ECE 2713 – Programming Module #3

Ideal & Practical Interpolation

Due Thursday, April 6 at the beginning of class

This third programming assignment will explore ideal interpolation of sampled (discrete-time) signals back to their original continuous-time form. Specifically, you will be given discrete-time samples, which you will then interpolate using two methods: 1) the ideal interpolation formula (using sinc functions), and 2) a sample-and-hold with lowpass filtering approach.

I will provide you with the basic equations and procedures necessary to implement the assignment. My goal in providing you with some application examples via simulation project is threefold: 1) to demonstrate the usefulness of the topics we are covering in class through application to practical problems; 2) to strengthen your understanding of the concepts covered in class; and 3) to give you practice in mapping technical problems to programming structures.

Once you have your program working, you should create several well-labeled plots showing that your program is working. The plots should be readable, and should have properly labeled x- and y-axes. You must also turn in a copy of your Matlab code. You **must** work with **1-2** partners (groups of 2-3) and turn in a joint report. Groups may talk to each other about the assignment, **but all Matlab code and report materials must be the result of your own group's work**. Code copied from another group is unacceptable.

Background: Ideal Interpolation

We have covered sampling and ideal reconstruction in class. To review, if we have a continuous-time signal $x(t)$, we can produce a corresponding discrete-time sequence by capture and storing values of $x(t)$ at uniform time intervals. We will let the sampling time interval be denoted by T_s; therefore, the sampled data sequence will be

$$x[n] = x(t)\big|_{nT_s} = x(nT_s). \tag{1}$$

The data record or sequence, $x[n]$ will be your starting point for this project. Your objective will be to reconstruct $x(t)$ from $x[n]$. From the textbook and class discussion, we know that the ideal interpolation formula is

$$x(t) = \sum_{n=-\infty}^{\infty} x[n]\,\mathrm{sinc}\left((t - nT_s)/T_s\right) \tag{2}$$

where

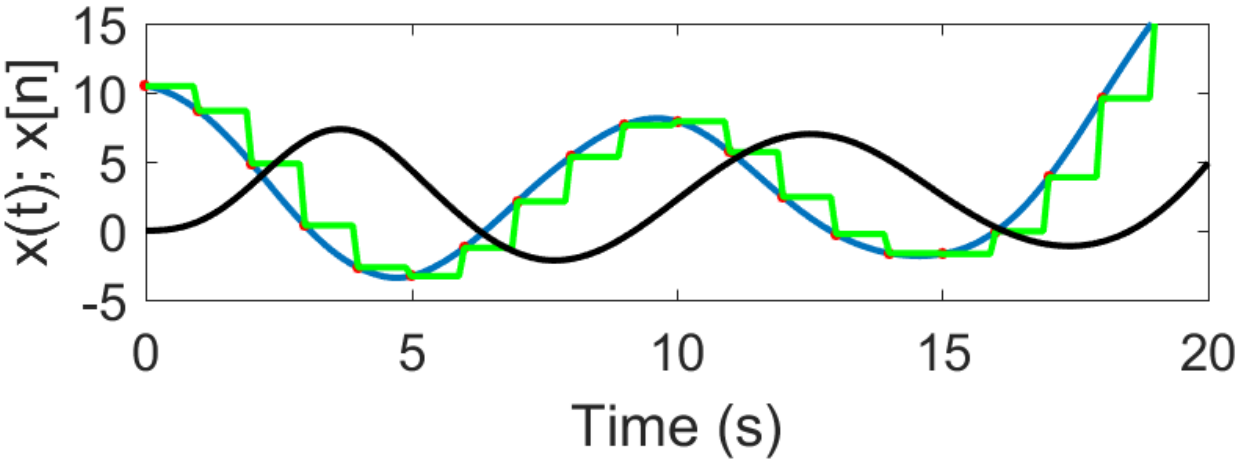

Figure 1. Sample plot showing original signal (blue), discrete-time samples (red), sample-and-hold signal (green), and output of lowpass filtering (black).

$$\text{sinc}\left(\frac{t}{T_s}\right) = \frac{\sin\left(\pi t / T_s\right)}{\pi t / T_s} \tag{3}$$

is known as the "sinc function." Equation (2) shows that the original continuous-time signal can be reconstructed by summing together an infinite number of scaled (by $x[n]$) and shifted (by nT_s) sinc functions, with width determined by the sampling interval.

Your goal for this project will be to implement (2) over a finite duration of the interval. As shown, (2) is valid for an infinite amount of time and requires an infinite number of data samples. This obviously won't work for us on a simulation because we can't store an infinite number of values, so we will be using a finite number of samples for the reconstruction and will perform reconstruction over a finite time window. Within this time window, we will emulate a continuous-time variable by computing a discrete-time sequence at a much higher rate than the original sequence.

Background: Sample and Hold with Lowpass Filtering

Ideal interpolation is not a practical approach; therefore, there are a number of approximate methods for discrete-time to continuous-time interpolation. One of the most straightforward is to perform a "sample-and-hold" followed by lowpass filtering. In this approach, each data sample is "held" for a sample period in order to form a continuous-time, stairstep-like signal (see Figure 1 for an example). Then, in order to remove the sharp transitions, this stairstep signal is smoothed via a lowpass filter where the cutoff frequency is matched to the original sampling rate. Because there is a time delay to pass through the lowpass filter, the reconstructed signal is somewhat delayed compared to the original.

Project Assignment

Figure 1 demonstrates a lot of what you will try to accomplish in this project. You will be given a data set of discrete-time samples (equivalent to the red dots in Figure 1), along with the sampling time interval. We will assume that the first sample in this data set was collected at $t = 0$. Based on this start time, the sampling interval, and the number of samples in the data sequence, you will be able to set up a time array that goes with the discrete-time data. Next, you will set up a new time array that covers the same time interval as the discrete-time data. This new time array is meant to mimic a continuous-time variable, and so the sampling intervals for the new time axis should be at least 10 times smaller than the original sampling interval. Once you have the new time array, you will use the two different methods to interpolate the signal values for this "continuous-time" signal.

Here are the basic steps:

- Load the data file(s) and set up a time array that contains the times at which the data was originally sampled. The sampling rate is contained as one of the variables in the data file(s), and is not the same for both files that you will process. The first data sample in each file is assumed to occur at $t = 0$.
- Set up a second time array that starts at $t = 0$ and has a sampling rate of 441 Ksamples/second.
- For ideal interpolation, implement (2) by creating properly shifted sinc functions, scaling them by the input data samples, and summing to create the reconstructed signal;
- For the sample-and-hold approach, first you must:
    - Repeat each input data sample to simulate the "hold" operation. You must repeat the input discrete-time samples as many times as necessary to match the new, higher-rate time array.
    - Design a lowpass filter (see notes below) and store the coefficients of the filter in an array.
    - Convolve (using the 'conv' command) the "held" signal with the lowpass filter.

Once the signal is interpolated, plot the original discrete-time samples versus time, the "sample-and-hold" signal versus time (for the non-ideal case), and the reconstructed signal versus time.

Notes:

1. For the filter design, you will use Matlab's built-in filter design by the window method. The command for this is "fir1". The two input values you need for the filter are the "filter order" and the cutoff frequency. For the filter order, the filter's cutoff frequency response improves with higher filter order, but the delay through the filter increases. If the input argument that you give for the filter order is N, then the length of the filter coefficient array will be $N + 1$. I suggest even values for N, and increase N until you find a good reconstruction.

For the cutoff frequency, you need to input a number as a fraction of half the sample rate. So, given the original discrete-time sampling rate of F_s, the highest frequency that can be represented (according to Nyquist), is $F_s/2$, which is what you want for the cutoff frequency in Hertz. The

higher-rate data will be at 441 KHz, so this is the sampling rate for the filter coefficients. The documentation for the "fir1" command wants a cutoff frequency normalized to half the sample rate; therefore, the cutoff frequency should be (F_s/2)/(441,000/2).

2. You should be able to play the input data set as a sound file. Use the 'soundsc' command, which is a scaled version of the 'sound' command. This command will play the Matlab array as audio through your computer speaker, but you need to give the sample rate. After interpolation, you should also be able to play the sound file at the higher sampling rate (be careful though – some sound cards have an upper limit on the sampling rate that they can handle).

3. For the "sample-and-hold" approach, try different filter orders and window shapes for the filter design, then see if they make any difference on the reconstruction.

4. The input data files have different lengths and sample rates. Do not hard code numbers for these – get the necessary values from the data file. Also, when developing your code, you might want to work on a short piece of the input signal rather than the full signal. The full signals have hundreds of thousands of samples, which may bog down your computer before you get things correct.

5. Comment on which approach seems to work the best.