Signal Processing First
Lab 06: Digital Images: A/D and D/A

Pre-Lab: You should read the Pre-Lab sections of this lab assignment and go over all exercises in the Pre-Lab section before going to your assigned lab session.

Verification: The pre-lab section of this lab must be completed before your assigned Lab time to receive credit and the steps marked Instructor Verification must also be signed off during the lab time. One of the laboratory instructors must verify the appropriate steps by signing on the Instructor Verification line. When you have completed a step that requires verification, demonstrate the step to the TA or instructor. Turn in the completed verification sheet to your TA when you leave the lab.

Lab Report: Your lab report should cover Sections 2 and 3 with graphs and explanations. You are asked to label the axes of your plots and include a title for every plot. In order to keep track of plots, include your plots inlined within your report.

Lab Objective

The objective in this lab is to introduce digital images as a second useful signal type. We will show how the A-to-D sampling and the D-to-A reconstruction processes are carried out for digital images. In particular, we will show a commonly used method of image zooming (reconstruction) that gives "poor" results.

1 Pre-Lab

1.1 Digital Images

In this lab we introduce digital images as a signal type for studying the effects of sampling, aliasing and reconstruction. An image can be represented as a function $x(t_1, t_2)$ of two continuous variables representing the horizontal (t_2) and vertical (t_1) coordinates of a point in space.[1] Moving images (such as TV) would add a time variable to the two spatial variables.

Monochrome images are displayed using black and white and shades of gray, so they are called gray- scale images. In this lab we will consider only sampled gray-scale still images. A sampled gray-scale still image would be represented as a two-dimensional array of numbers of the form

$$x[m, n] = x(mT_1, nT_2);\ 1 \leq m \leq M, \text{ and } 1 \leq n \leq N$$

[1] The variables t_1 and t_2 do not denote time, they represent spatial dimensions. Thus, their units would be inches or some other unit of length.

where T_1 and T_2 are the sample spacings in the horizontal and vertical directions. In MATLAB,we can represent an image as a matrix, that consists of *M* rows and *N* columns. The matrix entry at (*m, n*) is the sample value *x*[*m, n*]—called a pixel (short for picture element).

An important property of light images such as photographs and TV pictures is that their values are always non-negative and finite in magnitude; i.e.,

$$0 \leq x[m, n] \leq Xmax < \infty$$

This is because light images are formed by measuring the intensity of reflected or emitted light which must always be a positive finite quantity. When stored in a computer or displayed on a monitor, the values of x[m, n] have to be scaled relative to a maximum value X_{max} . Usually an eight-bit integer representation is used. With 8-bit integers, the maximum value (in the computer) would be $X_{max} = 2^8 = 255$, and there would be $2^8 = 256$ different gray levels for the display, from 0 to 255.

1.2 Displaying Images

The correct display of an image on a gray-scale monitor can be tricky, especially after some processing has been performed on the image. We will use the routine, *show_img.m*, to display the images, but the following points must be considered:

1. All image values must be non-negative for the purposes of display. Filtering may introduce negative values, especially if differencing is used (e.g., a high-pass filter).

2. The default format for most gray-scale displays is eight bits, so the pixel values x[m, n] in the image must be converted to integers in the range $0 \leq x[m, n] \leq 255 = 2^8$.

3. The actual display on the monitor is created with the *show_img.m*, function. The *show_img.m*, function will handle the color map and the "true" size of the image. The appearance of the image can be altered by running the pixel values through a "color map." In our case, we want "grayscale display" where all three primary colors (red, green and blue, or RGB) are used equally, creating what is called a "gray map." The function *colormap*(gray(256)) creates a linear mapping, so that each input pixel amplitude is rendered with a screen intensity proportional to its value.

4. When the image values lie outside the range [0,255], or when the image is scaled so that it only occupies a small portion of the range [0,255], the display may have poor quality.

1.3 MATLAB Function to Display Images

You can load the images needed for this lab from *.mat files. Any file with the extension *.mat can be loaded via the load command. After loading, use the command 'whos' to determine the name of the variable that holds the image and its size. Although MATLAB has several functions for displaying images on the computer, we will use *show_ img()* for this lab. This function handles the scaling of the image values and allows you to open up multiple image display windows.
Here is the help on :

```
function [ph] = show_img(img, figno, scaled, map)
%SHOW_IMG  display an image with possible scaling
% usage:   ph = show_img(img, figno, scaled, map)
%     img = input image
```

```
%     figno = figure number to use for the plot
%     if 0, re-use the same figure
%     if omitted a new figure will be opened
% optional args:
%     scaled = 1 (TRUE) to do auto-scale (DEFAULT)
%     not equal to 1 (FALSE) to inhibit scaling
%     map = user-specified color map
%     ph = figure handle returned to caller
%----
```

Notice that unless the input parameter figno is specified, a new figure window will be opened.

1.4 Printing Multiple Images on One Page

The phrase "what you see is what you get" can be elusive when dealing with images. It is VERY TRICKY to print images so that the hard copy matches exactly what is on the screen, because there is usually some interpolation being done by the printer or by the program that is handling the images. One way to think about this in signal processing terms is that the screen is one kind of D-to-A and the printer is another kind, but they use different (D-to-A) reconstruction methods to get the continuous-domain (analog) output image that you see.

Furthermore, if you try to put two images of different sizes into subplots of the same MATLAB figure, it won't work because MATLAB wants to force them to be the same size. Therefore, you should display your images in separate Figure windows. In order to get a printout with MULTIPLE IMAGES ON THE SAME PAGE, use the following procedure:

1. In MATLAB, use *show_img* and *trusize* to put your images into separate figure windows at the correct pixel resolution. The commands to display an image called *xpix* will look like

```
fignum = 1;
show_img(xpix,fignum,1, 'gray(256)')
trusize(fignum);
```

2. For each MATLAB figure window, press ALT and the PRINT-SCREEN key at the same time, which will copy the active window contents to the clipboard.

3. Use programs such as MS Word to assemble the different images onto one page.

1.5 Get Test Images

In order to probe your understanding of image display, do the following simple displays:

(a) Load and display the chainlink and shadow images from IPimages.mat. This image can be found in the MATLAB files link. The command "load IPimages" will put the sampled images into the arrays chainlink and shadow. Use whos to check the size of the images after loading. When you display the image it might be necessary to set the colormap via colormap(gray(256)). Display the image in matrix *shadow*.

(b) Use the colon operator to extract the 200th row of the "shadow" image, and make a plot of that row as a 1-D discrete-time signal.

```
shadow200 = shadow(200,:);
```

Observe that the range of signal values is between 0 and 255. Which values represent white and which ones black? Can you identify the region where the 200th row crosses the fence and its shadow?

2.0 Working with Test Images

2.1 Synthesize a Test Image

In order to probe your understanding of the relationship between MATLAB matrices and image display, you can generate a synthetic image from a mathematical formula.

(a) Generate a 256x256 test image in which all of the columns are identical by using the following outer product:

```
xpix = ones(256,1)*cos(2*pi*(0:255)/16);
```

The command, *ones*, creates a vector of all ones with the dimensions given in the argument, in this case 256 rows and 1 column. This is a useful method for creating a matrix with identical columns or rows (by post-multiplying by ones(1,ncols)).

Display the image and explain the gray-scale pattern that you see in terms of frequency. How wide are the bands in number of pixels? How can you predict that width from the formula for *xpix*?

(b) In the previous part, which data value in *xpix* is represented by white? Which one by black? Note that the values in xpix are in the range [-1,1]. The routine, *show_img* scales it to the range [0,255].

(c) Explain how you would produce an image with bands that are horizontal. Give the formula that would create a 400 × 400 image with 5 horizontal black bands separated by white bands. Write the MATLAB code to make this image and display it.

2.2 Downsampling Images

Images that are stored in digital form on a computer have to be sampled images because they are stored in an M × N array (i.e., a matrix). The sampling rate in the two spatial dimensions was chosen at the time the image was digitized (in units of samples per inch if the original was a photograph). For example, the image might have been "sampled" by a scanner where the resolution was chosen to be 300 dpi (dots per inch). If we want a different sampling rate, we can simulate a lower sampling rate by simply throwing away samples in a periodic way. For example, if every other sample is removed, the sampling rate will be halved (in our example, the 300 dpi image would become a 150 dpi image). Usually this is called *sub-sampling* or *down-sampling* or *decimation*. Downsampling throws away samples, so it will shrink the size of the image. Downsampling by a factor of *p* can be done by keeping every *p*-th sample:

```
[endx,endy]= size(ww);
wp = ww(1:p:endx,1:p:endy);
```

One potential problem with down-sampling is that aliasing might occur. This will be illustrated in a dramatic fashion in the next section. One difficulty with showing aliasing is that we must display the pixels of the image exactly. This almost never happens because most monitors and printers will perform some sort of interpolation to adjust the size of the image to match the resolution of the device. We will use the function called *trusize.m* to prevent this.

3 Sampling, Aliasing and Reconstruction

3.1 Down-Sampling

Load the IPimages.mat file which has the images stored in variables called *shadow* and *chainlink*. Now downsample the *shadow* image by a factor of 2.

(a) What is the size of the down-sampled image?
(b) Notice the aliasing in the downsampled image, which is surprising since no new values are being created by the downsampling process. Describe how the aliasing appears visually. Compare the original to the downsampled image. Which parts of the image show the aliasing effects most dramatically?
(c) This part is challenging: explain why the aliasing happens in the *shadow* image by using a "frequency domain" explanation. In other words, estimate the frequency of the features that are being aliased. Give this frequency as a number in cycles per pixel. (Note that the fence provides a sort of "spatial chirp" where the spatial frequency increases from left to right) Can you relate your frequency estimate to the Sampling Theorem?[2] You might try zooming in on a very small region of both the original and downsampled images.

3.2 Reconstruction of Images

When an image has been sampled, we can fill in the missing samples by doing interpolation. For images, this would be analogous to the examples shown in Chapter 4 for sine-wave interpolation which is part of the reconstruction process in a D-to-A converter. We could use a "square pulse" or a "triangular pulse" or other pulse shapes for the reconstruction.

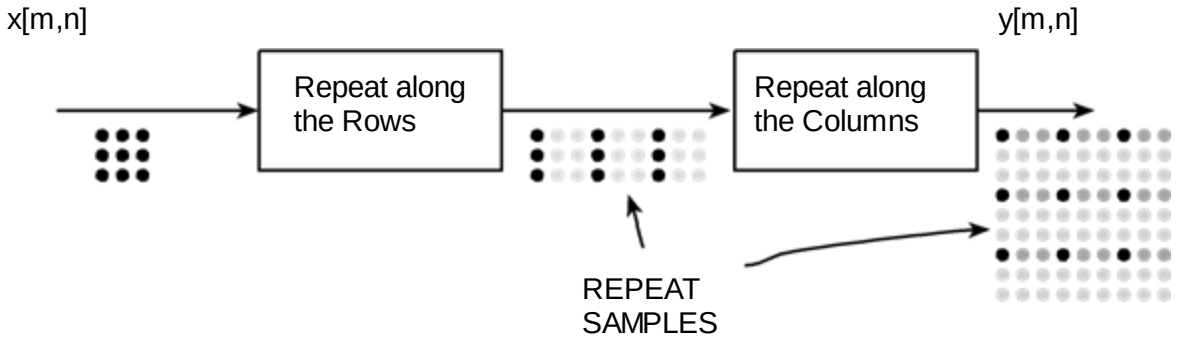

Figure 1: 2-D Interpolation broken down into row and column operations: the gray dots indicate repeated data values created by a zero-order hold; or, in the case of linear interpolation, they are the interpolated values.

For these reconstruction experiments, use the *chainlink* image, downsampled by a factor of 4. You will have to generate this by loading in the image from IPimages.mat to get the image which is in the array called *chainlink.* A down-sampled *chainlink* image should be created and stored in the variable *cl4*. The objective will be to reconstruct an approximation to the original *chainlink* image, which is 500x680, from the smaller down-sampled image.

[2] The Sampling Theorem applies to digital images, so there is a Nyquist Rate that depends on the maximum spatial frequency in the image.

(a) The simplest interpolation is reconstruction with a square pulse which produces a "zero-order hold." Here is a method that works for a one-dimensional signal (i.e., one row or one column of the image), assuming that we start with a row vector *xr1*, and the result is the row vector *xr1hold*.

```
n1 = 0:6; % Define a function with integer values from 0 to 6
xr1 = 8*cos(pi*n1/3); % Define the function to be interpolated
p = 10; % set the interpolation factor
tti = 0:1/p:6;  %-- locations between the n1 indices
xr1zoh = interp1(n1,xr1,tti,'nearest'); %-- function is INTERP-ONE
subplot(211);stem(n1,xr1);
subplot(212);stem(tti,xr1zoh);
```

Plot the vector *xr1hold* to verify that it is a zero-order hold version derived from *xr1*. Explain what values are contained in the indexing vector *nn*. If *xr1hold* is treated as an interpolated version of *xr1*, then what is the interpolation factor? Your lab report should include an explanation for this part, but plots are optional—use them if they simplify the explanation.

(b) Now return to the down-sampled *chainlink* image, and process all the rows of *cl4* to fill in the missing points. Use the zero-order hold idea from part (a), but do it for an interpolation factor of 4. Call the result *xholdrows*. Display *xholdrows* as an image, and compare it to the downsampled image *cl4*; compare the size of the images as well as their content. The interp1 function can operate on all rows or columns of a matrix at once so that you do not need to loop through the image one row or column at a time. The following code shows how to set this up for interpolating between rows. Note that interp1 operates on the first dimension.

```
[nrow,ncol] = size(cl4); % Get size of image
nr = 1:nrow; % Define a function with integer values from 0 to nrows-1
p = 4; % set the interpolation factor
nrint = 1:1/p:nrow; %-- locations between the row indices
xholdrows = interp1(nr,cl4,nrint,'nearest');
```

(c) Now process all the columns of *xholdrows* to fill in the missing points in each column and and call the result *xhold*. Compare the result *(xhold)* to the original image *chainlink*. Include your code for parts (b) and (c) in the lab report.

```
[nrow,ncol] = size(xholdrows);
nc = 1:ncol; % Define a function with integer values from 0 to ncols-1
ncint = 1:1/p:ncol;  %locations to interpolate between the column indices
xhold = interp1(nc, xholdrows',ncint,'nearest');
xhold = xhold';
```

(d) Linear interpolation can be done in MATLAB using the *interp1* function.

```
n1 = 0:6; % Define a function with integer values from 0 to 6
xr1 = 8*cos(pi*n1/3); % Define the function to be interpolated
p = 8; % set the interpolation factor
tti = 0:1/p:6;  %-- locations between the n1 indices
xr1linear = interp1(n1,xr1,tti,'linear'); %-- function is INTERP-ONE
```

Plot the vector *xr1linear* to verify that it is a linear approximation of *xr1*. For the example above, what is the interpolation factor when converting *xr1* to *xr1linear*?

(e) In the case of the *chainlink* image, you need to carry out a linear interpolation operation on both the rows and columns of the down-sampled image *cl4*. This requires two calls to the *interp1*

function, because one call will only process all the columns of a matrix. Name the interpolated output image *xxlinear*. Include your code for this part in the lab report.

(f) Compare *xxlinear* to the original image *chainlink*. Comment on the visual appearance of the "reconstructed" image versus the original; point out differences and similarities. Can the reconstruction (i.e., zooming) process remove the aliasing effects from the down-sampled *chainlink* image?

(g) Compare the quality of the linear interpolation result to the zero-order hold result. Point out regions where they differ and try to justify this difference by estimating the local frequency content. In other words, look for regions of "low-frequency" content and "high-frequency" content and see how the interpolation quality is dependent on this factor. A good way to compare the images is to take the difference between the two results (D=xxlinear-xhold) and display the resulting image.

A couple of questions to think about: Are edges low frequency or high frequency features? Are the chainlink edges low frequency or high frequency features? Is the background a low frequency or high frequency feature?

Lab 06
INSTRUCTOR VERIFICATION PAGE

For each verification, be prepared to explain your answer and respond to other related questions that the lab TA's or professors might ask. Turn this page in at the end of your lab period.

Name: Lab Section:

Part 2.1(c) Create a 400 × 400 image with 5 horizontal black bands separated by white bands. Write the MATLAB code to make this image and display it. Explain your code and the concept of "frequency" in images to your lab instructor.

Verified: Date:

Part 3.1 Downsample the *shadow* image to see aliasing. Describe the aliasing, and where it occurs in the image.

Verified: Date:

Part 3.2 b,c Reconstruct the *chainlink* image using the zero-order hold operation. Show the effect of operating on the rows separately and both the rows and the columns. Explain what you are seeing and how the zero-order operation works to the lab instructor.

Verified: Date: