In Lab 4, we sampled a signal to convert it into a bunch of numbers. In the extension (Part C), we saw how these numbers can be converted to a finite number of bits, represented as 0s and 1s. Hence, any signal, and any data, can be thought of simply as a stream of 0s and 1s. In Lab 5, we will transmit a stream of 0s and 1s over a wired communication channel, which can come from any data file on a computer, or a sampled signal as we saw in Lab 4.

**Part A:**

1. To begin, we will just transmit a stream of bits over a wired channel. Construct the $1 \times 1000$ MATLAB vector `data`, which consists of alternating 0s and 1s (write your own code to do this).

2. We will represent a 0 with a signal of magnitude 0, and a 1 with a signal of magnitude 1. Hence, for the `data` from Step 1, we can synthesize a signal consisting of alternating square pulses, based on Worksheet 5.1. Construct the $1 \times 100,000$ MATLAB vector `sig`, that consists of a square pulse of 100 samples each of magnitude 1 if `data(k)=1`, and 100 zeros if `data(k)=0`. Plot the signal `sig` and make sure it looks right (you might need to zoom in to check this).

3. Use `wavplay` to send this signal out through the headphone output of the computer, at 40000 samples a second. What is the data rate you are communicating at, that is, how many bits of information are you transferring per second?

4. Connect the headphone output of the computer to the microphone input of another computer running MATLAB. Use `wavrecord` to record the transmitted waveform, just as you did in Lab 4. You will need to record 100,000 samples, plus an extra 40,000 or so (because of the time delay between hitting enter on one computer and the other one). Call the received signal `rsig`, and plot it. You might need to use `rsig = -rsig;` to take the negative.

5. Does the received signal look similar to the transmitted signal? Can you guess why the received signal looks the way it does? Remember that the received signal looked more or less like the transmitted signal in Lab 4, what is different now?

6. Now add a tunable low pass filter in between the transmitter and receiver, and set the bandwidth to “norm” and the cutoff to a moderate value. Transmit and receive the signal again. Is there any difference to the received signal now?

7. Process the received signal `rsig` to extract the data from it, based on Worksheet 5.2. You will need to estimate the start of the data (you can use `find` to find the location where the data starts, just like in Lab 4) first, and then take the next 100,000 samples from that point onwards. You might notice that the received waveform is now centered
about 0 in the y-axis. This is because of a DC blocker used in the microphone input port of the computer. A good way to decide if a waveform represents the 0 or 1 waveform is to take its average value, and decide that it represents a zero if this average value is less than 0, and 1 otherwise (this is assuming you used \( \texttt{rsig} = -\texttt{rsig}; \) to remove the negative effect if necessary).

8. Reconstruct your received data vector \( \texttt{rdata} \), which has 1000 elements. Test that \( \texttt{rdata} \) also consists of alternating 0s and 1s, just as the original data did. You might want to ignore the first 200 and last 200 data bits, so that any channel peculiarities and charge-up times don’t affect your test.

A quick way to test this is to use \( \text{sum}(\text{abs}(\texttt{data}(200:800)-\texttt{rdata}(200:800))) \), where \( \texttt{data} \) is the original data vector from the transmitter computer (you can generate \( \texttt{data} \) on the receiver computer to be the same as \( \texttt{data} \) from the transmitter computer for testing purposes). This command will count the number of differences between \( \texttt{data} \) and \( \texttt{rdata} \). If you get 600 or 601 errors, you are probably out of sync, so try \( \text{sum}(\text{abs}(\texttt{data}(200:800)-\texttt{rdata}(201:801))) \) instead.

You can plot \( \text{abs}(\texttt{data}(200:800)-\texttt{rdata}(200:800)) \) (or \( \text{abs}(\texttt{data}(200:800)-\texttt{rdata}(201:801)) \)) to see where these errors occur. If the errors occur near the beginning or end of the data stream, you probably shouldn’t really count them because they’re probably the result of sync and startup errors, and not due to the noise in the channel.

9. After the LPF, but before the receiver computer receives the data, add some noise to the signal using the noise generator. Keep increasing the noise, and eventually you should notice that your received data will no longer be a sequence of alternating 0s and 1s.

You might find it increasingly difficult to pinpoint the start location of the data, when the noise is large. It is for this reason that separate sync pulses or synchronized clocks are used in actual digital data transmission. For now, just take the first 100,000 samples of \( \texttt{rsig} \) and extract \( \texttt{rdata} \) from this, if you cannot find the correct data start location. You can drop the first 200 and last 200 bits, and still test if the remaining bits are an alternating pattern of 0s and 1s. When they are not, you know that there are errors in communication.

It is surprising how resilient digital communications can be, you might notice very small error rates even when the noise is rather large, and the human eye can no longer tell whether the signal represents a 0 or a 1.

A regular phone-line MODEM from the dialup internet days (late 90s) communicates over a channel even smaller in bandwidth than the one you are using right now (only about 4 KHz, as opposed to the 20 KHz channel you are using now), but you might have noticed these MODEMs that can do 33.6 kbps, or even 56 kbps. This is because there are much more clever ways of transmitting digital data, some of which we will learn and implement in EE4501/EE4505.
Part B:
In this lab, we will transmit actual digital data (not just an alternating sequence of 0s and 1s), deal with some of the practical issues that come up while doing this, and compare this to doing pure analog transmission (which we did in Lab 4). Several MATLAB commands have been written for Lab 5B to make things easier, remember that these are not standard MATLAB functions.

Information on related MATLAB commands:

- Unzip “matlab-files.zip” (from Moodle) into your Documents/MATLAB folder (overwriting any existing files). Don’t worry about code in the files startrx, stoprx, rxtrigcallback, rxsampcallback, transmit_signal and txdonecallback. These are meant for transmitting/receiving signals over the soundcard, which you are already familiar with from Lab 4. However, make sure that you are familiar with the code in deocde_polar_rz and extract_timing_rz.

- You can use edit <filename> to view the code for any of these functions in MATLAB. You can also type help <function name> to view some details about how to use these functions.

- numbers2bits and bits2numbers are convenient utilities for converting a decimal numbers into binary digits, and vice versa, respectively. encode_polar_rz is an efficient implementation similar to the code you wrote in Worksheet 5.1. You don’t need to understand the code for these 3functions, but it is useful to understand what they do.

- The command startrx(total_samples_to_receive, threshold) is used to start the receiver (use it at the receiver computer). The microphone port will need to be connected for this. total_samples_to_receive is the total number of samples to be received (there will be 40,000 samples received per second, by default). threshold is the smallest signal value the receiver will wait for, before starting to receive data. By default, this is 0.2v (if you omit this value), which means that the receiver will automatically remove any silence or noise present in the received signal (below 0.2v) before your data starts transmitting (we used the find command to do the same thing in previous labs).

- After you start receiving, you can continue to use MATLAB.\(^1\) When finished, the receiver will store the received signal in the variable rsig, which you can access using the command load rsig. Use stoprx to abort the receiver, in case you need to do that.

- Use transmit_signal(sig) to transmit the signal stored in vector sig (at 40,000 samples/sec, by default). The signal is automatically converted to double, and scaled to be between -1 and 1. Just like the receiver, you can continue to use MATLAB commands while transmitting. Use stoptx to abort transmission.

\(^1\)In case you are unable to check out a license for the data acquisition toolbox, startrx, transmit_signal and so on will fall back to standard MATLAB commands such as wavread and wavrecord. In this case, you will have to wait for MATLAB to finish recording/playing before continuing to use it.
• The Polar-RZ standard encodes a binary “1” using the waveform in Figure 1, and a binary zero using the waveform in Figure 2. The fact that we’re using inverse waveforms for 1 and 0 gives it the name “polar”. Also notice that for either case, the waveform “returns to the zero level” halfway through, hence the term “RZ”, or return-to-zero.

In Lab 5A, we used a positive voltage for 1, and zero voltage for 0. This would be a Unipolar-NRZ standard (NRZ stands for non-return-to-zero).

• RZ standards are very useful because they include a transition for every bit, irrespective of whether it is 0 or 1. If we take the absolute value of the RZ signal, we will generate a clock signal that has a pulse for every data bit. This is useful during decoding, as we will see.

• The downside to using RZ is that we have an extra transition every bit period, which means we are essentially doubling our frequency, which doubles the bandwidth we would need to communicate.
• Use `encode_polar_rz(bits, x)` to convert any series of bits into a signal using this standard. `x` is the number of samples used to encode each bit (as marked in the figures). In Worksheet 5.1, we used `x = 100` samples to encode each bit. The `encode_polar_rz` function has the same functionality as Worksheet 5.1, but is for the Polar-RZ standard.

• `decode_polar_rz(rsig, x)` does the reverse, that is, given a Polar-RZ signal (or something that closely resembles a Polar-RZ signal), it decides if each piece of the waveform represents a 0 or 1. This functionality is similar to what you wrote in Worksheet 5.2. We will compare the mean of the first `x/2` samples (or the sum) to the mean of the next `x/2` samples, and conclude that we are looking at a waveform representing 1 if it is greater, and 0 otherwise.

Note on the lab report for Lab 5B: Include comments describing how the `decode_polar_rz` and `extract_timing_rz` functions work (what each line does, what the variables stand for, etc.). Alternatively, write your own code/pseudocode to implement the same functionality. This is worth 30% of the Lab 5B grade, since there are no extensions for credit. You won’t receive credit for merely duplicating what is already described in this guide.

Guidelines for the main experiment:

1. Use `img = rgb2gray(imread('test3.jpg'))`; to load a test image and convert it to gray-scale, as you did in previous labs. We’re using `test3.jpg` for the examples here (although you can use any image you like), make sure it is in your MATLAB folder (else, download it from Moodle).

2. Figure out how many samples a direct analog transmission of this image is going to have (you can do a trial transmission to figure this out). `transmit_signal(img)` will transmit the image directly as an analog signal (without encoding or quantization). You can use this command at the transmitter, and `daqscope` at the receiver, to check that you can receive a signal that is large enough. You want a signal larger than about .5v. Increase volume levels if it isn’t large enough, but not so large that it clips. It is always to better to increase amplification closer to the transmitter (lesser chance of amplify noise along the way).

   Next, start your receiver using `startrx`, then start transmitting. When finished, load your received signal, reshape it to the dimensions of your image, and use `imshow` to view it (remember to use auto-balancing while viewing the image). It should look similar to what you have already seen in Lab 4 (there should be no need to cut off the beginning or end of the received signal, or take the negative, as all that will be done automatically).

3. Load the same image on the receiver computer, and compute the correlation coefficient between the received image, and the original image. You will have to convert `img` to `double` at the receiver computer, before doing this. First compute the covariance:
   
   ```matlab
cov = mean(img(:).*rimg(:)) - mean(img(:)).*mean(rimg(:));
```

   and therefore the correlation coefficient:
corr = cov/sqrt(var(img(:)))/sqrt(var(rimg(:))); 
which is a number between -1 and 1. The higher the absolute value of the correlation 
coefficient (the maximum is 1), the more similar is the received image is to the original 
image. Note down these results, you might need it for comparison later.

4. Next, we will get a little familiar with digital communications. Use 
sig = encode_polar_rz(bits, x); to create a Polar-RZ waveform for a short se-
quence of bits (you can use any sequence, say, bits = [1 0 1 1]). Try x = 50, 100. 
Use plot(sig) to view the generated waveform.

5. 4-5 bits might be too short for transmission, so we’ll add some padding to the data. 
Define padding = ones(1, 256); and append this before the beginning and end of 
your data bits, before converting it into a Polar-RZ signal. This will give you a signal 
that is appreciably long in duration. Transmit and receive this signal. 
How does x affect the transmission bitrate (or how long the transmission takes)?

6. Let rsig be the received signal (you can view it using plot(rsig)). You can decode 
rsig and recover the bits using decode_polar_rz (and store the decoded bits in a 
variable called rbits). Remember to use the same value for x as you used in the 
transmitter.

Then, use rbits = rbits(257:end-256) to remove the first and last 256 bits, to re-
move the padding. You should be left with the bits you originally transmitted (hope-
fully, with no errors).

7. Now we’ve figured out how to communicate bits. The function numbers2bits can be 
used to convert any bunch of numbers to a stream of bits, so we can communicate any 
set of numbers now.

Instead of a random bunch of bits, we can make our data a little more meaningful. For 
example, try this (use any message you like):
message = 'This is a simple text message to nobody'
 nos = double(message)
This will convert each letter in your message into its ASCII code (a number between 
0 and 255).

Since each value in nos is between 0 and 255, we can quantize it using 8 bits per value 
with no loss (8 bits can represent $2^8 = 256$ different numbers):
bits = numbers2bits(nos, 8);

Now, convert the bits into a Polar-RZ signal and transmit/receive it. Decode these bits 
at the receiver, and use rnos = bits2numbers(rbits); to convert each group of 8 
bits into a number. You can convert rnos to text using char(rnos') (or char(rnos), 
depending on whether rnos is a column or row vector). You might notice that the 
first/last few letters have some errors, you can use padding to avoid this.

8. Now, we’ll revisit our test3.jpg image. Load this into img as before, and use bits = 
numbers2bits(img, 8) to convert the image into a binary stream. To check that this
is working, look at \texttt{img(1)} and \texttt{bits(1:8)}, and these should be decimal and binary representations of the same number, respectively.

You can then encode this into a Polar-RZ waveform, which we can transmit. If you use $x = 100$, you might run out of memory. $x = 20$ might work, how long do you think it will take to transmit your entire image using this value for $x$?

We’ll try to push our system, and transmit using $x = 8$. Figure out the corresponding number of received samples, and receive this signal.

9. Decode this signal as you did before to recover \texttt{rbits}. Next, use \texttt{bits2numbers} to convert the bits to numbers, the output of which you can reshape and use \texttt{imshow} with (remember to use auto-balancing while viewing the image). You might have realized that some of this can be more easily done if you write out your commands in a .m script file and save it.

10. You can count the number of errors you have: $\text{sum(abs(bits - rbits))}$, where \texttt{bits} is generated at the receiver the same way it was generated at the transmitter. Were you able to receive your signal successfully? Does your received image look convincing?

11. If you have trouble, you might be able to see that you’re probably being thrown out of synchronization. The first part of your image might be coming across fine, but slowly, sooner or later, the decoder goes out of sync. To fix this, we will “extract” the timing signal from the RZ digital signal.

Figure 3 shows an example received waveform (with $x = 10$ in the figure). We will define a \texttt{timing} vector as being the start times in \texttt{rsig} for representing each bit. For instance, times 1 to 10 is used for the first bit, 11 to 20 for the second and so on. So \texttt{timing} = [1 11 21 ...].

![Figure 3:](image)

Sometimes, we can build hardware to control our sampling times and make sure we are in sync, but we have no control over the exact sample times here (it just happens uniformly 40,000 times every second). However, we can take several samples for each bit (in our notation, we have $x$ samples for each bit), and somehow estimate any timing errors or delay. In any case, we have to do some form of timing extraction.
Here is an example method, assuming \( n \) bits in total, for generating the timing vector of length \( n \):

Step 1: Take the absolute value of \( rsig \), which generates an approximate clock signal.

Step 2: For the \( k \)-th bit (\( k = 1, \ldots, n \)), take the sum of values from the time \( k+m \), to the time \( k+m+(x/2)-1 \), for some choice of \( m \). Pick the value of \( m \) such that this sum is maximum, where \( m \) is between, for example, \(-x/4\) and \(+x/4\).

Step 3: Use these start times (stored in the timing vector, in our case) for each bit during decoding.

The sum of the values from \( k+m \) to \( k+m+(x/2)-1 \) is maximum only when we start and end with the beginning of the pulse, and end with the end of the pulse. If we are not aligned with the pulse, the sum will be less than maximum. Hence, we can find a value for \( m \) that will let us “align” with our signal, and this will tell us how much our signal has shifted to the left/right. In this case, the \( rsig \) values from position \( k + m \) and the next \( x \) samples will be used to decode the \( k \)-th bit. Using our notation, the above implies that \( \text{timing}(k) = t+m \).

An example: starting at the location 11 for \( k = 2 \) (in Figure 3), we can look at the sum of \( \text{abs}(rsig) \) values from times 9 to 13 (the interval length is \( x/2 = 5 \)), the sum of the values from 10 to 14, 11 to 15, 12 to 16 and 13 to 17. Let us say the sum of the values from 12 to 16 was the highest in this lot. Then we will consider \( rsig(12:21) \) to be the received signal for the second bit (\( k = 2 \)), and will compare \( \text{mean}(rsig(12:16)) \) to \( \text{mean}(rsig(17:21)) \), and conclude that the signal represents a 1 if it is greater, and 0 otherwise.

Thus, we can estimate delays or slight timing differences and maintain synchronization. Use \( \text{timing} = \text{extract\_timing\_rz}(rsig, x) \); to perform the above operations, and extract the timing vector. Look at the first 10 values of \( \text{timing} \) to make sure the algorithm is working, and the result makes sense.

12. Decode \( rsig \) now, but with timing information. Use \( rbits = \text{decode\_polar\_rz}(rsig, x, \text{timing}) \) to do this. Then test for the number of errors. Is it any less?

13. You can plot the location of your errors using \( \text{stem}(|\text{bits} - \text{rbits}|) \). If there are only a few, you probably won’t even notice it on the image. If the errors occur near the beginning or the end, you should be able to avoid that by using some padding.

Convert your bits to numbers, reshape and plot the resulting image. Does it look better than the default timing version? Does it look better than the analog signal you transmitted earlier?

Compute the correlation for the received image and the original. If you use padding, there is no reason why you can’t get the correlation to 1.

14. Transmit and receive the bits for the image using the same laptop (running 2 copies of MATLAB), using the same \( x = 8 \). You might find that you don’t need to extract timing information. This is because the microphone and soundcard are on the same
circuit board and most likely share the same clock signal, and are hence automatically synchronized. In wired, peripheral (such as USB) or chip-to-chip communications, we can usually afford a separate line for the clock signal. We might not have that luxury in practice while communicating over a distance, however.

**Note:** If you use very short data sequences (such as the short text message), you don’t really need to worry about synchronization. You can simply re-sync between short bursts of data. For longer bursts of data, which makes a lot of practical sense, you’d need to be careful about timing.

Further, increasing x will again make your receiver much more resilient to timing errors. Hence, communicating at higher data rates will make timing extraction increasingly important.

15. Another aspect to experiment with, if you have the time, is to add some noise to your transmitted signal. The analog signal will degrade in quality, while the digital signal might get a few bits in error here and there, but will still produce a very clean image. This is one of the main advantages of digital communications.

However, when the noise is very high, you might still be able to make out the noise-corrupted analog image, more or less, while the image after digital transmission is probably an unrecognizable mess. When noise levels are large, analog is, in a sense, superior to digital.