CURIE Day 3: Frequency Domain Images

Curie Academy, July 15, 2015

NAME: _____

NAME: _____

TA SIGN-OFFS

Exercise 7	Making 8x8 pictures	
Exercise 13	Compressing a grayscale image	
Exercise 17	Satellite image debanding	

As you learned in the lecture earlier, the Discrete Cosine Transform (DCT) is a way to map an image from the pixel domain to the frequency domain. The transformed image (which lies in the frequency domain) can be thought of as being composed of many wave components, each with different frequencies. We will explore these concepts visually in Matlab. Once we have the concepts down, we'll look at how the DCT lets us compress images to a small fraction of their original size. Feel free to browse through the Matlab reference sheet if you need to refresh yourself on Matlab commands discussed so far. You can also ask the TAs for help if you get stuck at any part of this activity and/or if you need further clarification.

Not every little exercise needs to be checked by a TA, but *make sure to get a TA's initials* after you've finished each of the bigger exercises listed on this cover sheet.

1 Familiarizing yourself with DCT

Let's start by getting our hands dirty with the Matlab command for the **discrete cosine transform** in two dimensions, `dct2`. This Matlab command takes in an image matrix in the pixel domain of size $m \times n$ and outputs an image matrix in the frequency domain also of size $m \times n$. As an example, let us compute the DCT of a matrix in the pixel domain given by

$$A = \begin{bmatrix} 3.5 & 2 & 6 \\ 1 & 4 & 6.8 \\ 10.1 & 2 & 7.5 \end{bmatrix}.$$

You can convert matrix `A` into an image `A_f` in the frequency domain by typing:

```
>> A = [3.5 2 6; 1 4 6.8; 10.1 2 7.5];
>> A_f = dct2(A)
```

What does it mean for `A_f` to be in the frequency domain? Recall that DCT decomposes an image into various wave components. Each entry of the matrix `A_f` corresponds to a wave component of a particular frequency. As you go down a row, or as you go across a column from left to right, the frequency of the wave component increases. Consequently, the upper left portion of the matrix `A_f` represent wave components of low frequency while the lower right portion of the matrix `A_f` represent wave components of high frequency.

What do the numbers in the matrix `A_f` mean then? You can think of `A_f(i,j)` as the "weight" of the wave component with row-frequency $i-1$ and column-frequency $j-1$, i.e. how much this particular wave component contributes to the image in the pixel domain.

Key Fact 1 *This reminds us of a concept introduced in the lecture in which an image can be decomposed as a weighted sum of wave components of various frequencies! More specifically, if* `A_f(i,j)` *is equal to 0, the wave component associated to it does not contribute to the image at all.*

Exercise 1 *Consider a* 2×2 *image of a wave component with row-frequency 1 and column-frequency 0. How do you expect the DCT of this image to look like? (We'll show you how to check this in a moment).*

Answer:

$$\begin{bmatrix} & \\ & \end{bmatrix}.$$

Since we've been talking about these wave components a lot, it would be helpful to produce plots of them to see how they look like. This will be the focus of the next section. But before that, try to think about this exercise first:

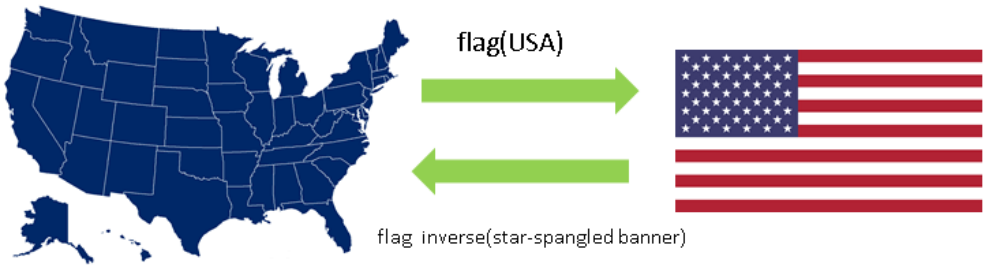
Exercise 2 *Without using Matlab, what do you think the DCT of* `2*A` *look like? Check your guess afterwards by performing the relevant Matlab computations.*

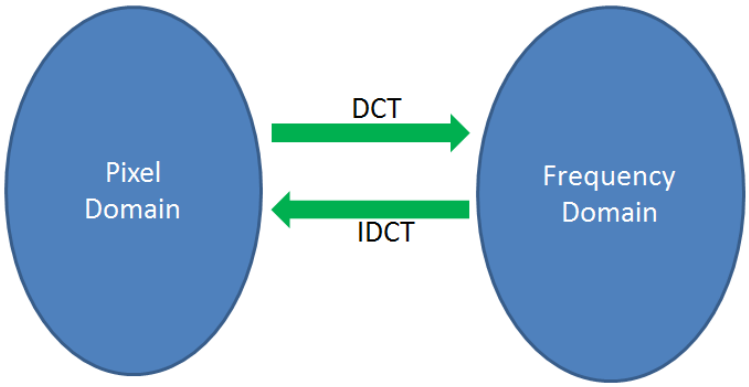
Answer:

2 Plotting wave components through the IDCT

Next we will visualize wave components with the help of Matlab. To do this, we will need the **inverse discrete cosine transform**, or IDCT.

The IDCT operates in the same way as the inverse of a function in math does. Take $f(x) = 2x + 3$ for example. If we evaluate $f(1)$, we get 5, i.e. the function f maps the input 1 to the output 5. Consequently, the inverse function of f maps 5 into 1. This example is boring, however, since 1 and 5 are both in the domain of real numbers. Instead, let us consider the `flag` function which maps a country into its flag. This function operates by taking an input from the country domain and producing an output in the flag domain. The inverse of `flag` then takes an input in the flag domain and yields an output in the country domain.





In a similar fashion, as DCT maps an image in the pixel domain to an image in the frequency domain, IDCT maps an image in the frequency domain to an image in the pixel domain. In Matlab, the command we use to apply IDCT to an image is given by `idct2`. As an example, if we were to execute

```
>> A_f = dct2(A);
>> A_new = idct2(A_f); % A_new = A
```

we would get the matrix `A` back.

How do we use this concept to produce an image of a specific wave component?

For now, we will be working with images of size 8×8. Let `A` be an image matrix in the pixel domain and `A_f` be its DCT. If all entries of `A_f` are zero except for `A_f(i,j)`, this implies that only the wave component with row-frequency $i - 1$ and column-frequency $j - 1$ is present in the decomposition of the image `A`. This tells us that the image `A` is just a multiple of the image matrix of the aforementioned wave component. So, in order to produce a plot of a specific wave component, we can start with an image in the frequency space, make all entries except one equal to 0, and perform IDCT to recover the wave component.

Let's now try to do this in Matlab. To get started, open the Matlab script `visualize_wave_components.m` which can be found in the day 3 folder at `http://www.cs.cornell.edu/workshop/curie2015`. This script creates a blank image `img_f` in the frequency space through the command:

```
>> img_f = zeros(8,8);
```

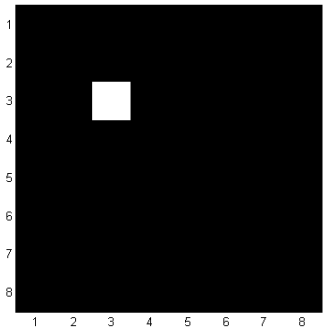
Since we are interested in only one wave component, we modify only one entry of `img_f` and make it non-zero. Suppose that we want to visualize the image of a wave component with row-frequency 2 and column-frequency 2. We only have to change the value of `img_f(3,3)` and set it to 1.

Exercise 3 *In the above, why do we set* `A_f(3,3)` *to 1, when we are interested in row and column frequencies being 2? Shouldn't we set* `A_f(2,2)` *to 1 instead? (Hint: What is the lowest possible frequency?)*

Answer:

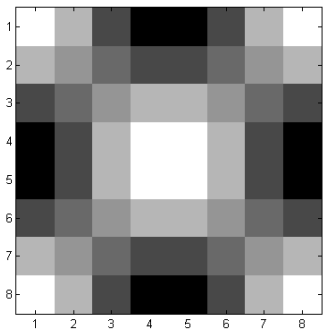
Exercise 4 *One section of the code in* `visualize_wave_components.m` *is missing. Fill in the missing part of the given code in order to draw pictures of the wave component with row and column frequencies 2.*

Once you complete exercise 4, running the script will call `imagesc(img_f)`, and you will see this plot of the image in the frequency space:



As a side note, since we aren't done walking through the code yet, we included the Matlab command `pause` to prevent Matlab from running the rest of the script, since we haven't discussed it all yet. Once you're ready to proceed, just hit enter on the command line.

The next thing the script does is to convert `img_f` into a frequency domain image with `idct2`, and then it shows us the image. If everything goes well, the plot of the wave component with both row- and column-frequency 2 should look like



Why do we say this has row and column frequencies 2 and 2? For the DCT wave components, the frequency is that number of times that the wave passes from high to low or from low to high.

Now it's your turn to try things out! Complete the following exercises:

Exercise 5 *In the code above, what do you think would happen if instead of setting* `img_f(3,3)=1`, *we had* `img_f(3,3)=3.14`*? Verify your answer by modifying the Matlab code provided.*

Answer:

Exercise 6 *Modify* `visualize_wave_components.m` *to plot images of different wave components. Can you produce plots of the wave component with the lowest frequency and the wave component with the highest frequency?*

Checkpoint: get a TA to sign off for your answers in this section.

Exercise 7 *Challenge: Modify* `visualize_wave_components.m` *to try to come up with a matrix* `img_f` *in the frequency domain whose corresponding image in the pixel domain contains the following patterns: two vertical lines? diagonal lines? something circular?*

3 Basics of Image Compression

Now that we have played around with the DCT wave components, it's time to use them to do something useful—image compression. As some of you may have experienced, uploading a high quality image into Facebook may take some amount of time, depending on the internet traffic. Did you realize that these images are already cleverly compressed to take up less than 10% as much space as an array of pixels would? You can only imagine how long you'd have to wait if you send multiple uncompressed high resolution photos of your family vacation to your relatives over e-mail.

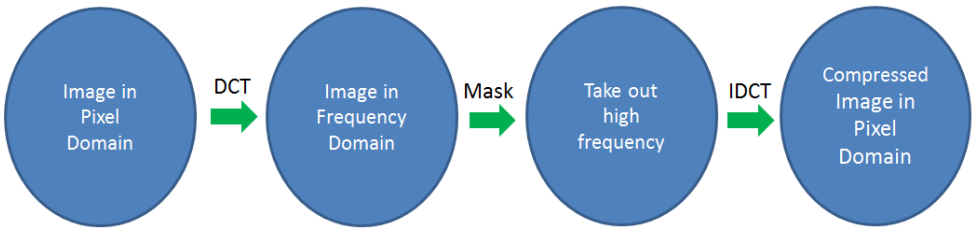
On a more serious note, image compression is widely used in addressing different challenges associated with transmitting data in the real world. Because of the difference in medium and other factors, there are difficulties and impediments of sending data from a facility or instrument underwater to a facility or instrument onshore. For example, transmitting numerous underwater images are essential for offshore drilling, for searching objects on the seafloor, or for detecting military submarines. If it would take too long for images to be sent from an underwater camera, we might not have access to relevant timely information which would aid in making decisions.

Aside from underwater image transmission, another cool application of image compression arises in sending images taken by satellites or rovers from outer space. As most of you are probably familiar with, Curiosity rover has been patrolling Mars since 2012 and has been taking images of Martian landscape for scientific research. Have you ever thought how data is relayed from outer space? In an activity later on in this lab, we will illustrate the importance of image compression by making quantifiable estimates of how long it takes to send a picture from Mars!

Here's the insight that we'll base everything on:

Key Fact 2 *The human eye is most sensitive to low frequency details. We can take out the high frequency content and then apply IDCT to recover a compressed image.*

Here's a flowchart of how this process ought to look:



Exercise 8 *Suppose you are writing a system to upload photos to social media. At what point in the flowchart would you transmit data? What would the receiver have to do to view the photos?*

Answer:

In what follows, we will walk you through how to carry this out in Matlab. To get started, open the Matlab script (`compress_image.m`. The next few paragraphs will walk through this script and explain what it does.

Let's work with the image of a mosaic called `'mosaic.jpg'`. In order to read this into Matlab, we use `imread` which converts the image into a matrix whose entries range from 0 to 255, and then we apply `im2double` to rescale the values into the range 0 to 1. The above operation looks like:

```
>> img = im2double(imread('original.jpg'));
```

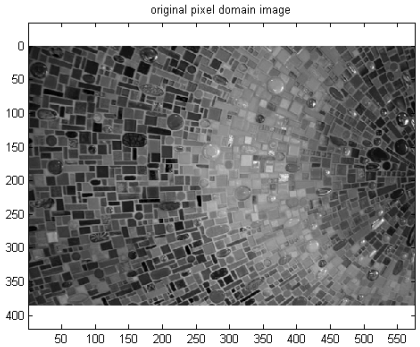
In this activity, we want to focus on grayscale images first. (You can look at color image compression for a final project.) Since our mosaic photo is in color, we can convert this into grayscale by typing the following:

```
>> img = rgb2gray(img);
```

Before processing the image, let us try to see what it looks like first. The following plot commands should be mostly familiar to you by now:

```
>> figure;
>> imagesc(img)
>> colormap(gray)
>> axis('equal')
>> title('original pixel domain image')
```

Notice that we used `axis('equal')` instead of `axis('square')`. Can you point out the difference between the two? Here's what the image is supposed to look like:



Now that we have our image in the pixel domain all set up, the next thing to do is to apply DCT to `img`

```
>> img_f = dct2(img);
```

Exercise 9 *The scripts draws* `img_f`*. Unfortunately, we can't see much! Try adding some debugging lines to get more info: how do the magnitude of the entries in the upper left corner compare with the entries in the lower right corner?*

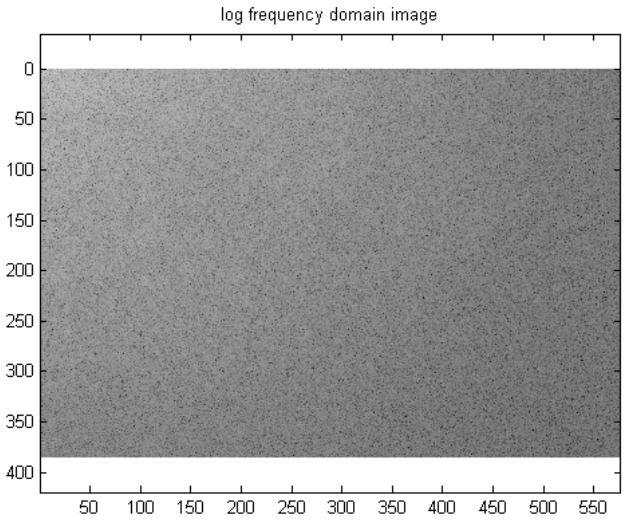
Answer:

The problem with the image in the frequency domain is that some of the entries in the matrix are substantially larger than the others. Because of this, you mostly see black. In order to be able to see what's going on in our frequency domain image, we'd like to rescale the values in `img_f`. A commonly used function for this is the base-2 logarithmic function, `log`. The command `log(img_f)` takes the logarithm of every element in the matrix `img_f`. Try applying this change to the code above.

Why do you think there is an error when `log` is applied to `img_f`?

If you guessed it right, that's because `img_f` has negative values which causes `log` to fail. To fix this, we can take the absolute value of `img_f` before applying the logarithm. As a summary, we should instead use `imagesc(log(abs(img_f)));` in the code above. If everything goes well, here's what it's supposed to look like:

Exercise 10 *Modify* `compress_image.m` *so that the plot of* `img_f` *is scaled with the* `log` *function. Do you get the picture below?*



Here comes the interesting part. In order to compress the image, we can take out the high frequency content since, as you may have noticed, most of the weights of large magnitude reside in the upper left corner of `img_f`. Recall that the upper left corner corresponds to wave components of low frequency.

To do this, we perform a mask operation. In a nutshell, our aim is to make a huge chunk of the entries of `img_f` equal to 0 with the help of the matrix `mask`. This matrix is

of the same size as `img_f` with entries equal to 0 or 1, i.e. a binary mask. If `mask(i,j)=1`, this means that we wish to retain the information contributed by the wave component with row-frequency $i-1$ and column-frequency $j-1$ and if `mask(i,j)=0`, we discard it. Here are two possible ways we could go about it.

A rectangular block mask is one of the easiest to create. The script `compress_image.m` comes with code for this already written. Since we want to preserve the upper left corner of `img_f`, `mask` has to be created such that a portion of the upper left corner of `mask` is equal to 1 with the rest of the entries being equal to zero. To set this up, we first specify row and column numbers, `m_trunc` and `n_trunc`, up to which we wish to preserve the entries of `img_f`. Take note that these variables cannot take values larger than the number of rows and columns of `img_f`. Once we have set these values, we declare a matrix of zeros of the same size as `img_f` and set the values in the sub-matrix defined by the first `m_trunc` rows and `n_trunc` columns to 1. In Matlab code, here's what it looks like:

```
>> mask = zeros(size(img_f));
>> m_trunc = 100;
>> n_trunc = 125;
>> mask(1:m_trunc,1:n_trunc) = 1;
```

You might want to make a plot of how the `mask` matrix appears to visualize what's going on.

A diagonal block mask is trickier to create. This is used when we only desire to keep frequencies in the upper left triangular corner. The matrix

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

is an example of a 4×3 diagonal block matrix. Notice that there are other possibilities as well. We let the variable `num_diagonal_rows` represent the row or column index (doesn't matter which one) up which to preserve entries of `img_f`. In the above example, `num_diagonal_rows` is equal to 3. Observe that the value of `num_diagonal_rows` cannot exceed the minimum between the number of rows and the number of columns of `img_f`.

Exercise 11 *Fill in the missing part of* `compress_image.m` *to construct a diagonal mask. Comment out the rectangular mask and add your new code in the indicated place. Don't forget to check whether this newly added code works by drawing a picture of the mask!*

Now that we have constructed the matrix `mask`, it's time to take out the high frequency content. This line of the script does that:

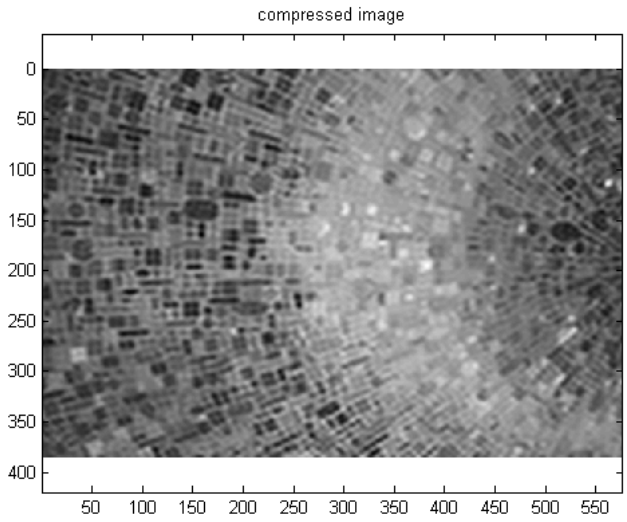
```
>> img_f_truncated = img_f.*mask;
```

Recall that the operation `'.*'` indicates entry-wise matrix multiplication. As such `img_f_truncated(i,j) = img_f(i,j)*mask(i,j)`. If `mask(i,j)=0`, then `img_f_truncated(i,j) = 0`, meaning that the weight of the wave component with row-frequency $i-1$ and column frequency $j-1$ has been changed to zero.

The final step of the image compression process is to take our truncated matrix `img_f_truncated` in the frequency domain and convert it back to the pixel domain using IDCT:

```
>> img_compr = idct2(img_f_truncated);
```

If everything went well, this is how the compressed image ought to look:



As you can see, the resulting image is not as sharp as the original image but the main features are still visible. However, the size of this image is greatly reduced.

Exercise 12 *Explore both rectangular and diagonal masks of different sizes in* `compress_image.m`. *What gets the most compression (the most zeros in the mask) without losing too much quality?*

Comments:

Exercise 13 *Pick an image of your choice, convert it to black and white, and try to compress it by deleting frequency information. (Watch out that* `m_trunc` *and* `n_trunc` *or* `num_diagonal_rows` *are within the size of the image, or else you will get an error). How good of compression can you get before the image starts to look bad?*

Comments:

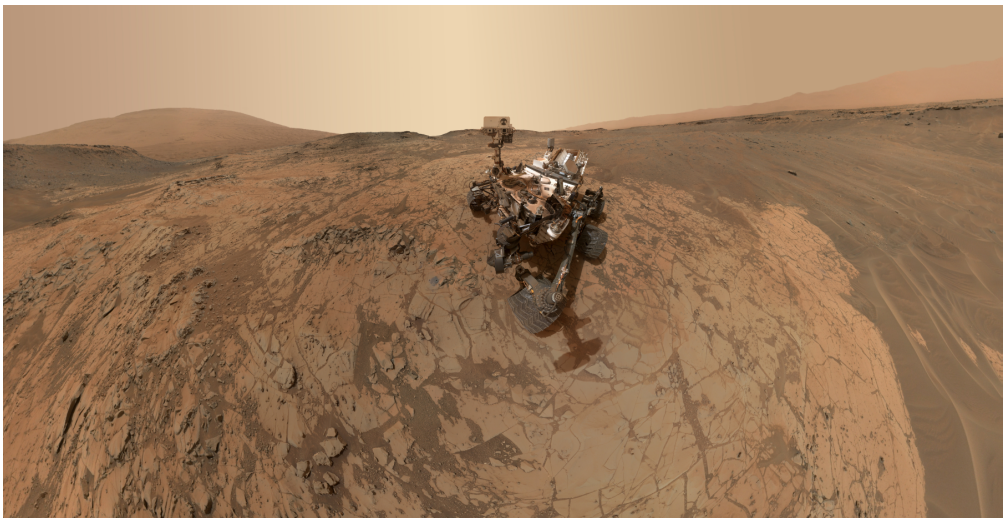
Checkpoint: get a TA to sign off for your answers in this section.

Exercise 14 *(Optional, but interesting!) What happens if instead of masking away high frequency information, you keep the high frequencies and mask out low frequencies?*

Answer:

4 Realistic Application of Color Image Compression

Now that you're familiar with image compression, let's address more realistic problems in which image compression is an indispensable tool. For this activity, we will consider the image below taken by Curiosity rover.



As usual, we will start by loading this image `'curiosity.jpg'` into Matlab and call it `curiosity`. Don't forget to convert the image to double. After doing this, you will notice that the image has large dimensions and that it takes Matlab quite some time to read

the image and to plot the image. If you apply `dct2` to each channel of the image you just read, there is a high chance that Matlab (and your computer) will hang up due to memory issues **so please do not attempt to do this**! This already suggests that for the purposes of our implementation, resizing the image would be the most practical option.

But before we proceed in that direction, let us try to make estimates of how long it would have taken Curiosity to transmit this image back to earth. To do this, we can look at the space that the variable `curiosity` occupies. If you type in

```
>> whos('curiosity')
```

on the command line, you will see that this image takes up 166461757 bytes. Yikes, that's a lot! Now according to the information provided by NASA on `mars.nasa.gov/mer/mission/comm_data.html`, the rover can send data direct-to-Earth at a rate between 3,500 and 12,000 bits per second . This process, however, is complicated by the fact that issues related to power and thermal limitations imply that rovers can only transmit data at most three hours/day. This brings up the question:

Exercise 15 *With the information provided, how long would it have taken Curiosity to send the above image back to Earth assuming that it did not make use of image compression? Recall that 1 byte is equivalent to 8 bits.*

Answer:

Hopefully, that calculation made the importance of image compression more tangible. At this point, you should create a new Matlab script in order to carry out the succeeding tasks. To process this image, we first have to resize it so we can perform calculations with it on Matlab. If were using very big computers we might not have to do this, but it could still be a good idea. As a reminder, the command `imresize` accomplishes this. Let's try a scale of 0.05—20 times smaller than the original image. How much less memory does it now take up?

The next step in this activity is to compress the image. This process should be familiar to you by now after the previous section.

Exercise 16 *Construct a rectangluar matrix* `mask` *such that you achieve a great degree of compression while simultaneously producing a compressed image that is nearly indistinguishable, upon first inspection, from the original resized image. Ideally, the closer the sharpness of the compressed image is to the original, uncompressed image, the better. Use a scale of 0.05 in rescaling the original image.*

5 Application: Image Debanding (Optional)

Here is a very different application of images in the frequency domain. This satellite image shows weird *striping* artifacts—a pattern of darker and lighter stripes across the image. How can we get rid of these artifacts?

Exercise 17 *Complete the Matlab script* `debanding.m` *to read in* `striping.png` *and remove the striping artifacts.* Hint: do you see anything weird in the frequency domain image?

6 The Takeaway

The main takeaway of this lab activity is that there are instances wherein working in a different domain might be much more convenient than working in the original domain. We saw this in the image compression exercises in which compressing an image would be hard to do in the pixel domain but simple to do in the frequency domain. This concept is not only limited to image processing applications but also presents itself in other scenarios such as trying to make noisy audio recordings more audible.