Lab 6: Frequency Domain Filtering

due TBD

Goals:

1. To gain an understanding of convolution filtering in the frequency domain.
2. To gain an understanding of Fourier Transforms and their application.

Introduction:

In this lab assignment, you will be guided through the steps for filtering a 2-D image in the *frequency* (or *spectral* or *Fourier Transform*) *domain*. The Fourier Transform (FT) provides, among other things, a powerful alternative to filtering in the *spatial domain*. It allows for both isolation and processing of particular *spatial frequency components* of an image, and, thus, it permits low-pass and high-pass filtering techniques with a great degree of precision.

The Fourier Transform technique relies on *periodic functions* to describe a signal. In its basic form, a 1-D Fourier Transform decomposes a 1-D signal (think of oscillations in a seismogram) into a (finite or infinite) summed set of sine and cosine waves. This summation is called a *Fourier series*. Fourier analysis deals with *complex numbers* in these summations. Recall that a complex number, z, is composed of a *real* part, x, and an *imaginary* part, y, such that: $z = x + iy$. Recall also that $i = sqrt(-1)$. Like numbers, functions can also have real and imaginary components. For example, a general, *complex-valued function f(x) of a real variable* (x) can be written as: $f(x) = u(x) + iw(x)$. The most important complex-valued function for our discussion of Fourier analysis is the *complex exponential function* (also known as *Euler's formula*): $e^{i\theta} = \cos\theta + i\sin\theta$. After a little algebra, you'll find that: $\cos\theta = (e^{i\theta} + e^{-i\theta})/2$ and $\sin\theta = (e^{i\theta} - e^{-i\theta})/2i$.

With this notation, the *1-D Fourier Transform* can be defined as:

$$\begin{aligned} F(k) &= \int_{-\infty}^{\infty} f(x)e^{-i2\pi kx}dx \qquad F(k) = \Im[f(x)] \quad \text{forward transform} \\ f(x) &= \int_{-\infty}^{\infty} F(k)e^{i2\pi kx}dk \qquad f(x) = \Im^{-1}[F(k)] \quad \text{inverse transform} \end{aligned} \qquad (1),$$

where x is distance and k is the *wave number*. Wave number is the reciprocal of the wavelength (λ), i.e. $k = 1/\lambda$. These equations are also commonly written in terms of time (t) and frequency (ν), where $\nu = 1/T$, and T is the *period*.

Note that for a real-valued function, the result of taking its Fourier Transform is a complex-valued function. Note also that, for our purposes in processing remote sensing imagery, we are talking about **spatial** wavelengths, frequencies, wave numbers, and periods as opposed to the spectral (i.e. electromagnetic spectrum) properties of the data!

For image processing, where we typically deal with digital numbers that are stored in 2-D matrices (images), there is an equivalent *2-D Fourier Transform*. Using this, an image is broken down into patterns that have a *sinusoidal* variation along each axis (e.g. row and column). We can display these patterns by taking the Fourier Transform of an image. This will yield a new matrix of the same size, but that contains information about the (spatial) *frequency content* of the image. 2-D Fourier Transforms simply involve a number of 1-D Fourier Transforms. More precisely, a 2-D Fourier Transform is achieved by first transforming each row, replacing each row with its transform, and then transforming each column, replacing each column with its transform. Thus a 2-D Fourier Transform of an *n* row by *n* column image requires a total of *2n* 1-D Fourier Transforms.

The *2-D Fourier Transform* is defined as:

$$F(\mathbf{k}) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(\mathbf{x}) e^{-i2\pi k(\mathbf{k}\bullet\mathbf{x})} d^2\mathbf{x} \qquad F(\mathbf{k}) = \Im_2[f(\mathbf{x})] \quad \text{forward transform}$$

$$f(\mathbf{x}) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(\mathbf{k}) e^{i2\pi k(\mathbf{k}\bullet\mathbf{x})} d^2\mathbf{k} \qquad f(\mathbf{x}) = \Im_2^{-1}[F(\mathbf{k})] \quad \text{inverse transform} \qquad (2),$$

where **x** = (*x*, *y*) is the *position vector* (analogous to distance in equations 1 above) and **k** = (k_x, k_y) is the *wave number vector* (analogous to the wave number in equation 1 above), such that **(k·x)** = $k_x\, x + k_y\, y$. These quantities are vectors because they have two components, one corresponding to the row direction and the other to the column direction.

The Fourier Transforms shown in Equations (1) and (2) transform a function from the *spatial* (or time) domain and put it into the *Fourier* domain (also called the *frequency domain*). There are also *Inverse Fourier Transforms*, in both the 1-D and 2-D cases. These transform a function from the *Fourier* domain back into the *spatial* domain.

The Fast Fourier Transform

While the Fourier Transform (and its inverse) can be applied to any complex-valued series or array of numbers, very large series or arrays may require considerable computational time, proportional to the **square** of the number of points (*n*) in the series, or n^2. For our purposes, think of *n* as the total number of pixels in an image.

A faster algorithm was developed by Cooley and Turkey (1965) and is appropriately called the *Fast Fourier Transform* (FFT). We will not get into the programming details of the FFT algorithm (http://en.wikipedia.org/wiki/Fast_Fourier_transform), but you should know that it is a remarkably efficient tool for solving large problems, and nearly every computational platform (including MATLAB) has a library of highly-optimized FFT routines. The computational time for the FFT is proportional to $n \log_2(n)$. For example, a process of total of n = 1024 points (pixels) using the regular Fourier Transform takes about 100 times longer than the FFT, a significant speed increase!

The only requirement of the FFT algorithm is that the number of points in the series must be a power of 2. So valid series sizes appropriate for an FFT are 2, 4, 12, 32, 64, 128, 256, 512, 1024, etc. Likewise, in order to perform a 2-D FFT on an image, the image must first be prepared so that the width and height (number of rows and columns) of the image matrix are both integer powers of 2. This can be achieved in one of two ways:

1) scale the image up to the nearest integer power of 2
2) *zero-pad* the image to the nearest integer power of 2.

The second option is typically chosen when working with MATLAB because MATLAB has a built-in function, **`fft2`**`(x,m,n)`, that takes an argument (see below) that allows you to specify the zero-padded size of your Fourier Transformed image (i.e., 256 x 256, 512 x 512, 1024 x 1024, etc.).

```
MATLAB's FFT2 Help:

FFT2 Two-dimensional discrete Fourier Transform.
   FFT2(X) returns the two-dimensional Fourier transform of matrix X.
   If X is a vector, the result will have the same orientation.

   FFT2(X,MROWS,NCOLS) pads matrix X with zeros to size MROWS-by-NCOLS
   before transforming.

   Class support for input X:
      float: double, single
```

The Fourier Convolution Theorem

One of the most powerful properties of the Fourier Transform is the *convolution theorem*. Suppose you wish to *convolve* an image f with a spatial filter g. The methods we have discussed so far for doing this in the spatial domain involve placing g over each pixel of f and calculating the product of all corresponding values of f and elements of g, and adding the results. This convolution is denoted as $f * g$, and this is essentially what we have been doing with the moving window kernels from the previous lab and lectures.

This method of convolution can be very slow, especially if g and f are large. The convolution theorem, however, states that the result $f * g$ can be obtained using Fourier Transforms through the following sequence of steps:

1) Pad f so that its dimensions (m x n) are a power of 2. Pad g with zeros so that it is the same size as f. Denote this padded result by g'.
2) Form the FFT of both f and g' to obtain $F(f)$ and $F(g')$. This takes the problem from the spatial domain and into the frequency domain.
3) Compute the element-by-element product of these two transforms: $F(f)\bullet F(g')$.
4) Take the Inverse Fourier Transform of the result: $F^{-1}[F(f)\bullet F(g')]$. This takes the solution to the problem back into the spatial domain.

In short, the convolution theorem states: $f * g = F^{-1}[F(f)\bullet F(g')]$. In simple terms, this means that:

***Convolution in the spatial domain is <u>equivalent to</u> multiplication the frequency domain*!**

Displaying the FFT in MATLAB

The *Fourier Transformed image* is essentially the real part of the complex-valued function you get by Fourier Transforming a real-valued function. The imaginary part is called the *phase* and is typically not displayed as an image (but it is required in order to reassemble the image in the spatial domain). For display purposes, it is convenient to rearrange the four quadrants returned by the 2-D Fourier Transform so that the *DC component* is in the middle of the Fourier Transformed image. The DC component is the component in the Fourier series of sines and cosines that has wave number (or frequency) = 0. For some reason, the default organization of the quadrants from most FFT routines (including that implemented in MATLAB) does not provide this arrangement. However, there is a MATLAB function, **`fftshift()`**, that you can use to shift the DC component to the center of the Fourier Transformed image plot. Figure 1 (below) illustrates how the Fourier Transformed image is displayed using **`fft2()`** and how it is displayed after shifting it using **`fftshift()`**. Essentially you reflect the quadrants along the diagonals (note that the black square is where the DC component is):

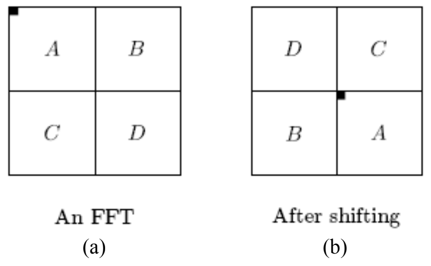


Figure 1

Some Relevant MATLAB Commands

Rather than make you write your own basic FFT routines, in this lab you will become familiar with the following built-in MATLAB commands:

`fft2`	which takes the FFT of a matrix;
`ifft2`	which takes the inverse FFT of a matrix; and
`fftshift`	which performs a shift as shown in Figure 1 (above).

Note that to <u>view</u> the result of an FFT (say one stored in MATLAB variable **`F`**) as the Fourier Transformed image, you must plot **`log(abs(F))`**. Filtering operations on the frequency content, though, must instead be performed on the real part, **`real(F)`**. (Recall from your previous math classes how the absolute value of a complex function is calculated and how that is different than the real part of that function). For operations such as filtering, you need to manipulate the real part of the Fourier Transformed image when working in the Fourier domain.

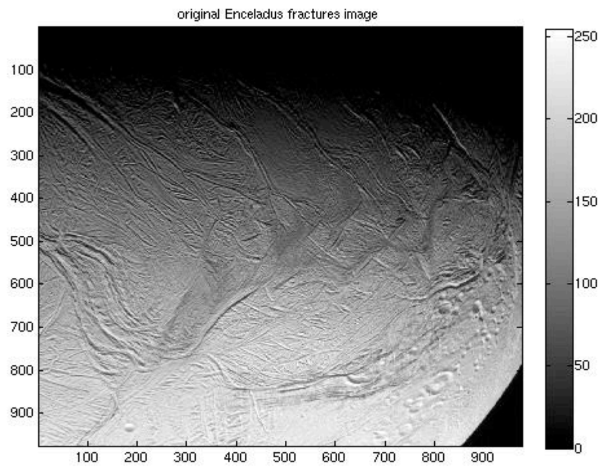
<u>Example Code</u>:

**The* `Enceladus_fractures.jpg` *image is located in the* `Assignments` *folder on* `//yowa` *if you'd like to try out this example. Output figures for the following code are appended at the end of this section.*

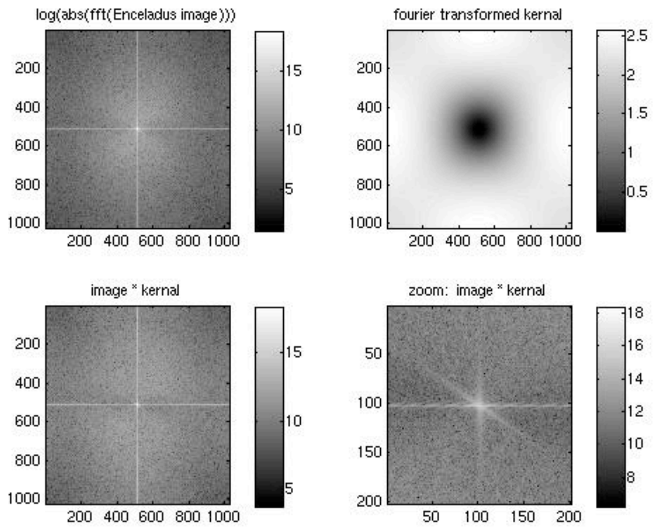
```
% matlab script to perform Fourier domain filtering

% load in Enceladus image

image = imread('Enceladus_fractures.jpg');
enceladus = double(rgb2gray(image));


% plot the given image

figure(1)
imagesc(enceladus)
title('original Enceladus fractures image')
colormap gray
colorbar


% get size of original image, will need these numbers later

[height,width]=size(enceladus)


% take Fourier Transform of the image
% but first zero pad image out to nearest power of 2
% possible tries:  1024  2048  4096

xdim_zeropad = 1024;
ydim_zeropad = 1024;
xhalf = xdim_zeropad/2;
yhalf = ydim_zeropad/2;


% Fourier Transform & shift the quadrants

Fenceladus = fft2(enceladus,xdim_zeropad,ydim_zeropad);
FSenceladus = fftshift(Fenceladus);


% display the log(abs) of the Fourier transformed image

figure(2)
subplot(2,2,1),imagesc(log(abs(FSenceladus)))
title('log(abs(fft(Enceladus image)))')
colormap gray
colorbar


% set pixval on or off and interactively zoom to find frequency spikes

pixval on
pixval off
```

```
% at this stage, you can either design a mask in the Fourier domain,
% or define one in the spatial domain and Fourier Transform
% to perform the convolution (multiply) in the Fourier domain.

%  create a 3x3 high pass filter mask
%  set up a matrix of zeros, and insert the 3x3 in the center

HP = zeros(ydim_zeropad, xdim_zeropad);
HP(yhalf-1:yhalf+1,xhalf-1:xhalf+1) = [-1 -1 -1; -1 9 -1; -1 -1 -1];


%  you can also set up a 3x3 low pass filter mask

LP = zeros(ydim_zeropad, xdim_zeropad);
LP(yhalf-1:yhalf+1,xhalf-1:xhalf+1) = [.25 .5 .25; .5 1 .5; .25 .5 .25]


% now Fourier Transform the high pass (HP) kernal,
% make it the size of the zero-padded enceladus image

Fkernal = fft2(HP,xdim_zeropad,ydim_zeropad);
FSkernal = fftshift(Fkernal);


% plot the Fourier Transformed kernal, display with log

subplot(2,2,2),imagesc(log(abs(FSkernal)))
title('fourier transformed kernal')
colormap gray
colorbar


% multiply the Fourier Transformed image and kernal

FS_HPenceladus = FSkernal.*FSenceladus;


% plot the result

subplot(2,2,3),imagesc(log(abs(FS_HPenceladus)))
title('image * kernal')
colormap gray
colorbar


% plot a zoom of the result

FS_HPenceladus_zoom      =      FS_HPenceladus(yhalf-100:yhalf+100,xhalf-
100:xhalf+100)
subplot(2,2,4),imagesc(log(abs(FS_HPenceladus_zoom)))
title('zoom:  image * kernal')
colormap gray
colorbar

% now unshift and inverse Fourier transform
% note*:  extra fftshift is needed here, but not always.
% You should experiment with ifft2(fftshift()) to see what this does.
```

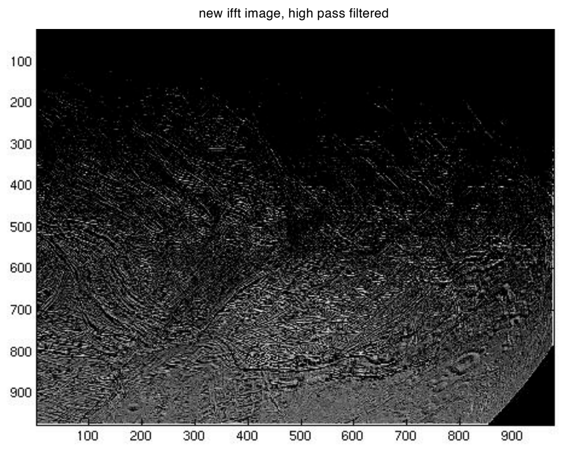
```
HP_enceladus_all = fftshift(ifft2(fftshift(FS_HPenceladus)));

% also reshape image to original size, take the upper left quadrant

HP_enceladus = HP_enceladus_all(1:height,1:width);


% Normalize grayscale between 0 and 255

junk = HP_enceladus - min(min(HP_enceladus));
enceladus_image_norm = junk/max(max(junk))*255;


% plot the final image, adjust color scale limits to your liking

figure(3)
cm=[120 200];                  % color map scale, adjust saturation
imagesc(enceladus_image_norm,cm)
colormap gray
title('new ifft image, high pass filtered(')

% you can also use the Low Pass (LP) filter provided on the following
% page to generate a low-pass filtered image
```



figure(1)



figure(2)



figure(3)

Instructions:

1. In the `Assignments` folder on `//yowa`, you will find an image titled `<yourlastname>.jpg`. There is one for each of you. Copy your image over to your working directory. Work on your own image first, but for fun you can work on others' images when you are done with your own.
2. Using the example code as a guide, open the image in MATLAB and display it in a figure window (`Figure1`). Save the figure window to a JPEG file, and name it `Figure1.jpg`.
3. You will notice that your image is contaminated with stationary, periodic noise. It is your job to remove the noise from the image using Fourier techniques. To begin, you will need to take the Fourier Transform of your image using the **`fft2()`** command. Pay attention to the method shown in the example code for properly shifting and zero-padding your image. Remember that you will need to properly zero-pad your image to some dimension that is a power of 2. **What two reasons require padding?**
4. Display the log of the absolute value of your transformed image in a new figure window (`Figure2`). Save this figure to a JPEG file, and name it `Figure2.jpg`.
5. Next you need to inspect your Fourier Transformed image for anomalous frequencies. Zoom in (several times; use the zoom tools in the figure window) to the near-center of your image and examine the locations of any anomalous "spikes", which will likely be gathered along several rows and columns. You can use the **`pixelval on`** command in MATLAB to interactively investigate the pixel values with your mouse. Make note of where (rows, columns) the spikes are located. Any spikes you see in the Fourier domain will likely be the result of periodic noise (such as the noise you saw in the spatial domain). Remember that the center of the Fourier Transformed image is the zero frequency (or DC) component. It should appear bright due to all of the non-periodic features in your image. Any true periodic signals, however, will show up as lines and clusters of points away from the center of the image. **Why is this the case?**
6. To remove the frequency spikes, you simply multiply your Fourier Transformed image by a *mask*. A mask is the Fourier domain equivalent of a convolution filter in the spatial domain. It should have *zero values* at the matrix index locations (e.g. row, column locations) where the spikes you want to remove are present, and *ones* everywhere else. Even better, the ones and zeros can be tapered using a windowing function. Using the locations of your identified frequency spikes, generate a mask of zeros and ones (extra credit if you create a mask with a Gaussian or other windowing function). Because you will be multiplying your image by this mask (in a pixel-by-pixel sense), the mask will need to be the exact same size (same total number of rows and columns) as your Fourier Transformed image. Remember that the mask will primarily contain ones. The exceptions are at the locations of the spikes that you identified. At these locations, you'll want your mask to contain zeros.
7. In a new figure window (`Figure3`), display zoomed-in versions of both your Fourier Transformed image and your mask. To do this, use the **`subplot`** command and zoom to a region approximately 200 pixels (horizontal and vertical) from the center (in both images). To zoom using the **`imagesc`** command, you can enter the range of pixels you want to display. For example: **`imagesc(matrix(400:600,400:600))`**. Zoom into an interesting and relevant part of your image to make this figure. Save the figure window to a JPEG file, and name it `Figure3.jpg`.

8. Next multiply your Fourier Transformed image (the whole thing) by the mask (the whole thing) and save the result into a new variable. Note that when you do this, you will be removing the anomalous frequency spikes (setting them to zero, the background value) by the action of multiplying them by zero. The rest of your image will remain untouched, as you will multiply these pixels by one. Don't forget to use `.*` to perform an element-by-element multiplication – you don't want matrix multiplication here! **What property of the Fourier Transform are you applying?**

9. Display the result of this multiplication in `Figure4`. Save the figure window to a JPEG file, and name it `Figure4.jpg`. Are your spikes mostly gone? If not, you may need to re-evaluate the spikes in the original Fourier Transformed image and try again. If you think you did a pretty good job at removing the spikes, then move on to the next step. You will find out whether you truly removed the offending frequencies after you have transformed your image back to the spatial domain!

10. Your next step is to Inverse Fourier Transform your mask-multiplied, Fourier Transformed image using the **`ifft2()`** command. Follow the steps shown in the example code to do this. Don't forget that you first need to shift back, undoing what you did in step 3. Also remember that after the Inverse Fourier Transform you must re-size the spatial-domain image to its original size, effectively un-doing the zero-padding that you did in step 3. *Note the symmetry of this entire process (e.g. step 3 and step 10)!*

11. Display your resulting image in a new figure window (`Figure5`). How does your image look? Do you recognize anything? If your image still contains distinct stripes or checkered noise, then you will need to go back and inspect the frequencies in the Fourier domain. However if you image looks pretty good, then you are almost done!

12. Clean up. You should normalize your image to the 8-bit range (0-255). Follow the steps in the example code to do this. Your image will also likely look a little dull (low contrast). Apply a contrast stretch to best enhance your image. Update Figure 5 with your enhancements, save the window to a JPEG file, and name it `Figure5.jpg`.

<u>What To Turn In:</u>

For this lab, please turn in a `.m` file MATLAB program that does the analysis in steps 1-12. Also turn in the JPEGs for Figures 1-5 you are asked to produce as described above. Be sure to adequately comment your code so that I know what your program does, how it works, and even perhaps why it works. Note the questions in boldface – answer these in the comments of your code at the appropriate places. Turn in your code and figures via the `DROPBOX` on `//yowa`.