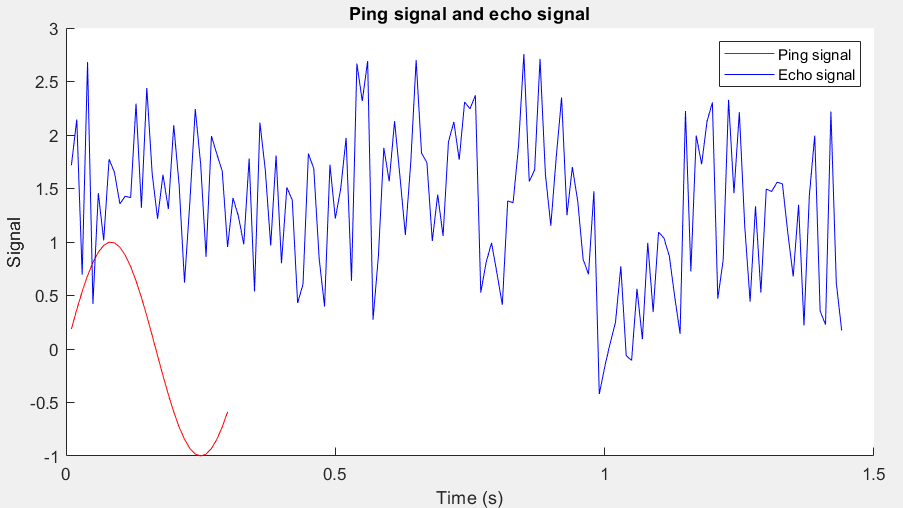
Signals and Systems Project 1 Report

Jingguo LIang (jliang35@jhu.edu)

Chang Yan (cyan13@jhu.edu)

**Part 1 Sonar systems**

**1.1** we first load the signal and plotted the ping signal and echo signal as follow:



**1.2** Steps:

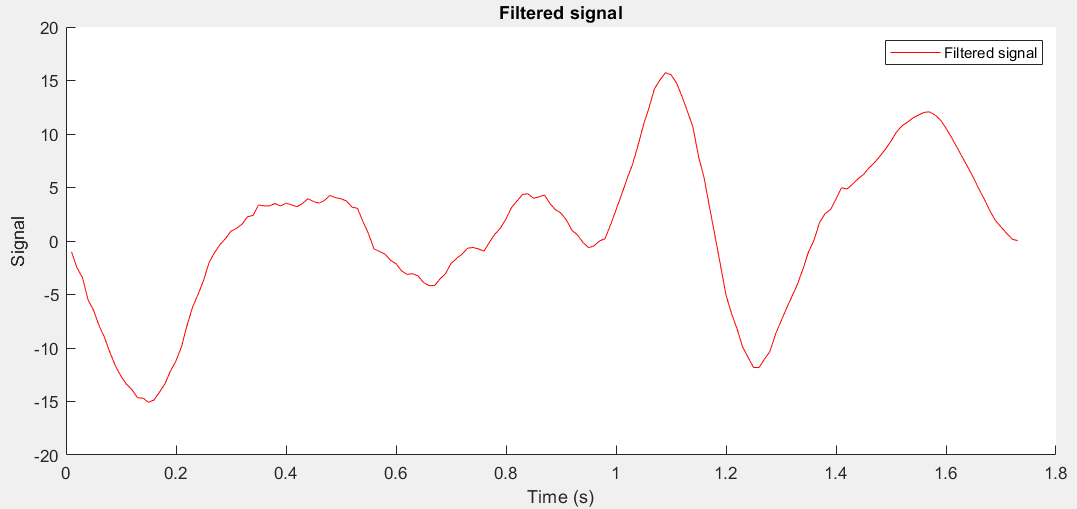
**Step1: Filter the signal using pattern match filter.**

The impulse response of the system is:

So, we need to flip the signal before convolution, and the result needs to be properly shifted by T before calculation time and distance.

We first use thefliplr function in Matlab to flip the ping signal, and then use the conv function to do the convolution between the flipped ping signal and the echo signal.

The filtered signal is shown below:

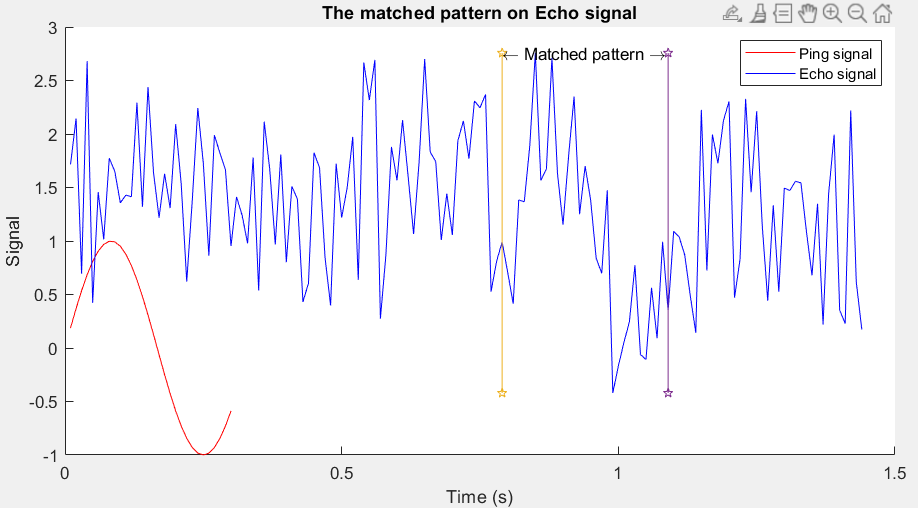


**Step2: Find the maximum point at the filtered signal.**

As said in the instruction, the filtering operation should achieve its maximum value when the pattern matches best. Thus, we use the max function of Matlab to find the index of the maximum point.

However, as we mentioned before, we still need to shift by T to get the actual time difference. This is because the maximum point is at the end of the matched pattern, but the end of the ping signal is at time T. The T can be determined from the length of the ping signal as it is just one period, and we subtracted T from the maximum index to get the time difference.

To better view and verify our result, we draw two vertical lines on the ping and echo signal plot to indicate the start and the end of the matched pattern:



As we can see, the matched pattern looks correct.

**Step3: Calculate the distance**

First, we convert the index value to second, which is just divided by sample rate:

Then, we multiply the time by the speed, and divide it by 2 as the signal go forward and back which is double distance:

The result is:

**Part 2: Digital Message Reception**

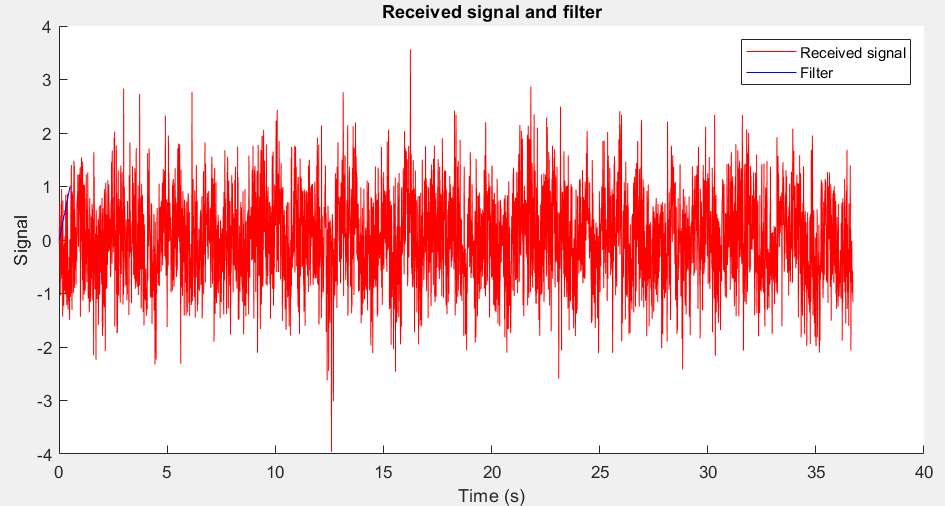
**2.1** Design of the function decode

The function takes the received signal and sample frequency as input, as well as an optional input of pulse signal, and returns the received message:

**Step1: Input processing**

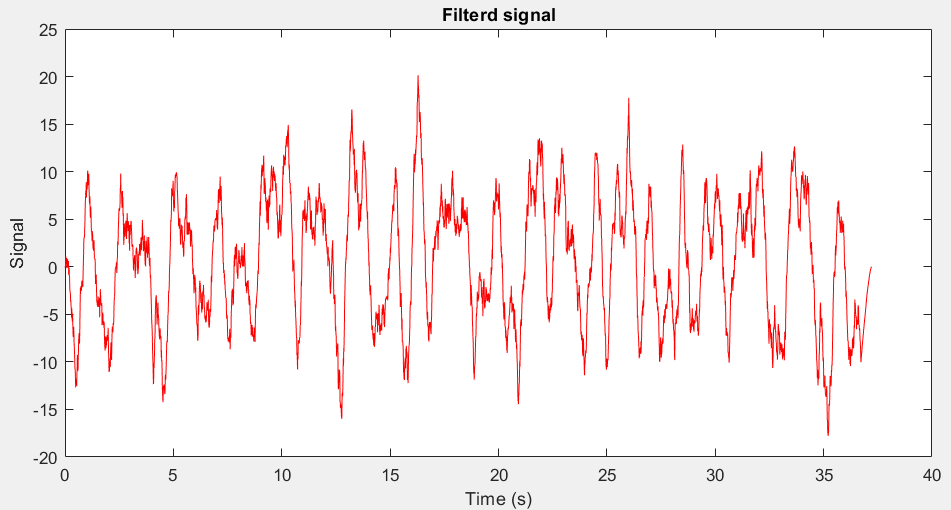
As we have one optional argument, I use the varargin to handle the optional input. If we get nargin = 3, we will just accept the third argument as pulse signal, but if not, we will create a 0.3s square wave using the sample frequency given.

The received signal and the pulse signal of one of the signals is shown below: (for signal 3)



**Step2: Filtering**

We use the same method as we did in part 1 to filter the signal: using fliplr and cov. Also, when doing all plots, we always use the sampling frequency to convert the x axis to the unit of second. The plot of the filtered signal is:



**Step3: Determine the 1,0 logical sequence.**

We first calculated the number of digits in the sequence, by dividing length of signal by the length of the pulse. We use floor for safety.

Then, we use N to do a for loop to determine each of the logical values. If Filtered(n\*T) > 0, we will set it to 1. (n is the looping variable) Otherwise, we set it to 0. Again, the T value is from the length of the pulse signal.

The result 1,0 logical digits are stored in an array of length N.

**2.2 Message output**

**Step1: Reading Ascii file**

We use fopen and textscan to read the ascii file and store it in a Matlab cell structure. One column contains all the characters (all ends with “:”, but will be removed later in comparison), and another column contains the Ascii code in decimal form (I used %b when reading it, so they are now all in decimal).

The stored structure looks like the following:

|  |  |
| --- | --- |
| : | 32 |
| !: | 33 |
| ": | 34 |

**Step2: Convert logical array to message.**

We first separate the logical array from **2.1** in to chunks of length 8 and use floor for safety. This separation is represented by the outer for loop, controlled by the length of the message L:

Then, in the inner for loop, we calculate the Ascii number in decimal represented by each 8-bit chunk use the binary to decimal formular:

Then, we search this number in the second column of the cell structure (where the decimal form of Ascii is stored) and get the corresponding string in first column. Then, we append the first character in the string to the message string. This method ignores the “:” at the end of each string, and when the length of the string is 1 (with only a “:”) we know it should be a space, so we append a space to message string.

In the end, the function returns the message string.

**Results of decoding:**

We use the three signals to test the decode function. We create a 0.5s positive triangle wave as the optional pulse input, and just load three different mat files to get the input signals. The sample frequency is 100. The result is shown below:

**Signal1:** SOS

**Signal2:** Help!

**Signal3:** Nevermind

**Part 3: Digital Message Transmission**

For this part, we implemented two functions: encode and transmit\_noise. The function encode takes an input as string which is the message that will be sent, and a pulse signal that represents a ‘1’. It will return the signal that represents the message. The function transmit\_noise takes as input a string which is the message that will be sent, the pulse signal that represents a ‘1’, the sample rate, and the level of noise to be added to the signal. It constructs the signal by calling encode and add noise to it. Then it will call decode and try to retrieve the message from the noisified signal. Finally, it will compare if the decoded message matches the original one.

**Encode：**First, compute the length of the signal, allocate space for the signal vector, and pre-set the signal to be 0. Then read in the Ascii file using the same method as in part 2, but this time read in all as strings. Reading the binary as string enables us to easily extract bits and convert the ‘1’s and ‘0’s from characters to values. Extract characters in the message one by one, and using the character as key, we can index into the Ascii cells and locate the converted binary. If there is a character in message that cannot be converted, the encoding process will halt immediately, and a signal with all zeros will be returned. The next step is to convert the binary to actual signal strength. The binary is also in string form. Again, extract characters from this string, and if it is a 1, append the pulse signal directly to the output. If it is a 0, negate the pulse signal and append it to the output.

**Transmit\_noise**: The first step is to encode the message by calling function encode. Then a random noise with the same length as the signal will be generated with function randn. The noise is scaled by noise level to control its magnitude. Then two graphs will be plotted. The first one contains the pulse signal and the encoded message without noise. The second one shows the noisified signal, which is the encoded message signal plus the noise signal. Then the decode function in part 2 is called on the noisified signal, which will return the decoded message. The decoded message is compared with the original message, and tested if they match each other.

**3.1.** For 3.1 and 3.2 we will use sample rate 100 samples/s and use rectangular pulse with duration 0.2s to represent a binary 1. We will try to send the message “Acknowledged”. The noise level will be set to 0.5. Enter the following command in command window:

>> pulse = ones(1, 21);

>> msg = 'Acknowledged';

>> transmit\_noise(msg, pulse, 100, 0.5);

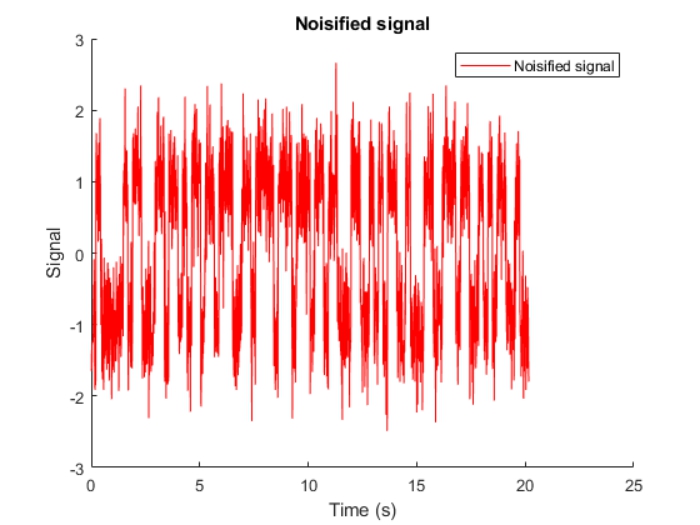
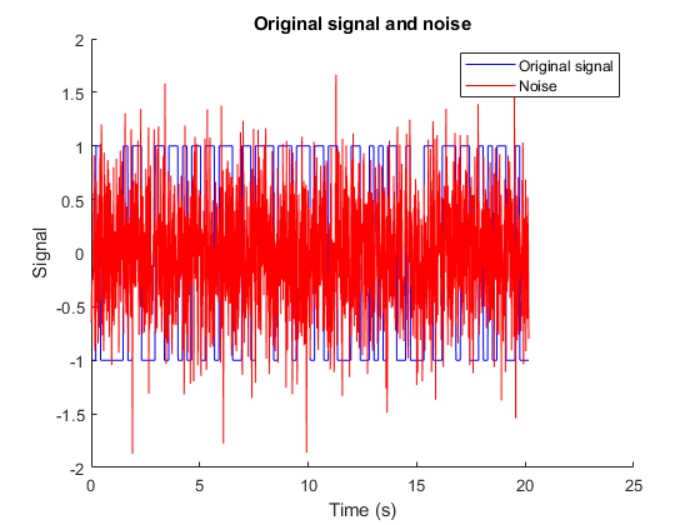
We have the following output from the command window:

Original message is: Acknowledged

Decoded message is: Acknowledged

Two messages match.

We also have the following graphs that shows the original and the noisified signal:



**3.2.** From the command window output we see that the decoded message is “Acknowledged”, and it matches the message that is sent.

**3.3.** First, we will construct the four types of pulse signals. Take sample rate 100, and all signals will have duration 0.2s. Enter the following command in command window:

>> pulse1 = ones(1, 21);

>> pulse2 = 0:0.05:1;

>> pulse3 = sin((0:20)\*0.1\*pi);

>> pulse4 = [1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0];

Then, we will call the function transmit\_noise using those pulses and the message “Acknowledged”, with the noise level increasing 0.1 every step.

At noise level 2.2, type1 pulse signal (rectangular) begin to unmatch.

At noise level 1.2, type2 pulse signal (triangular) begin to unmatch.

At noise level 1.3, type3 pulse signal (sinusoidal) begin to unmatch.

At noise level 1.3, type4 pulse signal begin to unmatch.

Type1 pulse (rectangular pulse) is most robust to noises.