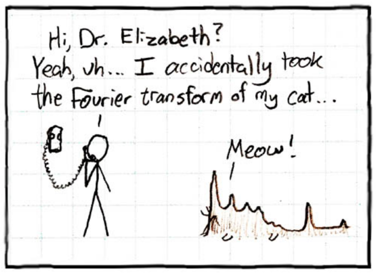
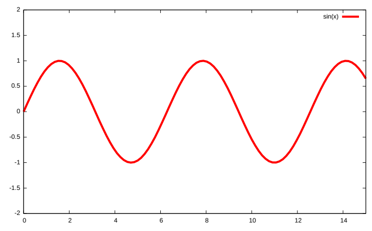
CS1114 Section 8: The Fourier Transform

March 13th, 2013

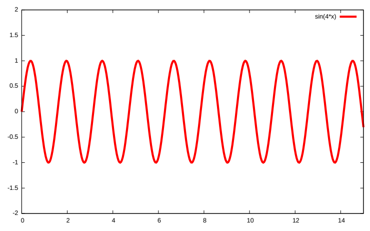

Hi, Dr. Elizabeth?
Yeah, uh... I accidentally took the Fourier transform of my cat...
Meow!

http://xkcd.com/26

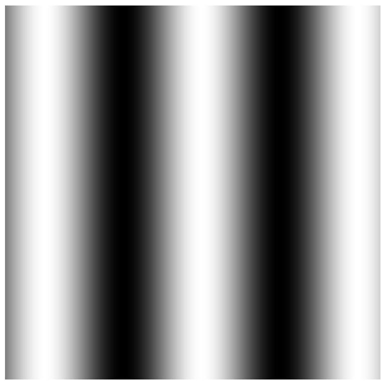
Today you will learn about an extremely useful tool in image processing called the **Fourier transform**, and along the way get more practice with manipulating matrices in Matlab. In class, we've learned about transformations that take one image to another image. The Fourier transform takes in an image (or, in 1D, a signal), and maps it to a different kind of space—*frequency* space. What are frequencies? Well, a frequency is essentially how fast a signal varies spatially across the domain. The figure below shows two sine waves of two different frequencies, one set in 1D and one set in 2D:



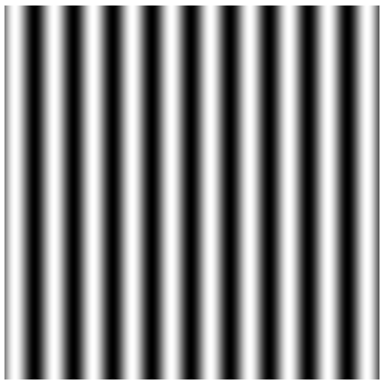
(a) Lower-frequency signal (1D)



(b) Higher-frequency signal (1D)



(a) Lower-frequency signal (2D)



(b) Higher-frequency signal (2D)

The *frequency* of a sine function is just the inverse of its period. A low-frequency function, like the ones on the left of the figure above above, oscillates very slowly, whereas a higher frequency function (like the ones on the right), oscillates much more quickly, with lots of variation in the same amount of space. For instance, the sines waves on the left of the figure might represent $\sin(2x)$, while the right signals might represent $\sin(8x)$. We say a sine wave $\sin(kx)$ (or a cosine wave $\cos(kx)$) has frequency k.

1 1D Fourier Transforms

Why are these sine waves of different frequencies useful? It turns out that we can represent any signal (or image) as a sum of sine and cosine functions with different frequencies, weighted by different coefficients. You might remember from calculus that you can take any function $f(x)$ and approximate it around some center point with a power series $f(x) \approx a_0 + a_1 x + a_2 x^2 + a_3 x^4 + \ldots$ (this is called a Taylor series). For instance:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

You can think of computing the Taylor series of a function as computing a "Taylor transform" of the function—transforming the function into its coefficients in Taylor space. We can represent the function in this new space as an array of the coefficients of the Taylor series; in other words, the sin function, under this "Taylor transform" becomes an (infinite) array of numbers:

```
[ 0, 1, 0, -3, 0, 5, 0, -7, ... ]
```

(In practice, we often approximate the function by only keeping the first few numbers in the series.)
Along the same lines, Joseph Fourier came up with the idea of representing an arbitrary function as an infinite sum of weighted sine and cosine functions with different frequencies:

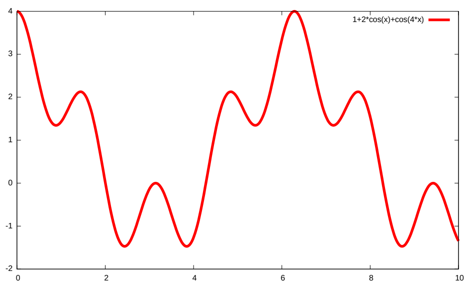
$$f(x) = a_0 + a_1 \cos(x) + b_1 \sin(x) + a_2 \cos(2x) + b_2 \sin(2x) + \ldots$$

or, in other words:

$$f(x) = a_0 + \sum_{k=1}^{\infty} \left[a_k \cos(kx) + b_k \sin(kx) \right].$$

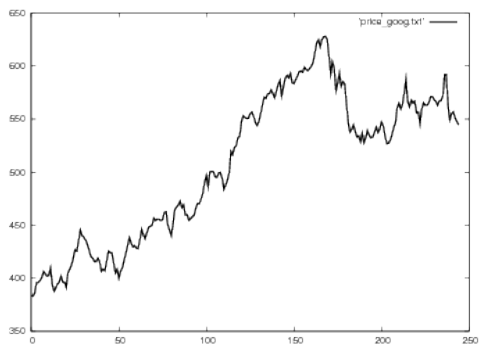
This summation is called the *Fourier series* of f. The sequence of (complex) numbers $a_0, a_1 + ib_1, a_2 + ib_2$ is called the (discrete) Fourier transform of f. We won't worry about the fact that these are complex, not real numbers for now. The important thing is that the Fourier transform takes one function and transforms it into another representation, this new set of Fourier coefficients.

Here's a simple example. Suppose our function is simply $f(x) = 1 + 2\cos(x) + \cos(4x)$. This function looks like this:

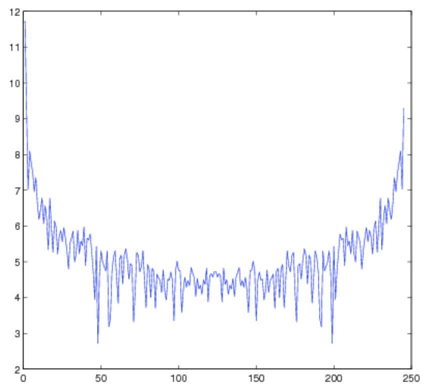


and the (discrete) Fourier transform of f is just $1, 2, 0, 0, 1, 0, 0, \ldots$.

The Fourier transform of a 1D function (i.e., a signal) tells us something about the *frequency content* of that signal. If only the first few coefficients are large, then that means the signal is fairly low-frequency—it doesn't oscillate very rapidly. On the other hand, if there are coefficients far along in the series that are large, then that means that the signal changes very rapidly, at least in some places. Here's an example of a signal somewhere in between:



This is a plot of the opening stock price of Google (ticker symbol GOOG on NASDAQ). We can compute the Fourier transform of this signal, and plot it as another signal. In Matlab, the function `fft` computes the Fourier transform of a signal (FFT stands for Fast Fourier Transform, incidentally). Here's what the Fourier transform of the Google opening price signal looks like:



Actually, this Fourier transform is complex (i.e., there is an imaginary component), so this plot shows the magnitude of the complex numbers (we won't worry too much about that, though—we have to save something for other classes). Also, this plot shows the *logarithm* of the magnitudes, as the difference between different Fourier coefficients can be large. As you can see, this function has large coefficients for low frequencies (numbers on the x-axis close to zero), relatively low coefficients for medium frequencies (x between around 50 and 200), and larger coefficients again for the highest frequencies ($x > 200$). This makes some sense if you look at the original function—over a long period of time, the overall shape of the stock price is relatively smooth (low frequency), but on very short time scales—a single day—the function can jump quite a bit (high frequency). So it is a mixture of low frequency and high frequency behavior, with some lesser amount of medium frequencies mixed in. Because this plot shows the log of the Fourier transform, the largest coefficients are actually many times bigger than the smallest coefficients.

Now it's your turn to try this out. Copy the files from `/courses/cs1114/section/fourier/` and open up Matlab. Load the data from the file `price_goog.txt` by typing:

```
>> goog = load('price_goog.txt');
```

This stores the data as an array called `goog`. You can then do things like `plot(goog)` to see a plot of the prices. To compute the Fourier transform, we do:

```
>> goog_ft = fft(goog);
>> plot(log(abs(goog_ft)));
```

Fourier transforms contain complex numbers and often have huge ranges between the smallest and largest elements. In order to display these values usefully, we use `abs` to compute the (real-valued) magnitude of these numbers, then `log` to compute the logarithm. Note that like most math functions in Matlab, when you call `abs` or `log` on an array or matrix, the operation is applied individually (element-wise) to each element of the array or matrix.

Now try plotting the Fourier transform of Apple's historical stock data, which is in `price_aapl.txt`. Remember how to use the `hold on` command to put two plots on top of each other (`hold off` turns this off), and plot the Fourier transform of Google's and Apple's stock data together. Do they look similar?

2 2D Fourier Transforms

We can also take the Fourier transform of a 2D signal, i.e., an image. Just as the Fourier transform of a 1D signal gives a set of numbers that we can think of as another signal, the Fourier transform of a 2D image gives us a 2D array that we can also think of as an "image" (although it will look nothing like the original image). In Matlab, we do this with the `fft2` function.

Let's try this out. Read in the image called `llama.png`. This image looks like a llama:

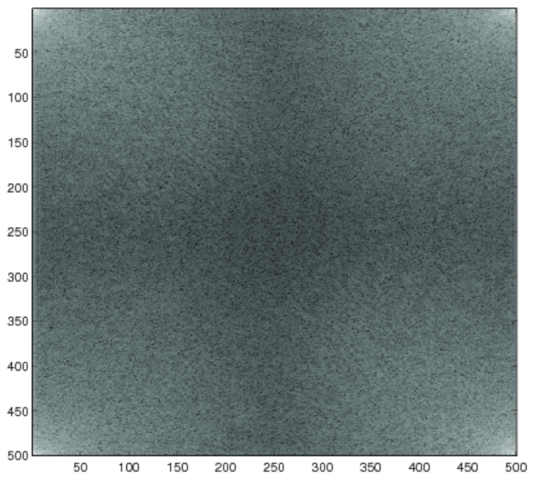
You'll want to convert this (and all other images in this lab) to a double image:

```
>> llama = im2double(imread('llama.png'));
```

If you do an `imshow(llama)`, you will see a llama. Now take the Fourier transform of this image, using the `fft2` function, storing the result in `llama_ft`. Use `imagesc` to display `llama_ft`.

Whoops! If something went wrong, you might have forgotten to put in the `abs` function, as Matlab doesn't like displaying complex-valued images. Also, if you forget to put in the call to `log`, you'll probably get an image that looks almost completely black, as some coefficients are *much* larger than others. You may also want to tell `imagesc` to display values using a gray color map (instead of the false-color "jet" theme) using the `colormap gray` command.

You should get an image that looks like this:



This image looks nothing like the original llama. It looks a lot like noise, except that the corners look somewhat brighter than the middle. In this case, the corners actually represent the low-frequency image content, and the middle represents higher frequency content. Amazingly, though this looks nothing like the original image, *all* of the original image data can be recovered from the Fourier transform. This can be done with the *inverse* Fourier transform, which in Matlab can be done with the `ifft2` function. This takes a Fourier-transformed images, and reconstructs the original image. To see this, try:

```
>> llama2 = ifft2(llama_ft);
```

If you view this image, the llama has reappeared!

Let's look at the Fourier transform of another image. Read in the image `ceiling.png`, and take a look at it. You'll notice that it has more repeated patterns than the llama image. Now compute the Fourier transform, and take a look at that as well. You should see that there is more evident structure in this Fourier transform than in the llama one. That has to do with the quasi-periodicity in the ceiling image.

Now this is where things start to get interesting. The neat thing about Fourier transforms is that certain operations which are complicated in image space become extremely simple in Fourier space. Let's see a few examples.

2.1 Blurring

One way to think of blurring an image is that we want to remove all of the high-frequency content, leaving only the smooth low frequency content. Earlier, we did this by convolving the original image with a box (or average) filter. But we can also do this very easily to the Fourier-transformed image, by simply setting all of the high-frequency coefficients to zero. Let's try this with the llama image. It turns out that this image is a 500×500 pixel, and the Fourier transform is also 500×500 (the size of the Fourier transform will always be the same as the original image, for now). Remember that the *corners* of the Fourier transform represent the low-frequency coefficients—let's set everything else to zero, everything except for, say, a 50×50 region around each corner. To do that, let's create an image called blocker, which is 0 everywhere except for around the four corners:

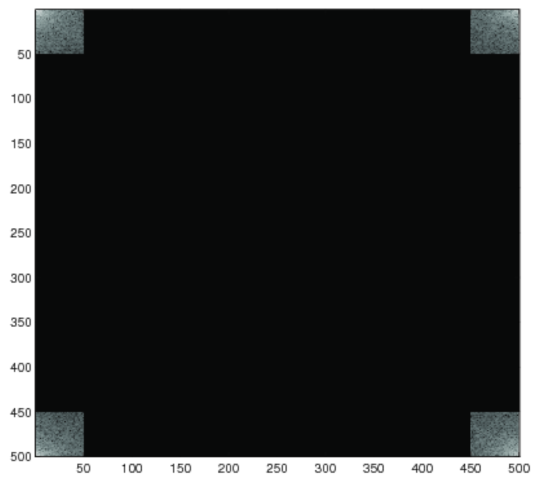
```
>> blocker = zeros(500,500);
>> blocker(1:50,1:50) = 1;
>> blocker(1:50,end-50:end) = 1;
...
```

If you haven't seen this before, `end` is a special name in Matlab that refers to the index of the last column (or row) of the matrix being indexed into—this can be very handy. Try out the above—you'll need a couple more lines to finish up the matrix `blocker`.

Now we can just multiply this matrix by the Fourier-transformed llama image to get a new Fourier transform:

```
>> llama_ft2 = llama_ft .* blocker;
```

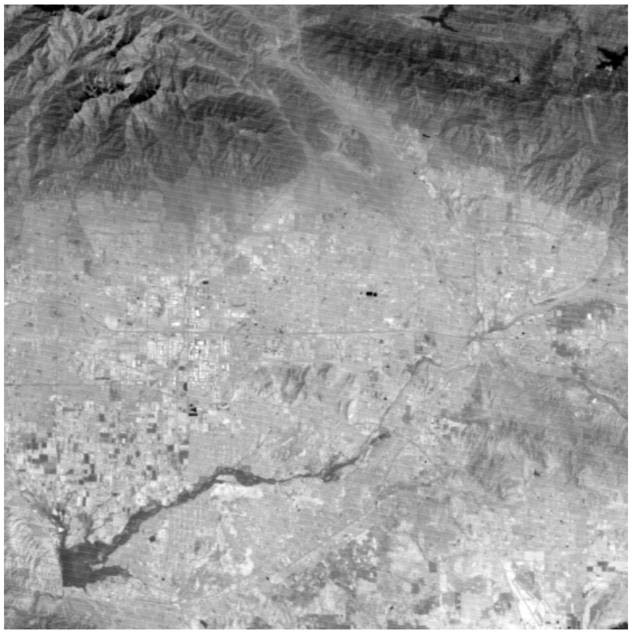
Note that we use `.*` instead of the regular matrix multiply operator (`*`). The reason is that we want to just multiply each element of `llama_ft` with the corresponding element of `blocker`, and not do a full matrix multiplication (this operation is called the entrywise product or Hadamard product, and is much simpler than the regular matrix multiply). `llama_ft2` should look like this:



What happens when we take the *inverse* Fourier transform of this new image? Try it out! Note that we may have introduced some small imaginary terms in the reconstructed image, so you may need to add in a call to `abs`. You should see a blurred version of the llama. You might also see some strange artifacts around edges in the image. These are called ringing artifacts, and they occur because simply setting a bunch of frequencies to 0 isn't actually the best thing to do. Feel free to try and figure out a better blurring operator in Fourier space.

2.2 Removing artifacts

Now for an even cooler application. Consider the following image, taken from a satellite:



There are a bunch of *striping* artifacts that are a result of the special high-resolution cameras used in satellites. How in the world can we remove the striping artifacts? It seems like a difficult task when we look at the image itself. However, let's take the Fourier transform of this image (you'll find the original image in `striping.png`). Do you notice anything interesting about the Fourier-transformed image?

You should notice some small bright lines that look a bit out of place. It turns out that these are related to the striping artifacts in the image. What would happen if we set just those frequencies to zero? To do this, we'll create another blocker image, that just sets a few entries of the Fourier-transformed image to zero. In fact, we've already created that image for you. Load the image `blocker.png` (remembering to do `im2double`), and have a look at it. You'll see that it's mostly ones, except for a few zeros, which are in key locations (which ones?). Try multiplying this blocker image with the Fourier transform, then do an inverse Fourier transform. If everything went according to plan, you should see a big difference...

2.3 If you have time...

We've seen how to easily blur an image and how to remove periodic artifacts in the Fourier domain. Here are a couple of things to think about if you still have time:

1. When we talked about convolution, we had blur kernels and also sharpening kernels that accentuated edges. Given that edges are by definition high-frequency, can you think of a way to do sharpening in the Fourier domain? Try it out on the llama image.

2. See if you can figure out how to do more even blurring without ringing artifacts that we saw with our corner blocker