B490/B659 Project 2: The frequency domain

Spring 2015
Due: Tuesday Feb 24, 2015, 11:59PM
(You may submit up to 48 hours late for a 10% penalty.)

Processing images in frequency space is very powerful, and lets us easily perform image operations that might otherwise seem difficult or impossible. The trade-off, however, is that thinking about what frequency space means or how to visualize it can be difficult, especially at first. This project will guide you through exploring the frequency space and understanding some of its power.

We've once again assigned you to a team with another student in your section, using the preferences you gave us in your Project 1 team feedback. We tried to accommodate as many preferences as possible. Please understand if we weren't able to accommodate yours, and you'll be able to change these teams in future assignments. You can once again find your assigned teammate(s) by logging into the IU Github website and looking for a new repository called *userid1-userid2*-p2. Please contact them ASAP, and let us know if there is a problem (i.e., they have decided to drop the class).

We once again recommend using C/C++ for this assignment, and we have prepared sketon code that will help get you started. The skeleton code is once again based on the simple but powerful CImage class. For this project, you *may* use the image processing methods of the CImg class, instead of having to write everything from scratch. You may also use additional libraries for routines not related to image processing (e.g. data structures, sorting algorithms, etc.). Please ask if you have questions about this policy.

Please ask questions on the OnCourse Forum so that others can benefit from the answers.

Part 1: Getting started

Clone the github repository with the skeleton code and test images:

`git clone https://github.iu.edu/cs-b490-b659/`*your-repo-name*`-p2`

where *your-repo-name* is the one you found on the GitHub website. Make the sample program on a Linux machine using the `make` command. The top of the skeleton code file contains some documentation on how to run it. It doesn't do very much at the moment, but hopefully is a good starting point for your work.

Part 2: Morphing images

In class we saw examples of "morphing images" which look strikingly different at different viewing distances, like the one in Figure 1. Here's the general recipe for creating one of your own. Start with two images, apply a low-pass filter to one (which reduces its high-frequency components) and apply a high-pass filter to the other (which reduces its low-frequency components), and then add the two together. A low-pass filter can be implemented through convolution with a Gaussian, while a high-pass filter can be implemented by simply subtracting the result of a low-pass filter from the original image. You can either use your own code from Project 1 or the convolution functions built in to CImg.

Implement a function that will create "morphing" images. Produce a cool hybrid image using photos of your own choosing to show in your report. Note that you'll have to experiment with the values for the sigmas of the low and high pass filters and different images before you find the perfect one to show.

Part 3: Fourier domain filtering

In class we discussed the discrete Fourier transform (DFT), which converts images into a frequency representation. Our skeleton code includes a function that computes the DFT for you. Given an image, it returns two CImg buffers; this is because the DFT of a real-valued signal includes complex numbers, so the code returns the real portion in one buffer and the imaginary portion in the other. We've also given you code for

Figure 1: This "morphing image" shows a tiger, but step back (or zoom out) and it morphs into a cheetah.

the inverse DFT (IDFT), which takes a real and imaginary buffer pair and converts back to a single image.

1. To understand the DFT better, it's useful to look at visualizations, but this is difficult to do directly because of the complex numbers. One approach is to compute the *spectrogram* of an image, which visualizes the energy at each frequency. To create a spectrogram, simply convert an image I to the frequency domain, creating a real matrix R and an imaginary matrix J, and then calculate the logarithm of the magnitude at each frequency like this:

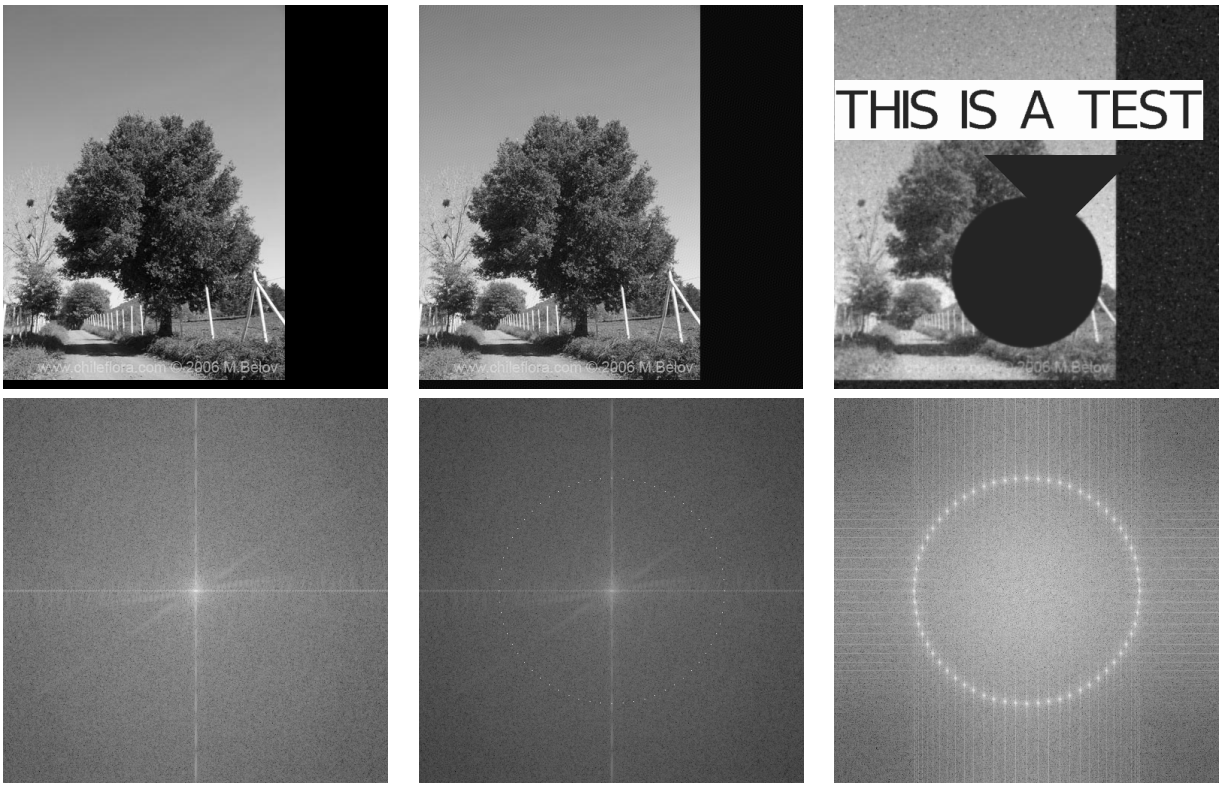
   $$E(i,j) = \log(\sqrt{R(i,j)^2 + I(i,j)^2})$$

   You can then save the resulting image buffer to an image file and view it as if it were an image. Try looking at the spectrogram for several images, including ones with very distinctive features (e.g. many horizontal lines or many vertical lines) – you should start to see patterns. Chapter 8 of the second volume of the Burger and Burge book (see OnCourse wiki) has examples of spectrograms and gives intuition about them. (It also has describes the math behind the Fourier transform, but you do not have to understand the math for this course.)

2. In Project 1 we saw two kinds of noise, additive Gaussian noise and salt-and-pepper noise. In your git repo, `noise1.png` has suffered another very common form of noise – interference. It may look hopeless to fix this image, but it turns out to be easy in the frequency space. Look at the spectrogram of this image – the problem(s) should be very obvious. Write a function that removes the interference, by changing the image in the frequency domain and then using the IDFT to produce a cleaned image. (Your code can be hard-coded for this particular image – it doesn't have to work for the general case.)

3. In class we mentioned a very important result called the Convolution Theorem. Intuitively, this theorem says that you can calculate the convolution between two images by taking the DFT of each of the two images separately, multiplying the resulting two (complex!) matrices together element-by-element, and taking the inverse DFT of the result. So the convolution operation you implemented in Project 1, which involved the complicated processing of moving a little kernel around the image and computing many dot products, is extremely easy to compute once the image is in frequency space. Write a function that convolves an image with a Gaussian filter using the DFT. Your result should look nearly the same as the code you ran in Project 1. How does the Gaussian filtering affect the spectograms?

4. Of course, we already wrote code for convolution in Project 1, so why do we need a new implementation? One reason is that when kernels become large, DFT-based convolution is much more efficient (we'll use this in later projects for object recognition). Another reason is that the Convolution Theorem has an important corollary: under some set of assumptions, convolution is an *invertible* operation. This means that if you have a blurry image, you can *perfectly* restore the original image! (...assuming you know the kernel that produced the blur, and some other conditions that are beyond our scope.) For



(a) (b) (c)

Figure 2: Example of watermarking process. An image (a) is injected with a watermark, creating an image (b) that looks nearly identical but has a clearly identifiable pattern on its spectrogram. Even after extensive editing of the image (c), the watermark is still visible in the spectrogram and can be detected. (Note: this is a visualization where we have exaggerated the spectrograms so that the watermark shows up clearly in the PDF; the real watermark is much more subtle.)

example, let's say there's an image I has been blurred with kernel K, so you only have blurry image, $J = I * K$. To infer I, you apply the convolution theorem "in reverse:" take the DFT of I and K, *divide* the former by the latter (using complex division), and then take the IDFT of the result.

Write code that, given an image and the sigma parameter of a Gaussian blur kernel, applies the blur to the image, and then uses this "deconvolution" to produce the original image again. Try it on some images – how well does it work? What happens when you try this on a box mean filter?

Part 4: Watermarking

The frequency domain has many other applications; for example, almost all modern image and video compression algorithms (e.g. MPEG, JPEG, etc.) encode content in frequency space. Another application is *watermarking*, where the idea is to mark an image with data (e.g. the name of the photographer or copyright information) that is nearly invisible to a human viewer. Of course, there are simple ways of watermarking an image – drawing some text on the image, for example – but these can be easily removed with a little photoshopping. Here we explore an idea that is much more robust – to embed the watermark in the frequency domain. We'll consider a relatively simple version here, but the best techniques inject watermarks that are nearly impossible to remove, even if the photo is heavily edited or even repeatedly printed in hardcopy and scanned again multiple times! Even our simple version will be remarkably robust – see Figure 2.

To simplify, assume the watermark we want to inject is an integer N (which might represent an ID number or cryptographic key), and later we'll want to test whether an image has been marked with this integer.

1. The first step is to produce a binary vector $v = (v_1, v_2, ..., v_l)$ (with $v_i \in \{0, 1\}$ and the length l a parameter of the watermarking algorithm) that appears to be a random sequence, but can be reproduced later on for any N. For instance, we can seed a random number generator with N, and then use the random number generator to produce the l binary digits (e.g. using `srandom` and `random` in C).
2. Given an image I to watermark, compute the fourier transform, which produces the real part R and imaginary part J.
3. We'll inject the watermark into some special frequencies of the image. Here is one way, by modifying the real portion of the fourier transform in a circular pattern of radius r (where r is another parameter, and controls which frequencies are modified). For instance, you could choose l evenly-spaced bins along the circle and modify each of those with one bit of the sequence. To inject bit v_i into bin $R(x, y)$, set its new value to:
$$R'(x, y) = R(x, y) + \alpha |R(x, y)| v_i,$$
where α is a constant (another parameter!).

   Hint: An important property of the fourier transform's real plane is that it is symmetric about the origin (which is at the center point of the real and imaginary buffers). That's why we recommend using a circle centered about the middle of the image. When you make a modification to bin that is some angle θ along the circle of radius r, you have to make the same change to the bin at angle $\theta + \pi$.
4. Take the modified real part R' and original imaginary part J, and create a new image I' with the IDFT.

To test whether a given image has been marked with a given number N, we:

1. Compute the same binary vector as before, using N.
2. Perform the DFT on the image.
3. Extract the values of the real components at the same locations as in step 3 above, giving a vector c.
4. If the correlation coefficient between vectors v and c is above a threshold t (another parameter!), declare the watermark to be present.

Implement the above algorithm for adding and extracting watermarks. You'll likely have to make some design decisions along the way, which will involve trial and error. For example, you'll need to select the parameters (t, r, l, and α) that give a good result; if set too high or too low, you'll either not be able to detect the watermark, or you'll detect the watermark when it's not there, or the watermark will create noticeable distortion in the image. Try to find a good compromise between these values. In your report, evaluate your approach both qualitatively and quantitatively. For qualitative results, show a couple of images where you embedded a watermark and tested whether the watermark survived. For the quantitative results, you might test whether your algorithm can find the correct watermark, but then also test whether it detects any of (say) 100 randomly-chosen watermarks also, since these would count as false positives.

Part 5: Deconvolution (Required for CS B659, Optional for B490)

This is a follow-up for Part 3. On OnCourse we've posted an executable program called which takes an image and produces a new image with simulated motion blur (i.e. the camera moving while the photo was taken). (This program will only run on the CS Linux machines.) The program does this by convolving with a kernel. Use your work in Step 4 to write code that undoes the motion blur. What happens if you add a bit of Gaussian noise to the image after running `noisify` but before running deconvolution?

What to turn in

Make sure to prepare (1) your source code, and (2) a report that explains how your code works, including any problems you faced, and any assumptions, simplfications, and design decisions you made, and answers the questions posed above. To submit, simply put the finished version (of the code and the report) on GitHub (remember to `add`, `commit`, `push`) — we'll grade the version that's there at 11:59PM on the due date.

Important hints and warnings

Design decisions. You'll likely encounter some decisions as you write your programs that are not specified in this assignment. Feel free to make some reasonable assumptions in these cases, or to ask on the OnCourse forum for clarification.

Academic integrity. You and your partner may discuss the assignment with other people at a high level, e.g. discussing general strategies to solve the problem, talking about C/C++ syntax and features, etc. You may also consult printed and/or online references, including books, tutorials, etc., but you must cite these materials in the documentation of your source code. However, the code that you (and your partner, if working in a group) submit must be your own work, which you personally designed and wrote. You may not share written code with any other students except your own partner, nor may you possess code written by another student who is not your partner, either in whole or in part, regardless of format.