Signals and Systems – Spring 2025

Lab Assignment #1

Issued: March 20, 2025 Due: April 17, 2025

You should hand in the Matlab code (.m files), graphics, audio files and a brief description of your reasoning as well as comments if any. Please make sure that your Matlab code can be run on Matlab R2012b or higher version. You should pack all of your files into a .rar or .zip file, titled as xxxxxxx(your student ID) xxxx(your name) Lab 1, and then submit it by sending an email to yangpujing@zju.edu.cn before 11:59pm of the due day.

Problem 1

1. Making Continuous-Time Pole-Zero Diagrams
In this excercise you will learn how to display the poles and zeros of a rational system function $H(s)$ in a pole-zero diagram. The poles and zeros of a rational system function can be computed using the function `roots`. For example, for the LTI system with system function

$$H(s) = \frac{s-1}{s^2+3s+2},$$

the poles and zeros can be computed by executing

```
>> b = [1 -1];
>> a = [1 3 2];
>> zs = roots(b)
zs =
      1
>> ps = roots(a)
ps =
     -2
     -1
```

A simple pole-zero plot can be made by placing an 'x' at each pole location and an 'o' at each zero location in the complex s-plane, i.e.,

```
>> plot(real(zs),imag(zs),'o');
>> hold on
>> plot(real(ps),imag(ps),'x');
>> grid
>> axis([-3 3 -3 3]);
```

The function `grid` places a grid on the plot and `axis` sets the range of the axes on the plot.

(a). Each of the following system functions corresponds to a stable LTI system. Use `roots` to find the poles and zeros of each system function and make an appropriately labeled pole-zero diagram using `plot` as shown above.

$$(i)\ \ H(s) = \frac{s+5}{s^2+2s+3}$$
$$(ii)\ \ H(s) = \frac{2s^2+5s+12}{s^2+2s+10}$$
$$(iii)\ \ H(s) = \frac{2s^2+5s+12}{(s^2+2s+10)(s+2)}$$

Several different signals can have the same rational expression for their Laplace transform while having different regions of convergence. For example the causal and anticausal LTI systems with impulse responses

$$h_c(t) = e^{-\alpha t}u(t), \qquad h_{ac}(t) = -e^{-\alpha t}u(-t),$$

have a rational system function with the same numerator and denominator polynomials,

$$H_c(s) = \frac{1}{s+\alpha}, \mathfrak{Re}(s) > -\alpha,$$
$$H_{ac}(s) = \frac{1}{s+\alpha}, \mathfrak{Re}(s) < -\alpha.$$

However, they have different system functions, since their regions of convergence are different.

(b). For each of the rational expressions in Part (a), determine the ROC corresponding to the stable system.
(c). For the causal LTI system whose input and output satisfy the following differential equation

$$\frac{dy(t)}{dt} - 3y(t) = \frac{d^2x(t)}{dt^2} + 2\frac{dx(t)}{t} + 5x(t),$$

find the poles and zeros of the system and make an appropriately labeled pole-zero diagram.

For the exercises in this problem, you will need to use the function `pzplot`, which is contained in the Computer Explorations Toolbox. The function `pzplot` plots a pole-zero diagram for the LTI system whose numerator and denominator polynomials have the coefficients in the vectors `b` and `a`, respectively. The function will return the values of the poles and zeros in addition to making the plot. An optional argument, `ROC`, can be used to indicate the region of convergence on the diagram. By selecting `ROC` to be a point within the region of convergence of the system, `pzplot` will appropriately label the region of convergence of the system. For example, try executing,

```
>> b = [1 -1];
>> a = [1 3 1];
>> [ps,zsl=pzplot(b,a,1);
>> [ps,zs]=pzplot(b,a,-2);
```

(d). Explain how `pzplot` can determine the region of convergence of a rational transform from knowledge of a single point within the ROC.

2. Making Discrete-Time Pole-Zero Diagrams

In this excercise you will learn how to display the poles and zeros of a discrete-time rational system function $H(z)$ in a pole-zero diagram. The poles and zeros of a rational system function can be computed using `roots`. The function `roots` requires the coefficient vector to be in descending order of the independent variable. For example, consider the LTI system with system function

$$H(z) = \frac{z^2 - z}{z^2 + 3z + 2}. \tag{1}$$

The poles and zeros can be computed by executing

```
>> b = [1 -1 0];
>> a = [1 3 2];
>> zs = roots(b)
zs =
       0
       1
>> ps = roots(a)
ps =
      -2
      -1
```

It is often desirable to write discrete-time system functions in terms of increasing order of z^{-1}. The coefficients of these polynomials are easily obtained from the linear constant-coefficient difference equation and are also in the form that `filter` or `freqz` requires. However, if the numerator and denominator polynomials do not have the same order, some poles or zeros at $z = 0$ may be overlooked. For example, Eq. (1), could be rewritten as

$$H(z) = \frac{1 - z^{-1}}{1 + 3z^{-1} + 2z^{-2}} \tag{2}$$

If you were to obtain the coefficients from Eq. (2), you would get the following

```
>> b = [1 -1];
>> a = [1 3 2];
>> zs = roots(b)
zs =
       1
>> ps = roots(a)
ps =
      -2
      -1
```

Note that the zero at $z = 0$ does not appear here. In order to find the complete set of poles and zeros when working with a system function in terms of z^{-1}, you must append zeros to the coefficient vector for the lower-order polynomial such that the coefficient vectors are the same length.
For the exercises in this problem, you will need the M-file `dpzplot.m`, which is in the Computer Explorations Toolbox. For convenience. The function `dpzplot(b,a)` plots the poles and zeros of discrete-time systems. The inputs to `dpzplot` are in the same format as `filter`, and `dpzplot` will automatically append an appropriate number of zeros to `a` or to `b` if the numerator and denominator polynomials are not of the same order. Also, `dpzplot` will include the unit circle in the plot as well as an indication of the number of poles or zeros at the origin — if there are more than one.

(a). Use `dpzplot` to plot the poles and zeros for $H(z)$ in Eq. (1).
(b). Use `dpzplot` to plot the poles and zeros for a filter which satisfies the difference equation

$$y[n] + y[n-1] + 0.5y[n-2] = x[n].$$

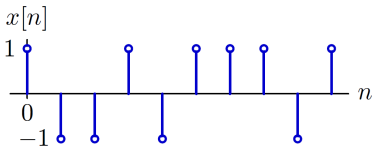
(c). Use `dpzplot` to plot the poles and zeros for a filter which satisfies the difference equation

$$y[n] - 1.25y[n-1] + 0.75y[n-2] - 0.125y[n-3] = x[n] + 0.5x[n-1].$$

Problem 2

1. Smiley

Consider the sequence of 1's and -1's shown below as $x[n]$.



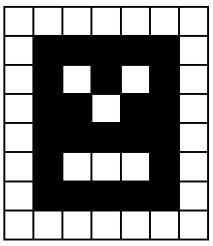
In this $x[n]$, there is a single occurrence of the pattern -1, -1, 1. It occurs starting at $n = 1$ and ending at $n = 3$. One method to automatically locate particular patterns of this type is called "matched filtering." Let $p[n]$ represent the pattern of interest flipped about $n = 0$. Then instances of the pattern can be found by finding the times when $y[n] = (p * x)[n]$ is maximized.

(a). Determine a matched filter $p[n]$ that will find occurrences of the sequence: -1, -1, 1. Design $p[n]$ so that $(p*x)[n]$ has maxima at points that are centered on the desired pattern, i.e., at $n = 2$ for the sequence above.

The same approach can be used to find patterns in pictures by generalizing the convolution operator to two dimensions:

$$y[n, m] = (x * p)[n, m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} x[k, l]p[n - k, m - l].$$

A file called `findsmiley.jpg` contains a random pattern of white pixels (coded as 255) and black pixels (0) as well as a single instance of the following smiley face:



You may need to use the function `imread` to load an image to the matlab workspace. `A = imread(filename)` reads a grayscale or color image from the file specified by the string `filename`. If the file is not in the current folder, or in a folder on the matlab path, specify the full pathname. Use `help imread` for more information.

(b). Find the row and column of findsmiley that corresponds to smiley's nose. Note: matched filtering will work best if matching white pixels AND matching black pixels contribute positively to the answer. For that reason, 255 and 0 may not be the optimum choices for the values associated with white and black pixels.

(c). One advantage of the matched filter method is that it works even when the signal contains some noise. One can make a noisy version of `findsmiley` using function `normrnd`. `R = normrnd(mu,sigma,m,n)` generates random numbers from the normal distribution (Gaussian distribution) with mean parameter `mu` and standard deviation parameter `sigma`, where scalars `m` and `n` are the row and column dimensions of `R`.

Problem 3
The purpose of this problem is to construct physically meaningful signals mathematically in MATLAB using your knowledge of signals. You will gain some understanding of the physical meaning of the signals you construct by using audio playback. We also hope that you have fun!
For this lab, you will need to use a PC with a sound card installed in it. Most computing facilities are equipped with sound cards. You will also need a set of headphones to listen to your musical creation.

1. Use the Sounds in Matlab
(1). Prerequisite Knowledge: About Function File
In this lab, you will learn about matlab function files, which are similar to the scripts. Function files also have the file extension `*.m` and thus are also called 'M-Files'. You will create a number of functions for manipulating sound signals, and use them to create a groove or short song.
The difference between a function and a script is that a script is simply a sequence of commands to be run, while a function can take parameters and return values. A function is therefore a generalized and flexible script.
A side effect of functions is that any variables created within the function are not available after the function has finished running. That is, the variables inside a function are only active within the scope of the function, and simply don't exist after the function exits. By contrast, the variables in a script are added to your workspace and you can look at them anytime afterward.
See that function files start with something like:

```
function[output1,output2] = FunctionName(parameter1,parameter2)
```

This function MUST be written in a file called `FunctionName.m` (use a descriptive name, obviously). Then this first line means that the function will accept some parameters, and will output whatever is in outputVariable when it's done running. If `output1` isn't defined in the function, then it won't return anything. Also note that the `output1` is only defined within the scope of the function. You can't use this value unless you explicitly capture the function's return value at the matlab prompt, like so:

```
>> x = FunctionName(param1,param2)
```

The first set of comments (immediately after the first line in the function) is what will be displayed on screen if you type `help FunctionName`. This should be a general description of your function and its syntax.
To learn more, read the following help pages:

```
>> help function
```

```
>> help script
```

To create function files, you need to use a text editor such as Notepad on a Windows PC or emacs on Linux and Mac computers. Matlab also has an internal editor that you can use within the matlab GUI. You can start the editor by clicking on an M-file within the matlab file browser. All of these editors are standard tools and will produce plaintext files that matlab can read. ***Make sure that your new file is in matlab's working directory, or else you won't be able to run it.***

(2). Function Files
We have prepared the sound samples for you. Remember to save them to your working directory. Use `wavread` to load `*.wav` files, and use `load` to load `*.mat` files. Note that `load` will instantiate any variables that are named inside a `.mat` file, but you need to explicitly capture the output of `wavread` via something like:

```
>>y = wavread('SoundFileName');
```

You may need to read the help pages for these functions. Sounds are represented digitally on a computer, which means that the analog signal is sampled at fixed time intervals and only these samples are stored. You'll learn more about this later. For now, you just need to keep track of the time interval T_s (or, equivalently, $F_s = 1/T_s$, which is the sampling rate) when playing sounds in matlab.
To learn how the time domain signal sounds, use the sound command to play it. You must specify the playback sample rate $\boldsymbol{F_s}$, which ought to be the same as the rate at which the sound was originally sampled. ***The sound files***

provided for this lab have a sample rate of 8000Hz. For example, to play a hypothetical sound called bell, you would enter:

```
>> Fs=8000;
>> sound(bell,Fs);
```

If your value for F_s is different from the sampling rate, then you will effectively be performing time scaling. An F_s that is lower than the sampling rate will slow down the sound, and a larger F_s one will speed it up.
You also need to know the sampling rate in order to accurately plot sounds against time. As you may know, `plot(t,y)` only works with vectors that are the same length. If you know the sampling frequency for `y`, and also how many samples are in `y`, then you can figure out how long `y` is, in seconds. You can now easily construct an appropriate time vector against which to plot sound `y`. You will need to construct a new time vector to plot each sound, unless you have two sounds that have the same duration.

Exercise 1: Plot each of the sound samples we have provided for you, and predict how the volume will change over time when you play them.
Exercise 2: Play each of the sound samples. How good were your predictions?

When working with sound in matlab, it is important to remember that the values of the audio signals are in the range [-1,1]. Anything out of that range will be clipped to 1 or -1 when you play the sound, which will distort it. Keep this in mind when you write your functions! Your functions should expect inputs with values in the range [-1,1] and produce outputs within that same range.

You need to write functions that can modify sound signals before you can combine them to create your musical groove. After all, wouldn't it be boring to make a tune that consists of exactly one note over and over again? You will soon create functions that can time-scale, reverse, delay, fade, and repeat sounds. You will also write a simple mixer that can layer two sounds on top of each other. Remember that the function files you create must have comments. However, instead of adding a header to your function files, add the information as a footer (at the very end of the file). This way the first lines of the function contain the function definition and help comments.

A. Time-reverse & Timescale
Matlab has many built-in functions. Two closely-related functions that you might use in this lab are `fliplr` and `flipud`, which allow you to time-reverse a signal in one operation. Read `help fliplr` and `flipud` for details.

Exercise 3: Time-reverse one of the sounds that you downloaded. Plot both the original and reversed signals, and play them.

Save the function file `timescale.m` into your working directory. Read the M-file to see how to run it.

Exercise 4: Use the `timescale` function on a sound file to both speed up and slow down one of the sounds. Why does the pitch change?

B. Fader
Save the function file `fade.m` (it's incomplete) into your working directory. Read the code, including comments. Now enter `help fade` at the matlab prompt. If you saved the file correctly, you will see the help text from the M-file in response to `help fade`. Notice that you can use the new command that you just added to matlab as if it were a built-in function.
Enter the following at the matlab prompt:

```
>> time = 0:0.01:1;
>> y = cos(time.*pi.*0.25);
>> plot(time,fade(y));
```

You can see in the plot that fade does fade out the cosine wave. You can use this function on audio signals as well. This function works fine, but offers the user no control. Therefore, your assignment is to edit the fade function so that you can adjust the slope of the ramp that fades the signal. In order to do so, notice in the code that there's an

unused parameter named `level`. The variable `level` should be the final volume level after fading, ***as a percentage of the original input.***
You must ensure that `level` only takes on values between 0 and 1, because you can't fade to less than 0% nor more than 100% of the original input volume. You also need to handle the case where the user does not specify the value of level. The desired behavior in this case is specified in the code. You may find the exist command helpful.

Exercise 5: Modify the fade function to use the parameter `level` as described above. Remember to comment your code, and add a footer with your personal information. Write down what you predict will happen if you fade the cosine wave `y` with `level` set to each of the following: 0, 1, 0.25, -2, 100. Now show what happens when you run `fade` with each of those values for `level`. Plot the input `y` and each of the faded outputs.

C. Repeater
Now that you've seen the insides of a function file, it's time to write one by yourself. A repeater is a function that plays a particular sound a specified number of times. It seems logical that the function should take a sound file `sound` as an argument, in order to know what to play. You wouldn't want to stifle anyone's creativity, so repeat should also take a parameter `N` in order to let the user specify how many times the sound should be played.
Therefore, the first line of your function might look like this:

```
function[out] = repeat(sound,N)
```

A simple way to repeat things an arbitrary number of times is with a `for` loop. Inside the `for` loop you will need to concatenate the sound signals. If you have two vectors `x` and `y`, you can concatenate them as follows:

```
>> x = [1 4 2 2];
>> y = [3 6 8 0];
>> x = [x y]
x =
1 4 2 2 3 6 8 0
```

Exercise 6: Implement `repeat.m` as described above. It should take two parameters: `sound` and `N`. Demonstrate what happens when you set `N` to be each of 3,0,-1.

D. Delay
The next function to implement is one that time-delays a signal by some amount of time delay. A time delay is the same as prepending silence to the original signal. Because you're working with digital data, a 0 sounds like silence. You can therefore implement a time delay by zero-padding the start of the vector containing the signal. To zero-pad something means to fill a space with only zeros, so in this case you're concatenating some number of zeros with the signal. The number of zeros to add will depend on the time delay and the sample rate of the signal. Adding 4000 zeros to a signal sampled at $4kHz$ will delay it by 1 second; adding 4000 zeros to a signal sampled at $8kHz$ will delay it by only 0.5 seconds. You may find the matlab command zeros to be useful.
Most of the sound files in these labs are sampled at $8000Hz$, but this is not always true. You may want to change this later, so it's good programming practice to have the sample rate F_s be another input to the function.

Exercise 7: Implement `delay.m` to time-delay a sound, as described above. It should take three parameters: `sound`, `delay`, and `Fs`. Plot the original signal and a delayed version of it (use `subplot`, of course) in order to verify your output. What happens when you delay by a negative amount?

E. Mixer
The last helper function you will write is a mixer (`mixer.m`). This mixer should layer two sounds on top of each other so that they play simultaneously. A simple way to do this is to add the signals together. Your code should be able to handle two sounds that are not the same length.
However, there are some considerations: Audio signals have a range of [-1,1], thus the output of your mixer must also fall within this range of [-1,1]. Note that the summed signal will not satisfy this requirement by default! Anything out of range will be distorted, which is bad, so you must re-scale the summed sound before you output it. You can find one way to solve this problem by looking at the source code to the matlab function `soundsc` (type "`type functionName`" to see the source code for a function). If you find the maximum value that your summed input takes, and you can scale that to fall within [-1,1], then your whole signal will also fall within that range.

Exercise 8: Implement the function `mix.m`, which mixes two sounds as described above. Pick two different sounds, plot them, mix them, and plot the result to verify your work.

2. Make Your Own Music

(1). Prerequisite Knowledge: About Music

In this section, we explore how to use simple tones to compose a segment of music. By using tones of various frequencies, you can either construct a segment of a cetain song existed like Scarborough Fair, or compose your own song. Each musical note can be simply represented by a sinusoid whose frequency depends on the note pitch. ***In this lab, you will still use a sampling rate of 8 kHz.***

There are seven natural notes: A, B, C, D, E, F and G. After G, we begin again with A. Music is written on a "staff" consisting of five lines with four spaces between the lines. The notes on the staff are written in alphabetical order, the first line is E as shown in Figure 1. Notes can extend above and below the staff. When they do, ledger lines are added.

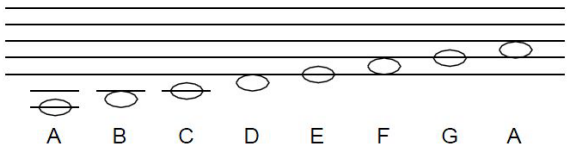


Figure 1. Natural notes.

Musical notes are arranged in groups of twelve notes called octaves. The notes that we'll be using for Scarborough Fair are in the octave containing frequencies from $220Hz$ to $440Hz$. The twelve notes in each octave are logarithmically spaced in frequency, with each note being of a frequency $2^{1/12}$ times the frequency of the note of lower frequency. Thus, a 1-octave pitch shift corresponds to a doubling of the frequencies of the notes in the original octave.

Table 1 shows the ordering of notes in the octave to be used to synthesize the opening of Scarborough Fair, as well as the fundamental frequencies for these notes.

Table 1. Notes in the 220 – 440 Hz octave

Note	Frequency
A	220
$A^{\#}$, B^{b}	$220\times2^{1/12}$
B	$220\times2^{2/12}$
C	$220\times2^{3/12}$
$C^{\#}$, D^{b}	$220\times2^{4/12}$
D	$220\times2^{5/12}$
$D^{\#}$, E^{b}	$220\times2^{6/12}$
E	$220\times2^{7/12}$
F	$220\times2^{8/12}$
$F^{\#}$, G^{b}	$220\times2^{9/12}$
G	$220\times2^{10/12}$
$G^{\#}$, A^{b}	$220\times2^{11/12}$
A	440

A musical score is essentially a plot of frequencies (notes) on the vertical scale versus time (measures) on the horizontal scale. The musical sequence of notes for the piece of Scarborough Fair you will synthesize is given in Figure 2. The following discussion identifies how musical scores can be mapped to tones of specific pitch and duration.

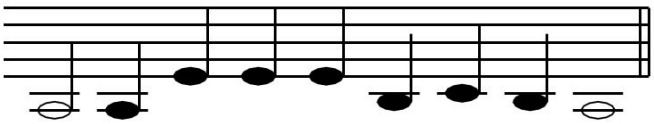


Figure 2. Musical Score for Scarborough Fair.

In the simplest case, each note may be represented by a burst of samples of a sinusoid followed by a shorter period of silence (samples of zeros, which are a pause). The pauses allow us to distinguish between separate notes of the same pitch. The duration of each note burst is determined by whether the note is a whole note, half note, fourth note, or an eight note (see Figure 3). Obviously, a fourth note has twice the duration of an eighth note, and so on.

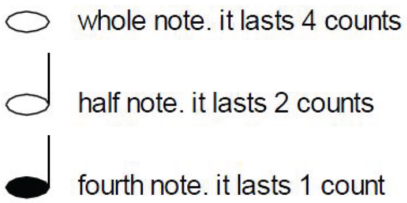


Figure 3. Types of notes.

In this Lab, use a duration of 4,000 samples for 1 count. Therefore, your whole notes should be four times the duration of your fourth notes. The short pause you use to follow each note should be of the same duration regardless of the length of the note. Longer periods of silence that are part of the musical score can be indicated by one of more rest symbols.
Note that A-G only yields seven notes; the additional notes are due to changes in pitch called sharps (denoted by the symbol #) or flats (denoted by the symbol b) that follows a given note. A sharp increases the pitch by $2^{1/12}$ and a flat decreases the pitch by $2^{1/12}$.
In the musical score in Figure 2, the first half note and fourth note are both A. The next three fourth notes are all E and so on. You can get the fundamental frequencies for these notes from Table 1.

(2). Improving perceived quality
There are many ways of improving the perceived quality of a synthesized sound. Here you will learn about one methods: ***Volume variations***.
Typically, when a note is played, the volume rises quickly from zero and then decays over time, depending on how hard the key is struck and how long it is depressed. The variation of the volume over time is divided into four segments: ***Attack, Decay, Sustain, and Release (ADSR)***. For a given note, volume changes can be achieved by multiplying a sinusoid by another function called a ***windowing function***. An example of such function is shown in Figure 4.

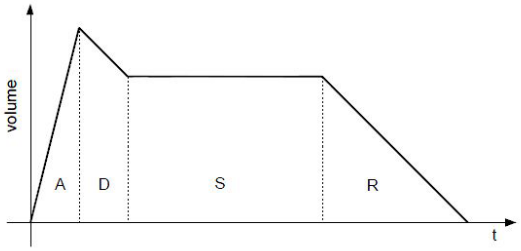


Figure 4. An ADSR Envelope.

So it's time to use the tools you've created! Write a script (not a function) to assemble each note of a musical groove in one long sound vector. You can also use your concatenation and repeater to play sounds in sequence in a track, fade, shift or use your mixer to layer the tracks together. You can also make additional functions if you want. For example, you could modify `repeat.m` to allow you to insert silence between repeats, and so on.
After you've created your sound file, you should save it with the `wavwrite` command. Remember to specify your sample rate F_s, which is $8000Hz$ for the sounds!

Exercise 9: Synthesize all notes of your music groove and save the entire music synthesis in an `YourName.m`-file. Write it to the file groove `YourName.wav` with the `wavwrite` command and play your music groove for the TA.