in the system, the maximum of the resulting signal's absolute value should be $\frac{|\beta|}{2}E_{\text{synch}}$, where E_{synch} is the energy (squared L2 norm) of the signal `config.synchsymbol`. Thus, we estimate $\beta/2$ by finding the maximum absolute value of the matched filter output, divided by E_{synch}. If the corresponding absolute maximum was negative before taking absolute value, we then also multiply by -1.

We are now ready for the implementation:

Question 6

a.* Implement in the functions `TX` and `RX` the up/down conversion, as well as estimation of $\beta/2$ and its cancellation, according to the instructions above. Use the low pass filter whose coefficients are in `RxLPFpulse`. In the transmitter we should generate the communication signal just as in Question 5, and then multiply it by $\cos\left(\frac{2\pi F_c}{F_s}n\right)$. In the receiver, we first multiply the result by $\cos\left(\frac{2\pi F_c}{F_s}n\right)$ and apply the low pass filter, and then perform synchronization and channel estimation, followed by decoding the information bits.

Verify that you are able to obtain BER=0 at SNR=100dB.

b. Apply your receiver `RX` with the reference signal stored in `RefInputMod.mat`, which was recorded at SNR=100dB. Verify you obtain BER=0.
What it the estimated value of $\beta/2$?

c. Plot a BER vs. SNR curve as in Question 4c for the SNR points going from -12dB to $+12$dB in 3dB.

3 Using the Audio Channel

In this final section, we modify the system we built, in order to transmit and receive a text message over an acoustic channel. The text is transformed to bits using ASCII with 8 bits per character. You are given the functions `text2bitstream.m` and `bitstream2text.m` which convert ASCII to bits and vice versa. The system is designed to transmit a string of 25 letters, namely, 200 bits, and the number of degrees of freedom is $K = 200$.

You are also given a function `ChannelTXRXAudio` that transmits and receives the acoustic signal. This function should replace `ChannelTXRX` but works with the same inputs and outputs. Please note the following points regarding the implementation:

- If you use a laptop, it is recommended to use its the internal microphone. If you use a desktop you should use an external microphone. If you have a webcam, you can use the microphone on it.
- Try using the internal speaker of your computer. If you do not manage to decode the message, try disabling the default echo cancellation on the computer. Otherwise, you can try using an external speaker (either a Bluetooth speaker connected by physical connector).
- If you have audio enhancement software on you computer, equalizer etc. disable it.
- Be aware that the audio signal is sent only to the left channel. Take that into account if you have two speakers connected to the computer.
- Try to preform this experiment in a quiet environment.

Question 7

a.* Run the script `audioSim.m` with the functions `TX` and `RX` you have constructed in Question 6. See the video clip in moodle for a demonstration. If you are not able to run the audio experiment, this will not effect your grade. But try to run it as it is very cool.
Note that you may have to run the experiment several times until it works. The reason for this is the mismatch in the phase ϕ between the cosine the transmitter uses for up conversion and the cosine the receiver uses for down conversion. See footnote 4. When $\cos(\phi)$ is small, the experiment will fail, and if it is large, it will succeed. The phase ϕ is a random variable, and hence, if we run the experiment enough times (about 3-5 trials should suffice) the experiment will succeed. This problem of phase mismatch can be circumvented almost completely, but for simplicity we ignored this problem in this exercise.

b. You are given the file `RefInputAudio.mat`. The vector `ChannelOutVec` in this file corresponds to running the script `audioSim.m` with some unknown string in `strvec` up to (including) the code-line:

```
ChannelOutVec = ChannelTXRXAudio(config,ChannelInVec);
```

Apply your function `RX` on this vector, and recover `strvec`. What is it? You should get a proper sentence in English.

c. Submit your transmitter function `TX(config,infobits)` and your receiver function `RX(config,ChannelOutVec)` from Question 6. To check that those functions work well, we will run them both over the simulated channel you used in Question 6, and over the audio channel. 50% of the grade of this assignment will be given in accordance to whether or not we can achieve low BER using these functions, provided that the SNR is high enough. If the functions you submitted achieved BER=0 in Question 6.b and recovered a proper sentence in English in item (b) above, they should also work well in our evaluation system and receive the full score. Make sure to verify this before submission.

Be sure that your MATLAB submission in this paragraph is correct: Note that in order for your functions to be properly tested after submission, they have to be in a separate MATLAB files, which have exactly the title of the function that will be called during the test. Namely, for this paragraph, submit two MATLAB files whose titles are 'TX' and 'RX', which contain only the function 'ChannelInVec = TX(config,infobits)' and 'rxbits = RX(config,ChannelOutVec)', respectively. Make sure that you follow these steps, as otherwise your function will not be tested!

A List of attached files

Fie Name	Content
`config.mat`	Contains the configuration structure
`RefInputRect.mat`	Reference input for Question 4. Rectangular pulse
`RefInputSinc.mat`	Reference input for Question 4. Sinc pulse
`RefInputSynch.mat`	Reference input for Question 5
`RefInputMod.mat`	Reference input for Question 6
`RefInputAudio.mat`	Reference input for Question 7
`text2bitstream.m`	a function converting an ASCII string to a binary vector
`bitstream2text.m`	a function converting a binary vector to an ASCII string
`ChannelTXRXaudio.m`	The audio interface channel
`audioSim.m`	A script running the TX, RX and audio interface for Question 7
`CommDemo.mp4`	A video clip of the demo from Question 7