EE 112: Final Project

Spring 2017

Instructor: Professor Santacruz

Group 11

Umar Sohi Alexander Yee Anthony Watson Qinglun Huang

May 15, 2017

Contents

List of Figures

Instructor Verification Sheet Answers

Part 2.1(a)

Generate and display a digital image. Explain the width of bands in the test image formed from the "outer product" of ones and a cosine. Give the width of the bands in number of pixels and explain how you can predict that width from the formula for `xpix`?

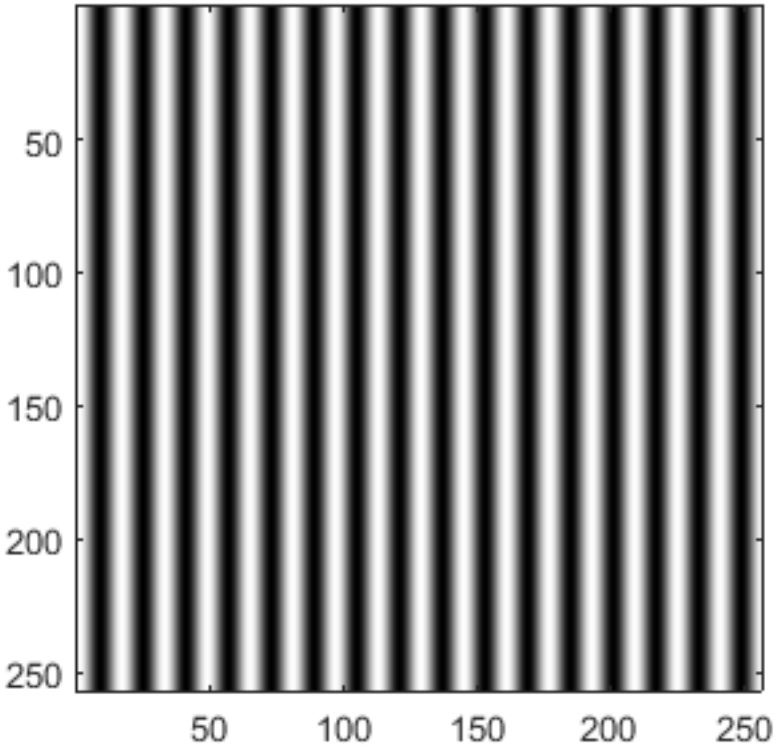


Figure 1: Cosine Outer Product, A Stacked Cosine Matrix xpix

The outer product creates a matrix, that is 256 copies of the cosine vector stacked on top of each other. It is because of this structure that the resulting image is a bunch of vertical bands.

`xpix` is a pattern of vertical black and white bands, but the edges of the bands have a "soft" transition from white to black and vice-versa. There are 16 black bands and 16 white bands (15 full bands + 2 half bands), so a total of 32 bands. Knowing that the image width is 256 pixels, gives us an 8 pixel width of $\frac{256}{32}$ per band. You can predict the width from the frequency of the cosine function used to generate xpix. We can extract the equation for the angular frequency $\omega = 2\pi f$ from the cosine in the MATLAB code:

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

We found that ω should have a frequency of 16. So we can find the period as follows:

$$\begin{aligned}\omega &= 2\pi f \\ &= \frac{2\pi}{16}\end{aligned}$$

$$\begin{aligned}\Rightarrow T &= \frac{1}{f} = \frac{1}{\frac{1}{16}} \\ &= 16\end{aligned}$$

Part 2.1(c)

Create a 400 x 400 image with 5 horizontal black bands separated by white bands. Write the MATLAB code to make this image and display it.

The easiest way to produce a similar image with horizontal bands instead of vertical ones is to use MATLAB's transpose operator on our xpix matrix, which is implemented using an apostrophe.

```
ypix = ones(400,1)*cos(2*pi*(0:(400-1))/(400/5));
ypix = ypix'; % transpose
```

Using this operator, we are able to transform a 400 x 400 image with vertical bands into one with horizontal bands instead. We can do this because the outer product creates a matrix that is 256 copies of the cosine vector stacked on top of each other. It is because of this structure that the xpix image is a bunch of vertical bands. The transpose operator remaps the elements of the original matrix such that $\left[A^T\right]_{ij} = [A]_{ji}$. Essentially each row becomes a column, and each column becomes a row. So if the xpix matrix creates vertical bands column wise, the transpose operator will make the vertical bands horizontal.

$$ypix = \begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix}\left[\cos(\frac{2\pi\cdot 0}{80})\cdots\cos(\frac{2\pi\cdot k}{80})\cdots\cos(\frac{2\pi\cdot 399}{80})\right]$$

$$ypix = \begin{bmatrix} \cos(\frac{2\pi \cdot 0}{16}) & \cdots & \cos(\frac{2\pi \cdot k}{80}) & \cdots & \cos(\frac{2\pi \cdot 399}{80}) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \cos(\frac{2\pi \cdot 0}{80}) & \cdots & \cos(\frac{2\pi \cdot k}{80}) & \cdots & \cos(\frac{2\pi \cdot 399}{80}) \end{bmatrix}$$

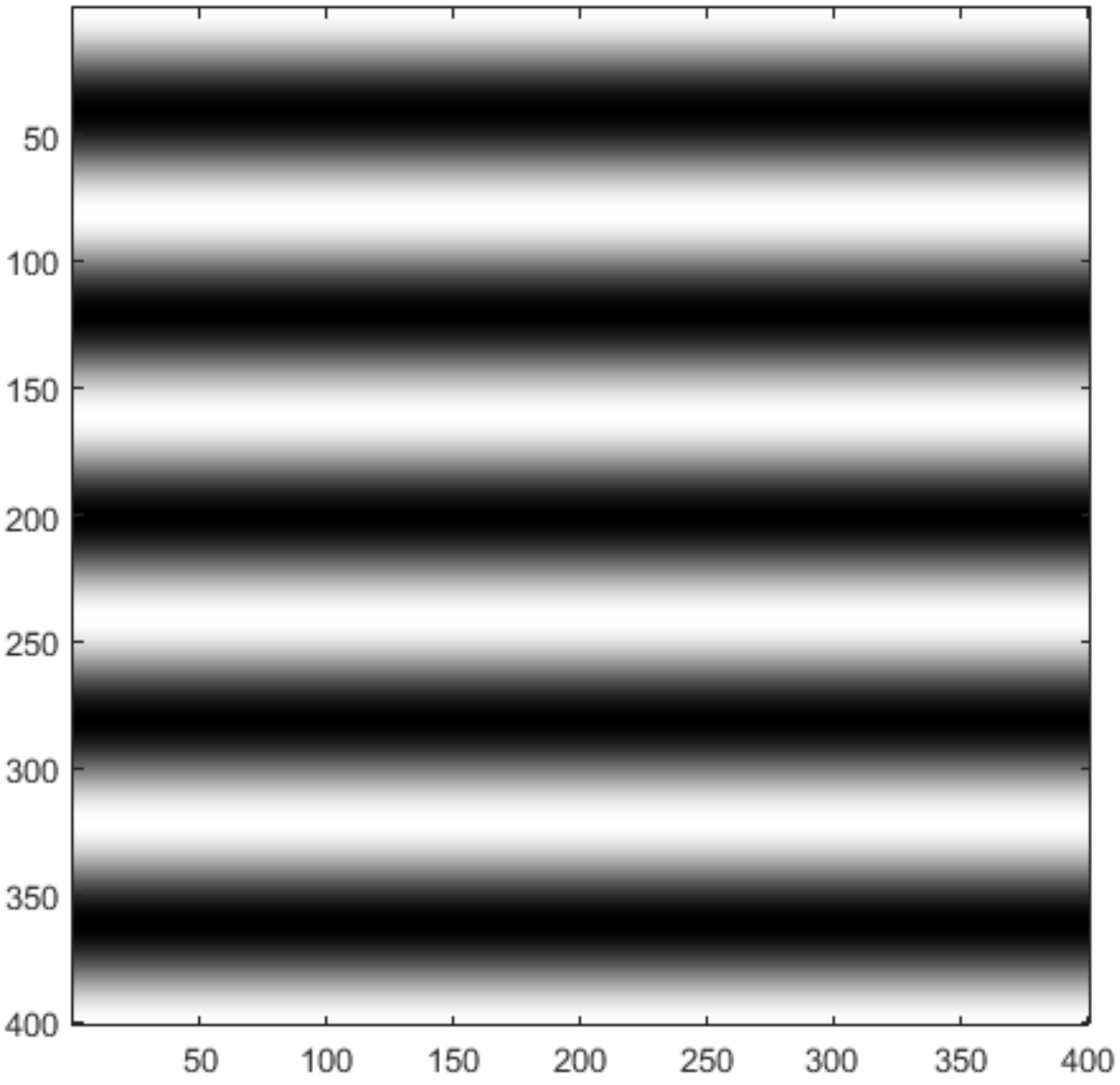


Figure 2: 400x400 Horizontal Bands Matrix

Looking at the formula for `ypix`, the biggest change we made to the given formula, `xpix`, was changing the

denominator in the cosine function. To get 5 black bands, we needed to divide our 400 pixel image into 5 sections. $\frac{400}{5} = 80$ pixels, which was the period we implemented into our cosine function.

Part 2.3(a)

Downsample the `lighthouse` image to see aliasing. Describe the aliasing, and where it occurs in the image.

Using the given `show_img()` function we can view the original lighthouse image with true pixel representation ratios.

```
load lighthouse
show_img(ww);
```

Figure 3: Original Lighthouse Image

To downsample the `lighthouse` image we use:

```
p = 2;  % downsampling factor
wp = ww(1:p:end,1:p:end);
```

The `1:p:end` selects every p^{th} value of the original lighthouse image yielding a downsampled version of `ww`.

Figure 4: Downsampled Lighthouse Image with p = 2 factor

The white fence in Figure 4 has a dark background, which results in an alternating pattern of light and dark pixels. Due to the angle of the fence, the distance between each fence post generally decreases as you move from left right. Thus, the frequency of this alternating pattern increases as we move to the right along the fence. Downsampling the original image is equivalent to capturing the image with a lower sampling rate. These high frequency portions of the image are not properly caught by our sampling rate, resulting in aliasing. Essentially, downsampling causes severe blurring effects in the aforementioned regions.

Lab Exercises

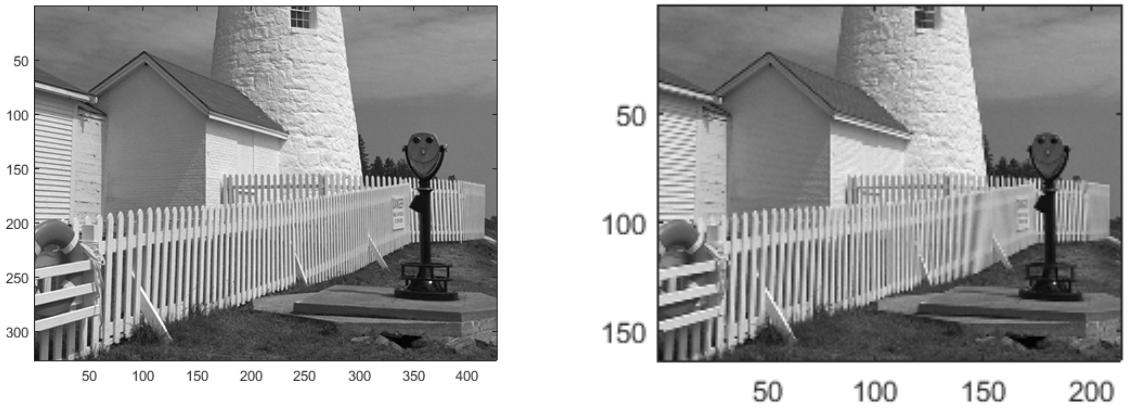
Section 3.1

Part 3.1(a)

Describe how the aliasing appears visually. Compare the original to the downsampled image. Which parts of the image show the aliasing effects most dramatically?

The downsampled image (in unscaled form) looks almost identical to the original `lighthouse` image, just

half the size of the original. The biggest difference is that portions of the downsampled image aren't as "crisp" as the original, the most noticeable being portions of the fence, where aliasing causes it to look blurry.



(a) Lighthouse Original Image

(b) Downsample p = 3 factor

Figure 5: Comparison of Lighthouse with p = 2 factor Downsampling

Part 3.1(b)

This part is challenging: explain why the aliasing happens in the lighthouse 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? You might try zooming in on a very small region of both the original and downsampled images.

Frequency, in our image, refers to the transition between white and black pixels. For example, the fence posts and the spaces in between each post are white and black, respectively. Essentially, this creates a high frequency region, where the frequency is equal to the number of posts per pixel. It is important to note that the fence's "spatial frequency" increases as you move towards the right. Particularly in the regions between pixels 140-180, we notice that the image is significantly blurred. This is the visual representation of our digital signal being aliased. We can be more precise in this discussion by estimating this region's frequency, in cycles of fence posts per pixel.

Using the following we can analyze the plot values along the fence posts.

```
plot(ww(200,:));    % plot the values of the 200th row
```

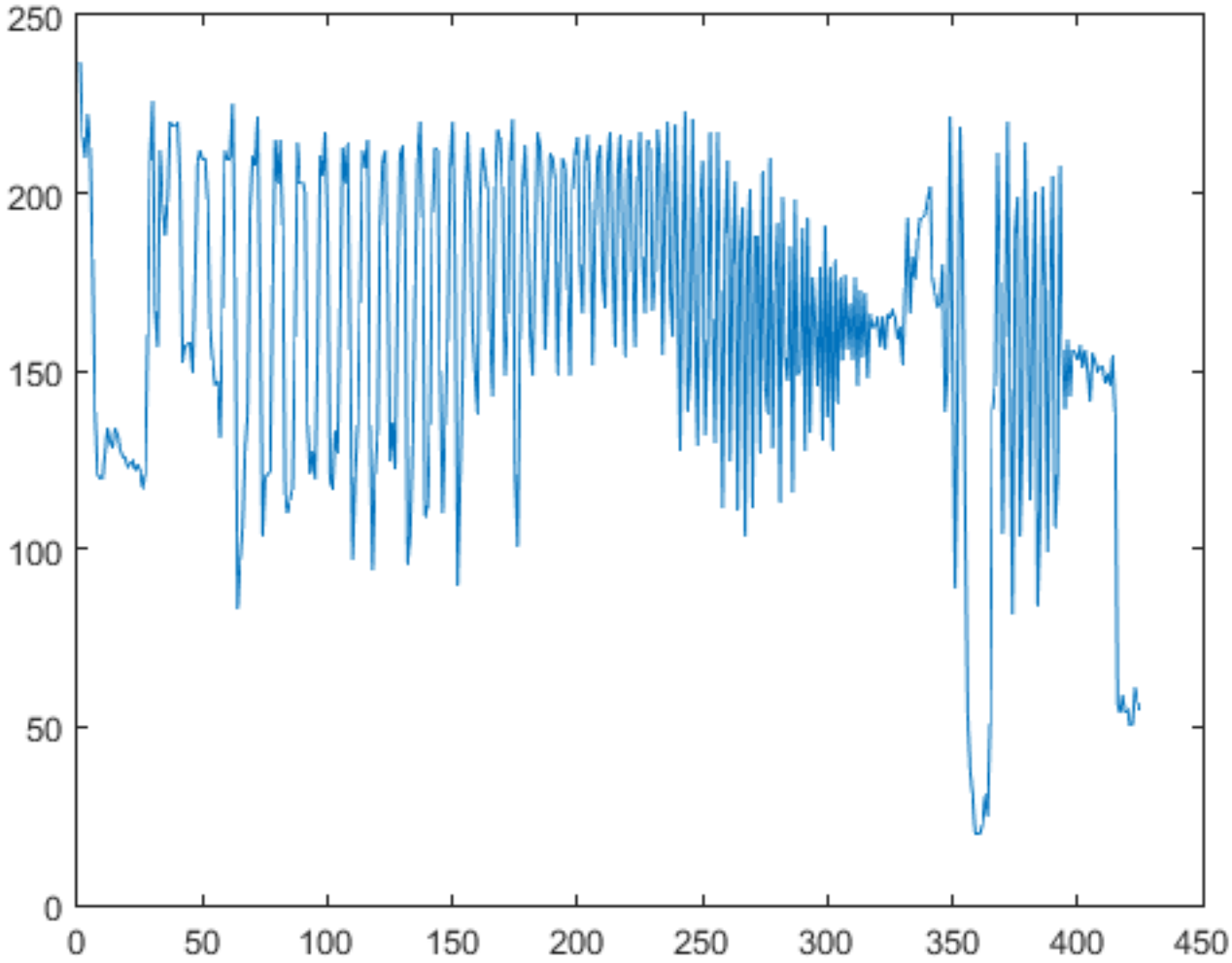


Figure 6: Plot of Intensity Values of Lighthouse Image at 200th Row

Figure 6 allows us to determine the frequency in the left, low frequency, and right, high frequency, regions. 34 troughs were counted between pixels 50-250, resulting in a $34/200 = 0.17$ cycles per pixel frequency in the left half of the fence. The right, aliased, half of the fence had about 35 troughs from pixels 250 -350 resulting in a frequency equal to $35/100 = 0.35$ cycles per pixel. This "Spatial Chirp," or change in frequency across a space, is clear here, as the frequency nearly doubles.[1] Our original image had a high enough sampling rate, so the high frequency portions of the image were properly displayed without aliasing. In the downsampled version of the original image, the sampling rate was halved, and wasn't large enough to display the high frequency portions of the image without aliasing. The sampling theorem supports this understanding, as our

[1]Backround information from: http://frog.gatech.edu/Pubs/Gu-SpatialChirp-OptComm-2004.pdf, "Spatial chirp in ultrafast optics" by Gu, Akturk, Trebino.

input signal (fence posts') frequency becomes greater than 2 times our sampling rate.

Section 3.2 Reconstructing Images

For these reconstruction experiments, use the lighthouse image, down-sampled by a factor of 3 (similar to what you did in Section 2.3). Perform this down-sampling after loading in the image from lighthouse.mat and saving it in the array called xx. A down-sampled lighthouse image should be created and stored in the variable xx3. The objective will be to reconstruct an approximation to the original lighthouse image, which is 256 x 256, from the smaller down-sampled image.

Part 3.2(a)

The simplest interpolation would be 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.

```
xr1 = (-2).^(0:6);
L = length(xr1);
nn = ceil((0.999:1:4*L)/4); %<-- Round up to the integer part
xr1hold = xr1(nn);
```

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.

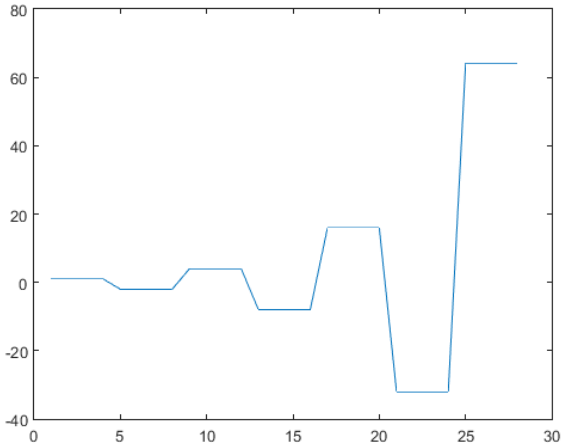


Figure 7: xr1hold Plot

The indexing vector nn contains the indices of values from xr1 to be stored in xr1hold. For example, if the 5th value of nn is 1, that means the 1st value of xr1 will be stored as the 5th value of xr1hold.

The interpolation factor of xr1hold is 4, meaning that xr1hold "holds" each value of xr1 4 times. So if xr1 contains 1, 2, 3, xr1hold would contain 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3.

Part 3.2(b)

Now return to the down-sampled `lighthouse` image, and process all the rows of `xx3` to fill in the missing points. Use the zero-order hold idea from part (a), but do it for an interpolation factor of 3. Call the result `xholdrows`. Display `xholdrows` as an image, and compare it to the downsampled image `xx3`; compare the size of the images as well as their content.

Figure 8: Downsampled Lighthouse Image with p = 3 factor (xx3)

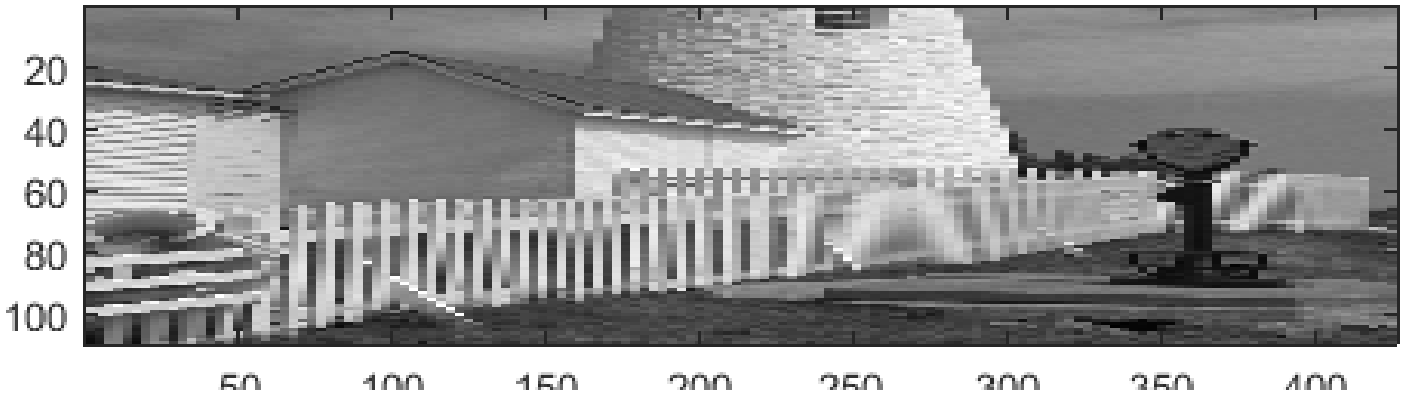


Figure 9: Interpolation using zero-order hold across rows

The size of xholdrows is 109 x 426 (rows x columns), whereas xx3 is 109 x 142. Visually, xholdrows looks like xx3 has been "stretched" to 3 times the original width, but having the same height. Sure enough, zooming in on xholdrows reveals that each pixel has been "held" 3 times horizontally instead of just once like in xx3.

Part 3.2(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 lighthouse. Include your code for parts (b) and (c) in the lab report.

Using the following code, we expanded the lighthouse image row-wise:

```
load lighthouse;
p = 3;
xx3 = ww(1:p:end, 1:p:end);
SIZE = size(xx3);

R = SIZE(1); % rows
C = SIZE(2); % columns
xholdrows = zeros(p, p*C); % 3.2(b)
xhold = zeros(p*R, p*C); % 3.2(c)

% zero-hold interpolation across rows
for i = 1:R                        %  R = number of rows
    nn = ceil((0.999:1:p*C)/p);     % C = number of columns
```

```
    temp = xx3(i,:);                % p = downsampling factor
    xr1hold = temp(nn);
    xholdrows(i,:) = xr1hold;       % result for 3.2(b)
end
    xhold(i,:) = xr1hold;
```

We expand column-wise to generate a full interpolated image, approximating the original lighthouse:

```
for i = 1:p*C
    mm = ceil((0.999:1:p*R)/p);
    temp = xhold(:, i);
    xr2hold = temp(mm)';
    xhold(:,i) = xr2hold; % result for 3.2(c)
end
```

Figure 10: Zero-hold Interpolated Lighthouse from p = 3 Downsampling (xhold)

Compared to the original lighthouse image (ww), xhold looks to be about the same size, but basically a very pixelated version of ww. Additionally, even though xhold has been rescaled back to the size of the original image, it still retains the aliasing (blurred portions of the fence) that appeared due to downsampling ww. This makes sense, since the "hold" method of interpolation isn't doing anything to estimate the missing pixels when scaling to a new size.

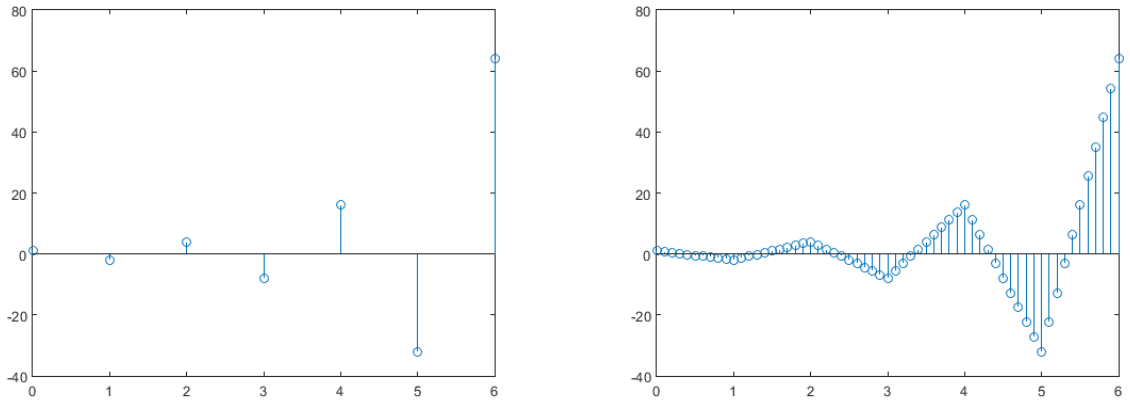
Part 3.2(d)

Linear interpolation can be done in MATLAB using the interp1 function (that's "interp-one"). Its default mode is linear interpolation, which is equivalent to using the '*linear' option, but

interp1 can also do other types of polynomial interpolation. Here is an example on a 1-D signal:

```
n1 = 0:6;
xr1 = (-2).^n1;
tti = 0:0.1:6; %-- locations between the n1 indices
xr1linear = interp1(n1,xr1,tti); %-- function is INTERP-ONE
stem(tti,xr1linear)
```

For the example above, what is the interpolation factor when converting xr1 to xr1linear?



(a) xr1, Signal of -2^{n_1} (b) Interp1 Built-in Function Demo (xr1linear)

Figure 11: MATLAB Built-in Interpolation Function Example interp1()

The interpolation factor when converting xr1 to xr1linear is 10. Looking at the stem plots, xr1 has a point at 0, 1, 2, 3, 4, 5, and 6, for a total of 7 points. xr1linear has these same 7 points, but an additional 9 points between each original point. This interpolation factor of 10 (9 + 1) can also be seen by tti, which increases the number of points between each value by 10 (1/0.1).

Part 3.2(e)

In the case of the lighthouse image, you need to carry out a linear interpolation operation on both the rows and columns of the down-sampled image xx3. 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.

```
load lighthouse;
p = 3;
xx3 = ww(1:p:end, 1:p:end);
SIZE = size(xx3);

R = SIZE(1);
C = SIZE(2);
xxlinear = zeros(p*R, p*C); % 3.2(e)
n1 = 1:C;
n2 = 1:R;
tt1 = 1:.3310:C;
tt2 = 1:.3310:R;

for i = 1:R
    xr1linear = interp1(n1, xx3(i,:), tt1);
    xxlinear(i,:) =  xr1linear;
end

for i = 1:p*C
    xrlinear = interp1(n2, xxlinear(1:R,i), tt2);
    xxlinear(:,i) = xrlinear; % result for 3.2(e)
end
```

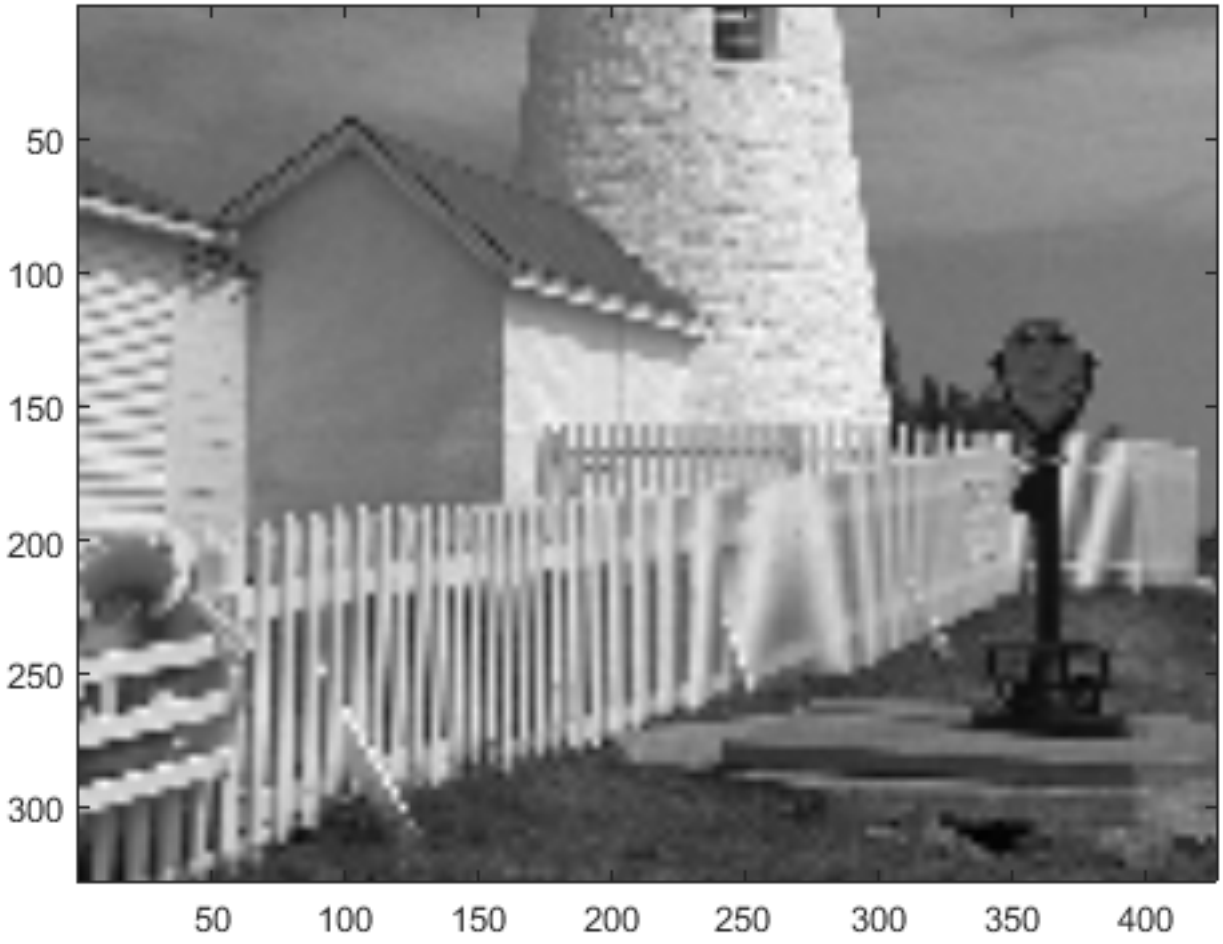
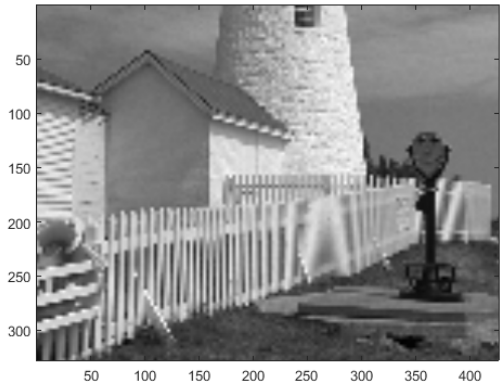


Figure 12: interp1() Interpolated Lighthouse from p = 3 Downsampling (xxlinear)

Part 3.2(f)

Compare xxlinear to the original image lighthouse. 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 lighthouse image?

(a) Original Lighthouse



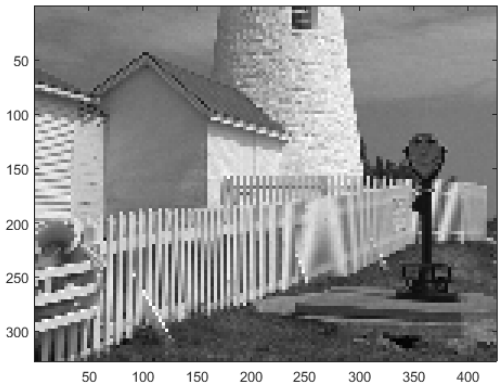
(b) Interp1 Interpolated Lighthouse (xxlinear)

Figure 13: Comparison of Lighthouse w/ Interp1 Lighthouse

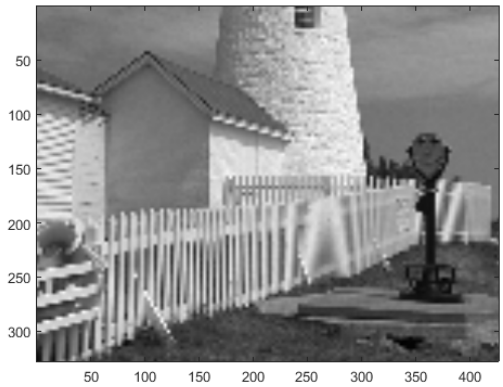
Both the original image (ww) and the reconstructed one (xxlinear) are the same size, but xxlinear still shows some pixelation and appears "blurry" compared to ww. Also, xxlinear still has the same blurred fence regions as the down-sampled lighthouse image, so the reconstruction process (at least using the interp1 function) can't remove the aliasing effects.

Part 3.2(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) Zero-hold Interpolation (xhold)



(b) Interp1 Interpolation (xxlinear)

Figure 14: Zero-Hold Intperolation vs Interp1 MATLAB Interpolation

The linearly interpolated image, xxlinear, is an improvement over the zero-order hold image, xhold. The interpolation in xxlinear manages to reduce pixelation of xhold, making the resulting image more like the original lighthouse, but is still a bit blurry in comparison. However, both xxlinear and xhold both have the same aliasing effects from the down-sampled image.

In xhold, the closest portion of the fence (near the lifesaver) is hard to make out; it almost doesn't look like a fence due to the high frequency components of it. In xxlinear, however, that same portion of the image looks much more like a regular fence (although not as good as the original image). This is because the linear interpolation has changed this section of the image from high frequency content to low frequency.

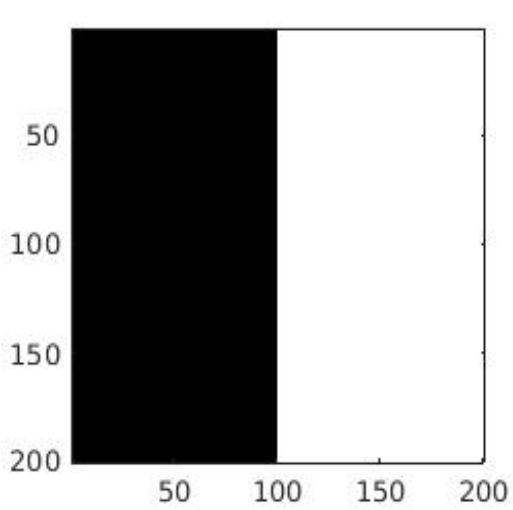
A couple of questions to think about: Are edges low frequency or high frequency features? Are the fence posts low frequency or high frequency features? Is the background a low frequency or high frequency feature?

The relationship between frequency and spatial variables in an image is closely tied to the fourier representations of the spatial function $f(x, y)$. To demonstrate a simplified example of this relationship, consider fixing $y = k$ where $k = constant$. To simulate this use the following MATLAB code:

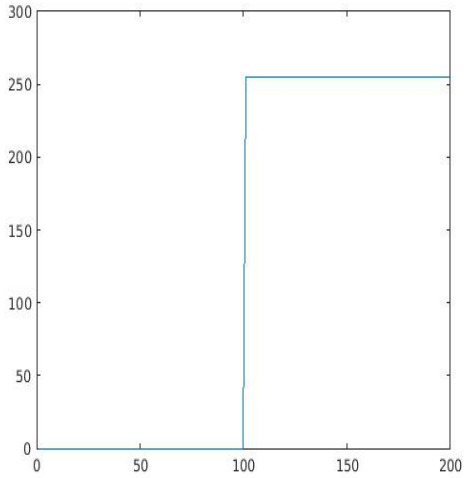
```
% example pics and code for frequency to spatial coordinate relations
R = 10;
C = 10;

image = zeros(200,100);
image = [image, 255 * ones(200,100)];

show_img(image) % shows edge example image
plot(image(100,:)) % shows waveform of intensity values for a simulated edge
```



(a) Example of an edge in an image



(b) Edge's spatial waveform of intensity values for row at fixed height k

Figure 15: Example of an edge in an image

Notice that the rate of change of the intensity curve in figure 15b is asymptotic near the edge occurance. In lecture we discusses how waveforms like this would require an infinite number of terms in the equivalent Fourier Series representation of the signal. Consider the less severe case: an edge with $\left|\frac{df(k,y)}{dy}\right| \rightarrow$ large. Edges that are more gradual but still have $\left|\frac{df(k,y)}{dy}\right| \rightarrow$ large, will have its number of sinusoidal terms approaching ∞, as it becomes like the step function. For these kinds of $f(x,y)$, the equivalent Fourier Series representation will require many high frequency components / sinusoidal waves to build the edge behavior. Each successive sinusoidal term is an increasing multiple of the fundamental frequency.

$$f(k,y) = A_0 + \sum_{n=1}^{N} A_n cos(2\pi n f_0 y + \phi_n)$$
$$N = Large$$

In this way, we can relate the frequency to the spatial function's derivative values. For

$$\left|\frac{df(k,y)}{dy}\right| \rightarrow \text{large}$$
$$\Rightarrow N \rightarrow \text{large in } f(k,y) = \left(A_0 + \sum_{n=1}^{N} A_n cos(2\pi n f_0 y + \phi_n)\right)$$

It is because of this relationship that edges are considered high frequency features. An example of high frequency features would be the fence posts in the lighthouse image. Low frequency areas of images are areas

where the color/intensity doesn't change drastically, i.e. smoother transition of color intensity values. For instance, the roof of the background building would be a region of low frequency features. This is because all the pixels in the roof are about the same value in light intensity, i.e. $\left|\frac{df(k,y)}{dy}\right| \rightarrow$ small. The inverse relationship between spatial function rates of change and frequency also hold.

The best way to think about it is we are capturing light waves that are translated into values between 0 and 255. Once our digital processor/computer reads in values from an image, each pixel is assigned a higher value for white spaces (more light) or a lower value for darker spaces (less light). So when we look at edges, the abrupt change in values from light to dark or vice versa means that we are transitioning between high and low values (255 to 0). It is high frequency because physically, the object's steep difference in light intensity means a lot of information must be transmitted over small area.

Regions of low frequency are areas that have only small changes with respect to the pixels/light representation of objects around them. Visually this translates to areas of similar color and light intensity (although we only care about light intensity if we are working with monochrome colors). These are low frequency regions as there is little information change in the region. Hence, the light needs to transmit less data in the region and thus a lower frequency wave is used.