Lab 1: Introduction to DSP

Professor Deepa Kundur

Objectives of this Lab

This lab:

- introduces you to implementation and performance aspects of digital linear filters through simple case studies;
- consolidates your past knowledge on theoretical DSP concepts by revisiting topics of quantization and filter phase.

Prelab

Prior to beginning this lab, you must carefully read over this lab in order to plan how to implement the tasks assigned.

Deliverables

- During the lab, you must show your TA the Simulink results discussed in the lab instructions; and
- After the lab, each group must submit a separate report answering the questions in this lab.

Grading and Due Date

The prelab, lab results and report will be graded on correctness, comprehensiveness and the insight you are able to provide when answering the questions. For full points, reports should be written in complete sentences with correct grammar.

Please note STRICT DEADLINE for report on the course web site.

Lab 1: Introduction to DSP

Introduction and Background

Welcome to your first *real* digital signal processing (DSP) lab. Before we jump into the lab itself, let us outline the aim of all the labs in a nutshell. The goal of these labs is to prepare you for the "real-world", where you may be programming DSPs for signal processing, energy, biomedical, robotics, or communications companies amongst others. One main intention is to introduce you to what a DSP engineer would be doing from the software, hardware and application-based perspectives. This means the labs will cover theory, design, testing, comparison and evaluation, implementation, as well as user interfacing. Every step is significant, and it is important to be familiar with all steps even if you eventually specialize in only one aspect of this development process.

In this lab, we will introduce you to digital filters. In practice, it may seem that you only need to design the filter coefficients (see Lab 2), and never have to worry about what is in that "black box". However it is still essential that you know how digital filters work. Why? So that you can compare and evaluate solutions as well create innovative streamlined new solutions that one can't just "google" the answer to. Consider the basics of a system. A system is simply a black box that takes an input, and returns an output as a function of the input. If this system is user-defined and has the goal of enhancing certain aspects of the input signal while attenuating others, it is often termed a "filter". In this course, we will consider single-input single-output filters.

We focus on single-input single-output *digital* filters. Digital filters are filters that process and output discrete-time and discrete-amplitude signals (often collectively called digital signals). A digital signal can be represented as a vector with countably many elements, whose amplitudes come from a finite alphabet; for example, consider a signal measuring the number of stocks traded per day by the NASDAQ. Physically, everything we experience is continuous. For example, when light streams in, we perceive it as having a spectrum of possible colors. In order for a DSP (or computer) to process such an input from the physical world, it must change this spectrum of continuous inputs to something it is compatible with; i.e., numbers in digital (specifically, binary) format. This involves two steps. The first step involves discretizing the independent signal variable through sampling in time (or space if your independent variable represents physical length). Consider the sampling of an audio signal for input into a DSP. The sampler will usually take equally spaced samples of the audio stream every $T > 0$. The second step deals with discretizing the dependent variable of the variable, i.e., its amplitude. This is often done by rounding (more generally called quantizing) the signal values to the nearest integer multiple of a quantization factor, and "thresholding" if needed to make sure the signal is within a given range of values, e.g., between $-A$ volts and A volts. For some types of signals such as audio, digital image and digital video, because of the limited dynamic range of human perception, if the quantization process is "fine-enough" you will not notice the information loss when converting from an analog signal to a digital one.

Analytically, in this lab we represent a given one-dimensional digital signal input as x_k where the dependent variable is represented by x and the independent variable by k. For any fixed k_0, the value x_{k0} is

known as a sample of the input. We will design and implement linear filters. Linear filters are by far the easiest filters to implement, analyze and often design, which is the reason they are so often used in practice. The operations involved in implementing linear filters are scalar multiplication of the input (and/or output) samples with associated *weights*, and then addition of the resulting weighted inputs/output components. For example, averaging a group of 10 integers that are in the range of 0 to 5 involves a linear filter; the input to this filter is a collective group of 10 integers and the output is their average value. Each of the 10 numbers is multiplied by the scalar constant 1/10 (which represents the filter weight for each input sample), and then is summed together. Although this is not a very general or interesting filter, the concept easily extends to what are known as discrete-time Finite Impulse Response (FIR) filters. Digital FIR filters necessarily act on a digital input to produce a digital output such that each output value is a weighted sum of a select subset of input values that are appropriately synchronized to the weight-values. These weights completely characterize the FIR filter and are collectively termed the filter coefficients.

Consider a sequence of input samples. We would like to average, in real-time, every *pair* of input samples (current input value with previous input value) to produce the output. If the input is denoted x_k, for integer k, the FIR filter weights are given by $h_0 = h_1 = 0.5$, and $h_k = 0$ for all $k \neq 0, 1$, then the output samples can be represented generally as:

$$y_n = \sum_{k=-\infty}^{\infty} h_{n-k} \cdot x_k \qquad (1)$$

where y_k represents the output signal. Equation (1) represents, the general input-output convolution relationship of an LTI system where h_k for all k is more generally interpreted as the impulse response of the associated linear time invariant (LTI) system.

You can verify that indeed (1) does the job of a real-time two-sample averager. For example, $y_{10} = 0.5\ x_9 + 0.5\ x_{10}$ can easily be verified. The weights h_k can also be interpreted as the FIR filter coefficients, where this filter is also known as a *moving average* (MA) filter. Thus FIR filters are simply linear time-invariant (LTI) filters with a *finite* number of non-zero components, and effectively a finite impulse response.

In a real-time DSP course, we are concerned with causal filters because of their ability to be realized. For this course, we will consider time to begin at $t = 0$ making both signal and filter coefficients 0 for negative time values. If you recall, we require causality because any output sample at time n must depend only on input samples at times n, n-1, n-2, ..., 0; future input values should not be needed to compute the present output sample value, which would defy the laws of physics associated with cause and effect. One can easily show that a FIR filter of (1) with $h_k = 0$ for $k < 0$ always satisfies this causality requirement. Thus, employing causality requirements for the filter weights, one can write the input-output relationship for FIR filters as:

$$y_n = \sum_{k=0}^{n} h_{n-k} \cdot x_k \qquad (1)$$

Furthermore, given that we know M which represents the upper support value of h_k such that $h_k = 0$ for $k < 0$ and $k > M+1$, using the finite-length requirement of the FIR filter, (1) can be rewritten more specifically as:

$$y_n = \sum_{k=\max(0,n-M)}^{n} h_{n-k} \cdot x_k \qquad (1b)$$

Similarly, with a change of the dummy index, one can write the input-output representation of the MA filter also as:

$$y_n = \sum_{k=0}^{M} h_k \cdot x_{n-k} = \sum_{k=0}^{\min(M,n)} h_k \cdot x_{n-k} \qquad (1c)$$

A more general kind of filter can be described as depending on not only on the inputs, but also on past outputs. When a filter depends on its own past outputs, it is called *autoregressive*, with "auto" referring to the "self". When it depends on both past outputs and past and present inputs, the filter is called autoregressive moving average (ARMA) and has the following form:

$$y_n = \sum_{k=0}^{\infty} b_{n-k} \cdot x_k - \sum_{k=0}^{n-1} a_{n-k} \cdot y_k \qquad (2)$$

Notice that we now have two sets of coefficient weights denoted $\{ a_k \}$ and $\{ b_k \}$. Also note that (2) is in fact a generalization of (1), degenerating to (1) if we set $a_k = 0$ for all k. Alternatively we can write (2) differently via a change of index variable as in (3). Also, for practicality, restricting the coefficient weights to have finite support (i.e., $a_k = 0$ for $k < 0$ and $k > N$ and $b_k = 0$ for $k < 0$ and $k > M$) gives us the following popular difference equation form of a discrete-time filter:

$$\sum_{k=0}^{M} b_k \cdot x_{n-k} = \sum_{k=0}^{N} a_k y_{n-k}, \quad a_0 = 1 \qquad (3)$$

In this case, the number of filter coefficients is explicit, i.e., $M+1$ and $N+1$ for $\{b_k\}$ and $\{a_k\}$, respectively. This form is also z-transform friendly, and at the same time it also gives a clear picture of how to implement the filter. For example, to compute the output y_n based on the input x_n, one can use delays (i.e., memory elements) to store the past y_k and x_k, and then form the two sums and subtract one from the other to give a Direct Form I realization as you should have seen before. With some ingenuity, we can cut the number of memory elements we use by $\min(M,N)$, which is almost a savings of one half of the memory elements for $M \approx N$.

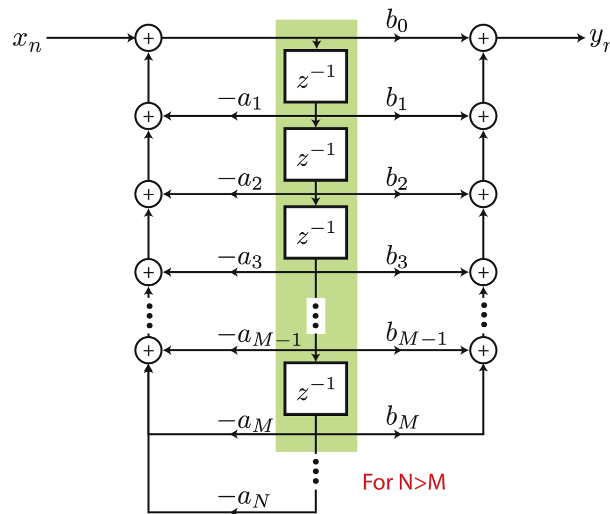


Figure 1: Direct II Transpose Realization

Figure 1 shows a simple implementation of (3) using the minimum number of memory elements; a z^{-1} block represents a until delay memory element implemented using a shift register and each coefficient above every arrow represents scalar multiplication of the associated signal with the corresponding weight value. Filters of this form represent a class of LTI filters (with what we call rational system functions), however they differ from FIR filters in that the equivalent h_k can have infinitely many non-zero samples, thus these filters are also called Infinite Impulse Response (IIR) filters. There is an even better implementation of filters that reduces propagation of quantization noise and offers faster performance, and it is based on decomposing the transfer function (in the z-transform domain) using partial fractions, but we do not discuss it here.

Design and Implementation

Building Basic Filters

In this lab you are going to build two simple FIR filters using basic Simulink blocks.

1. Open a new Simulink model.
2. Build a FIR moving average filter discussed in the Introduction and Background section with gains $h_0 = 0.5$ and $h_1 = 0.5$.
3. In the same Simulink model, build a second FIR filter using Direct II Transpose form. This time, use the gains $h_0 = -0.5$ and $h_1 = 0.5$. You will use the same input for this filter and the filter you created in Step 2 above; essentially, the filters will operate in parallel so you can test them at the same time.
4. Let us quickly take a little detour and use MATLAB to view the frequency responses of the filters you just made in Simulink. First, for the moving average filter, go to the MATLAB command window and type `f1 = dfilt.dffir([0.5 0.5])`. Here, `dfilt` stands for digital filter, and `dffir` stands for direct form FIR. This function takes the impulse response that you provide (in the form of a vector) and creates an FIR filter object (`f1` is the just the name of this object that you assign). Now, type `freqz(f1)` and press enter. MATLAB's Filter Visualization Tool should appear,

showing you both the magnitude and phase responses of the filter. Note in particular the magnitude response.

5. Repeat Step 4 for the FIR filter from Step 3. Again, note carefully the magnitude response.

Creating a Noisy Input Signal

The input to your filters will be a sinusoid that has been corrupted by high frequency noise. Both the sine wave and the noise will be discrete; thus, your whole Simulink model will be discrete (this is purposely done to simplify things). Here's how to create such an input signal:

1. Add a Sine Wave block to your model. Set the frequency to 100 Hz and the Sample time to 1/8000.
2. Add a Uniform Random Number block to your model. Set Sample time to 1/8000 and Maximum and Minimum to 5 and -5, respectively. As its name suggests, this block outputs random numbers, which for our purposes is a good model for noise.
3. We want to restrict our noise to only have high frequency components. To do this, we are going to send it through a highpass filter. Add a Highpass Filter block and open the Block Parameters. Under Filter specifications, set Impulse response to IIR and Order mode to Minimum. Under Frequency specifications, set Frequency units to Hz, Input Fs to 8000, Fstop to 3600, and Fpass to 3800. Under Magnitude specifications, set Magnitude units to dB, Astop to 60, and Apass to 1. Finally, under Algorithm, set Design method to Elliptic and Structure to Direct-form II SOS. Right now, all that you have to understand about this filter is that it only passes frequencies above 3800 Hz (Fpass); anything below 3600 Hz (Fstop) is attenuated by at least 60 dB (the region between these two frequencies is a transition region).
4. Send the random number signal from Step 2 through the highpass filter and add it to the sine wave you made in Step 1. Take the result of this addition and use it as the input to both of your FIR filters.
5. To view the outputs of your filters, use Scope blocks. Also, use two other scopes to view the original sine wave and the noisy sine wave.
6. Verify that all blocks have the same sample time. You can do that by double clicking on the desired block to open the block parameters and set the sample time to 1/8000 in that case (or simply -1 which gets you the inherited sample time).

Testing the Filters

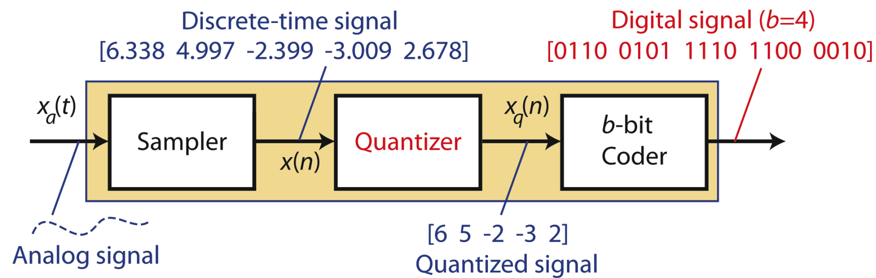
For this lab, you're only going to test your filters in non-real-time for a finite duration.

1. Go to Simulation → Configuration Parameters. In the Solver pane, Under Simulation time, set the start time to 0; choose the stop time so that several periods of the 100 Hz sine wave occur. Under Solver options, set Type to fixed-step and Solver to discrete.
2. Run your Simulink model by selecting Simulation → Start.
3. View the results of your simulation in your scopes and show your simulated model to your TA.
4. What do you notice about the outputs of the FIR filters? Explain why the outputs are as they are using words (not mathematically).
5. Try changing the Fpass and Fstop parameters of the highpass filter to allow some lower frequency noise into your signal. Run your simulation again. What are the outputs like now? Explain your results.

Questions

The answers to these questions should be provided in report form for grading after the lab; exact deadline will be posted on the course web page.

1. *Output Quantization Noise*. We have discussed quantization briefly in the introduction of this lab as well as in the lectures. The general quantization process can be summarized as follows.



Suppose the input signal x_k is a b-bit digital signal (i.e., the input is a discrete-time signal represented in fixed point format with b bits; whether or not it is signed/unsigned depends on the amplitude range of x_k). One source of signal processing error is due to quantization error from the rounding process. Because quantization error is a type of a noise that is applied to the input signal, passing the input samples through a filter may have the effect of amplifying this error further. One way to analyze these effects is to compute the *output quantization noise* of the corresponding filter and see how this could have negative effects on the overall processing.

In this problem, we model the quantized input signal x_k as the sum of the original non-quantized signal denoted x'_k and the associated quantization error/noise denoted e_k. If a *uniform* quantizer is used then e_k is often modeled as a uniformly distributed random variable with zero-mean and variance given by $\sigma^2 = 2^{-2b}/12$ that is independent of x_k. Suppose we have a filter given by $y_k = a\, y_{k-1} + x_k$.

(a) For what a is the filter stable?

(b) What is the mean and variance of the quantization noise at the output of the filter? For this part, model $x_k = x'_k + e_k$ as described above. Since, the filter is linear, the output of the filter to the input $x_k = x'_k + e_k$ can be represented as $y_k = y'_k + f_k$ where y'_k is the component of the output due only to x'_k and f_k is the component of the output due only to e_k. The signal f_k is the output quantization noise of the filter.

(c) For the range of a that provides a stable filter, how large can the variance of f_k get, i.e., what is the bound on the variance given a stable filter? For what value of a does upper bound on variance occur?

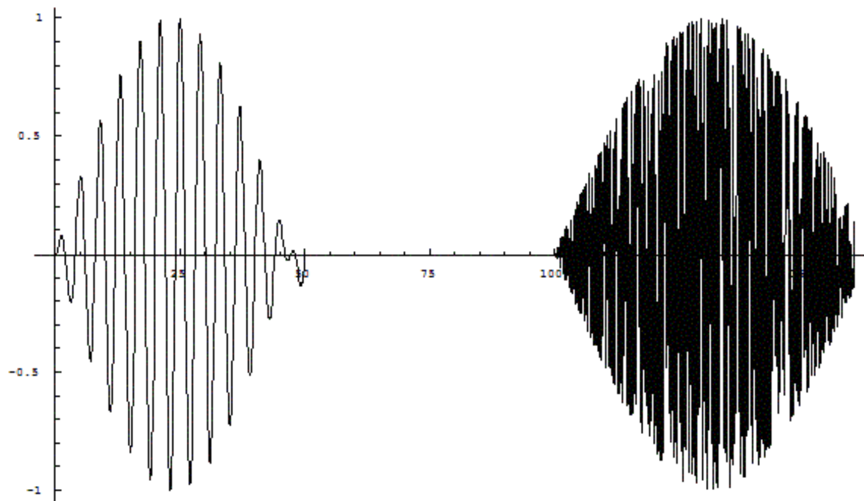
(d) What is the minimum variance of f_k that is possible and for what value of a does this occur?

2. *Linear Phase*. This question asks you to be a bit creative in your thinking. First-time filter designers often concentrate only on obtaining an attractive magnitude response for a given filter while neglecting its phase. Recall, the ideal filter in which the pass band region had a gain of 1 (or 0 dB) while the stop band region had a gain of 0 (or negative infinity dB ☺). But, what about the phase? The answer is that one generally wants *linear phase*. The concept of linear phase can be described using what is called *group delay*, which is defined as the negative derivative (with respect to radial frequency) of the filter phase. Specifically, it is expressed as:

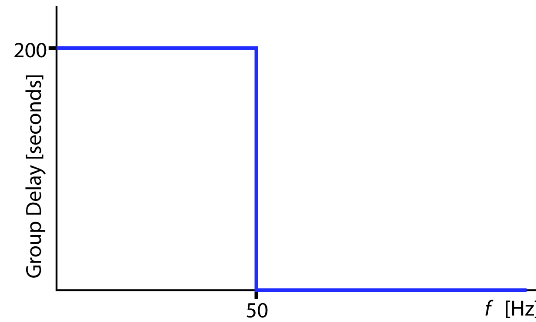
$$\tau(\omega) = -\frac{d}{d\omega}[\angle H(\omega)] \qquad (4)$$

Note that the frequency ω has units if radians/sec, and thus group delay has units of seconds.

If the group delay of a filter is constant, then it exhibits linear phase. What does this mean? It means that every sample in the original input signal is delayed by the same amount – the constant given by the group delay. We will investigate what happens when the group delay is not constant. Suppose we are given the following input signal:



This is a continuous-time signal that is comprised of two pulses as evident, representing for example audio data. Suppose the first narrowband pulse has predominant frequencies around 0.5 Hz, while the second narrowband pulse has predominant frequencies around 75 Hz. Now suppose that the filter in question has a flat magnitude response of gain 1 across all frequencies. However, the group delay is given by:



where the value of the delay is 200 for frequencies between 0 and 50 Hz and is 0 for all frequencies above 50 Hz.

(a) Plot the phase of the filter assuming the filter is real. Please phase-wrap your plot so that the phase (vertical-axis) is confined in the range $(-\pi, \pi]$. For full points your plot must be unambiguous and accurate at significant points of interest.

(b) Draw the corresponding output of the filter. *Hint*: Use the additive property of this linear filter and see how it behaves with each of the pulses separately and then superpose the individual results in the final output.

(c) If the original signal is a speech signal where the two narrow band pulses starting at 0 seconds and 100 seconds correspond to the spoken words "forward" and "thinking", respectively, please state what the output will sound like interpreted as spoken words.