Lab 4 Information

- The material in this section is informational. Please read through the section as it helps you work on the lab exercises in the next section. There may be code examples in this informational section. You are welcome to copy-and-paste them to MATLAB to run the code, but no submission is needed on any test run.

Implementing Discrete-time Systems in MATLAB

For a linear time-invariant (LTI) system, we know that the output signal can be obtained by convolving the input signal with the impulse response of the system. Suppose the impulse response of the LTI system and the input signal are stored in the MATLAB vectors `h` and `x`, respectively. We can then generate the output in the vector `y` (i.e., implement the system) by using the MATLAB function `conv` to perform convolution as below:

```
>> yc = conv(h,x);
```

where `yc` is the output signal. You may want to note the length of `yc` in relation to those of `h` and `x`. The order of `h` and `x` in `conv` may be reversed since the convolution operation is commutative. While the implementation is exact for any FIR filter (why?), it is an approximation for an LTI system whose impulse response is not finite in length. Recall that any MATLAB vector can hold only a finite number of samples of a signal. Hence, we can store only a truncated version of the impulse response (if its length is infinite) in a MATLAB vector; so the implementation using `conv` gives an approximation to the actual output signal of the LTI system.

We can also directly implement a discrete-time system from its difference equation specification. For an FIR filter, we may implement the difference equation using the MATLAB `filter` function as follows:

```
>> yf = filter(h,1,x);
```

where `h` is the vector that contains the impulse response of the FIR filter as above. Use the `help` command to understand why `filter(h,1,x)` implements the difference equation of the FIR filter. Note that the output vectors `yc` and `yf` obtained from the two different implementations above are NOT exactly the same. In particular, their lengths are different. I would suggest that you test out the two implementations by picking an example input vector `x` and an example FIR filter (i.e., an example `h`). From your examples and the `help` command, study and understand the differences between the two different implementations. Note it is also possible to employ the two functions `conv` and `filter` in slightly different ways than the above to make them produce the same output.

For a non-LTI system, neither of the functions `conv` and `filter` can be used to implement the system. You have to implement the difference equation yourself. For example, consider the nonlinear system described by the following difference equation:

$$y[n] = x^2[n] + x^2[n-1]$$

where the signal $x[n]$ is given in the MATLAB row vector `x`. Then you will need something like the following code to implement the system:

```
>> x_delayed = [0 x]; % x[n-1]
>> x_longer = [x 0]; % Add zero to end of x to make it as long
                     % as x_delay
>> y = x_longer.^2 + x_delayed.^2;
```

Note that when no information about the value of the sample before the start of the vector `x` is given, we typically make the assumption that the value is 0. This is called the *initial rest assumption*. Hence, we insert the value 0 at the beginning of the vector `x_delayed`, which represents the delayed input signal $x[n-1]$.

For two-dimensional signals (images) and impulse responses, one may instead use the MATLAB functions `conv2` and `filter2`. Use the `help` command to learn more about these two functions, and how to employ them to implement two-dimensional filtering .

Lab 4 Exercises

- Unless stated otherwise, you must submit your solutions to all the lab exercises in this section.
- Your laboratory solutions should be submitted on Canvas as a single PDF. The simplest way is to put your codes and answers for all the lab exercises in a single MATLAB Publisher script and use `%%` to separate the codes and answers for different exercises into different sections as described in the information section of Lab 1.

Exercise 4.1: *(Reverb and Tremolo)*

This exercise shows you how to implement a reverb effect and a tremolo effect on a song by creating two discrete-time systems and applying them to an audio signal.

(a) The effect of reverberation can be modeled by an LTI system that is described by the following difference equation:

$$y[n] = x[n] + Ax[n - s]$$

where A is the reverb amplitude and s is the reverb delay. Since the system is discrete time, s must be a positive integer, expressing the delay in the unit of samples. Give an expression, in terms of s and the sampling frequency f_s, for the reverb delay in seconds.

Give an expression for the impulse response $h[n]$ of the reverb system. Generate $h[n]$ in the MATLAB vector `h` for the case of $A = 0.8$ and $s = 8000$ (hint: you may use the function `zeros`). Create a MATLAB function `y = reverb(x, s, A)` to apply the reverb system on the input signal `x`. Give three different implementations of `reverb` using `conv`, `filter`, and your own direct implementation of the difference equation above. You may name the three different versions of implementation for example as `reverb_conv`, `reverb_filter`, and `reverb_own`. Try to comment on the potential differences in computational efficiency of the three different ways of implementing the reverb system.

(b) Read in the WAV file `TreatYouBetter.wav` provided together with this document. Put the song signal in the vector `tyb_orig`. Refer back to the document for Lab 1 if you forget how to do that. Apply your `reverb` function to the song to get output `tyb_reverb` by setting the delay to `s = 8000` samples and amplitude to `A = 0.8`. Use `soundsc` to hear the reverb effect. Save `tyb_reverb` to the WAV file `tyb_reverb.wav` and submit it together with your report. You may choose any one of your three implementations in (a) to answer this part. However, you may want to test them all to see if they give you the same sound effect, and perhaps as a way to compare them in terms of computational efficiency.

(c) The tremolo effect is, on the other hand, modeled by a linear but timing-varying system:

$$y[n] = x[n] + A\cos(2\pi \hat{f}_m n)x[n]$$

where A is the tremolo amplitude and $\hat{f}_m$ is the tremolo modulation frequency (in normalized cyclic frequency). Create a function `y = tremolo(x, fm, A)` that implements the tremolo system.

(d) Apply `tremolo` to `tyb_orig` to get the output vector `tyb_tremolo`. Set the amplitude to `A = 0.3` and modulation frequency to `fm = 10 / fs`, where `fs` is the sampling frequency. Use `soundsc` to play `tyb_tremolo` to hear the tremolo effect. How does `tremolo` affect the audio and why? Save `tyb_tremolo` to the WAV file `tyb_tremolo.wav` and submit it together with your report.

(e) Again apply `tremolo` to `tyb_orig` to generate the output vector `tyb_faded`, but now with the amplitude `A = 1` and modulation frequency `fm = 1 / length(yb_orig)`. Use `soundsc` to play the output. Describe what you hear and explain why.

(f) We have discussed in class that a system is time-varying if the system $\mathcal{T}\{\cdot\}$ satisfies

$$\mathcal{T}\{x[n-N]\} \neq y[n-N] \qquad \text{such that} \qquad \mathcal{T}\{x[n]\} = y[n]$$

for some delay N.

Delay `yb_faded` in (e) by $N = 123480$ samples (half the signal length) to get the delayed signal vector `tyb_faded_N`. Use `soundsc` to play `tyb_faded_N` . Then delay the input `tyb_orig` by $N = 123480$ samples to get `tyb_orig_N`. Input `tyb_orig_N` into `tremolo` with the parameters in (e) to get `tyb_N_faded`. Use `soundsc` to play `tyb_N_faded`. Save `tyb_faded_N` and `tyb_N_faded` to the WAV files `tyb_faded_N.wav` and `tyb_N_faded.wav`. Do `tyb_faded_N` and `tyb_N_faded` sound the same? Explain why or why not.

Exercise 4.2: *(Image Filtering)*

Spatial filtering is often used to modify images. We can regard the image as an input signal and the spatial filter is our LTI system. The output of this system is obtained by doing two-dimensional convolution between the input image $g[x, y]$ and the filter impulse response $w[u, v]$. For example, the output image $f[x, y]$ with 3×3 filter impulse response $w[u, v]$ can be obtained by the 2-dimensional convolution

$$f[x, y] = \sum_{u=-1}^{1} \sum_{v=-1}^{1} w[u, v] g[x-u, y-v].$$

This lab exercise shows you how to use some simple spatial filters to modify images.

(a) Consider the filter impulse response (also known as a kernel) below.

$$\begin{array}{|c|c|c|}\hline w_a[-1,-1] & w_a[-1,0] & w_a[-1,1] \\ \hline w_a[0,-1] & w_a[0,0] & w_a[0,1] \\ \hline w_a[1,-1] & w_a[1,0] & w_a[1,1] \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline 1/9 & 1/9 & 1/9 \\ \hline 1/9 & 1/9 & 1/9 \\ \hline 1/9 & 1/9 & 1/9 \\ \hline \end{array}$$

Put this impulse response in the matrix `wa`. Apply this filter to the image `lighthouse` in Lab 3 by using the MATLAB function `filter2`. Make sure that the size of image remains unchanged before and after filtering. There is no need to write your own function for these implementations of the filter. It suffices to directly use the built-in function `filter2` in your command window or your publisher script.

Use `imagesc` and `subplot` to plot side by side the original image and the image obtained after applying the filter. Compare your plots and answer the questions below:
Does this filter blur, sharpen, or extract edges in the image? How / why do you know that from the impulse response / kernel?

(b) Consider the filter impulse response / kernel below.

$$\begin{array}{|c|c|c|}\hline w_b[-1,-1] & w_b[-1,0] & w_b[-1,1] \\ \hline w_b[0,-1] & w_b[0,0] & w_b[0,1] \\ \hline w_b[1,-1] & w_b[1,0] & w_b[1,1] \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline 1/9 & 1/9 & 1/9 \\ \hline 1/9 & -8/9 & 1/9 \\ \hline 1/9 & 1/9 & 1/9 \\ \hline \end{array}$$

Put this impulse response in the matrix `wb`. Apply this filter to the image `lighthouse` by using the function `filter2` such that the size of the image remains unchanged before and after the filtering process. There is no need to write your own function for these implementations of the filter. It suffices to directly use the built-in function `filter2` in your command window or your publisher script.

Use `imagesc` and `subplot` to plot the image before and after applying the filter. Compare your plots and answer the questions below:
Does this filter blur, sharpen, or extract edges in the image? How / why do you know that from the impulse response / kernel?

(c) Create a function `im_out = image_unsharp_masking(im_in)` that performs the following operations on the input grayscale image:

  (i) Apply the filter `wb` to `im_in`.

  (ii) Subtract the filtered image in (i) from the original input image `im_in` to give output image `im_out`.

As in (b), use `filter2` in your function to implement any filtering such that the size of the image remains unchanged before and after the filtering process. This function implements the process known as *unsharp masking*. This process of unsharp masking can be modeled by a spatial filter. Determine the impulse response of the unsharp masking filter.

Apply `image_unsharp_masking` to the output image in (a). Use `imagesc` and `subplot` to plot the image before (the output image in (a)) and after applying the unsharp masking filter. Does this filter blur, sharpen, or extract edges in the image?

(d) Repeat (a)–(c) using the MATLAB function `conv2` instead:
For example, use the line

```
lighthouse_filtered_by_wa = conv2(wa, lighthouse);
```

to apply the filter `wa` to the image `lighthouse` as an alternative to using `filter2` in (a). Similarly, use `conv2` to redo (b). For both cases, report the size of the image before and after filtering. Modify your function `image_unsharp_masking` in (c) to use `conv2` instead. Pay special attention to the sizes of the matrices in step (c)(ii). Hint: Consider and make use of the (3×3) two-dimensional unit impulse signal

$$\begin{array}{|c|c|c|}\hline \delta[-1,-1] & \delta[-1,0] & \delta[-1,1] \\ \hline \delta[0,-1] & \delta[0,0] & \delta[0,1] \\ \hline \delta[1,-1] & \delta[1,0] & \delta[1,1] \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}$$

as the impulse response of a spatial filter.

Exercise 4.3 (Extra credits: +10 points):

This exercise shows you how to process an audio signal to emulate playing the audio in an environment with specific acoustic properties. The acoustic properties of a concert hall or a room can be approximately modeled by an LTI system / filter. As shown in the Week 4's supplemental videos, one may capture the impulse response of a concert hall experimentally. To emulate playing an audio signal in the concert hall, one may simply apply the filter to the audio signal using the captured impulse response.

Load the impulse responses of a concert hall from the MAT file `hall.mat` provided. After loading the MAT file, you should see two variables, `hh` and `fs`, added to your workspace. The variable `fs` stores the sampling frequency (in Hz) at which the impulse responses are captured. The matrix `hh` has two columns; each column gives the impulse response of the concert hall captured for the corresponding stereo channel. Note that a stereo sound has two channels, one each for the left and right speaker. Hence, to emulate the stereo effect of playing an audio signal in the concert hall, one needs to convolve the corresponding impulse responses with the audio signal to generate an output audio signal for the left channel and another for the right channel. To play the generated stereo audio using `soundsc` or save it using `audiowrite`, one may put the two output signal vectors (one each for the left and right channel) as two columns of a matrix. The format is exactly the same as that in the impulse response matrix `hh`. Then pass the audio signal matrix as input to `soundsc` or `audiowrite`.

Write a MATLAB function that takes in an impulse response matrix (such as `hh`) and an audio signal vector as input arguments to emulate playing the input audio in an environment specified by the input impulse response matrix. The output of the function should be an audio signal matrix with two columns, each of which contains the emulated audio signal for the left / right channel. Apply your function with `hh` and `tyb_orig` as input arguments. Use `soundsc` to listen to the emulated stereo sound. Save the output of your function to the WAV file `tyb_stereo_hall.wav` and submit the WAV file.