Ken Steiglitz
ken@cs.princeton.edu

Image Processing and Fourier Transforms

Digital image processing is an important example of applied numerical computing. There are at least the following three reasons why people want to compute with the numbers in an image:

1. Enhancement: improving or changing a picture in some way.
2. Compression: reducing the storage required, usually to economize on storage or speed up transmission.
3. Recognition: automatically recognize objects, like faces or missile installations. Clearly, this application is in its early stages.

This assignment will have you computing with the elements (pixels) of digital images. Fortunately, a lot of the messy detail in reading images in, getting hold of the individual pixels, and writing images out can be circumvented by using a few functions that have been (most kindly!) provided by Prof. Adam Finkelstein (af@cs.princeton.edu).

Mechanics

You can download simple C code for reading and writing images in PPM "raw" format from links provided on the course assignment schedule (and also in the main "links" page). The Makefile there illustrates how to compile and link functions to your own program. The `struct Image` for an image is defined in `ppm.h`. For reference, here are the functions:

```
Image *ImageCreate(int width, int height);
Image *ImageRead(char *filename);
void   ImageWrite(Image *image, char *filename);
int    ImageWidth(Image *image);
int    ImageHeight(Image *image);
void   ImageClear(Image *image, u_char red, u_char green, u_char blue);
void   ImageSetPixel(Image *image, int x, int y, int chan, u_char val);
u_char ImageGetPixel(Image *image, int x, int y, int chan);
```

Each, respectively, creates an image, reads an image, writes an image, returns the width of an image, returns the height of an image, clears an image, sets the value of a pixel, and gets the value of a pixel.

Picture storage

I'll give a quick rundown on how color images are stored on a computer in the ppm raw format we'll be using. Each image is broked down into small "picture elements," called *pixels*. Each pixel in each image is represented by a fixed number of bits. For color images (which we'll be using exclusively) each pixel has three parts, representing the red, green, and blue components, respectively. These are sometimes called the red, green, and blue *channels*, and

are coded by channel 0, 1, and 2. Each color component of each pixel is represented by one byte (8 bits), and can be stored in C as type `u_char`. Thus, each pixel takes up 24 bits of storage, and this is sometimes called "24-bit color." In advertizing lingo, "more than sixteen million colors are possible!" Each byte represents an integer between 0 and 255; 0 means none of that color is present, 255 means that as much of that color as possible is present. Therefore, a pixel with values $\{0, 0, 0\}$ is black; a pixel with values $\{255, 255, 255\}$ is white; a pixel with values $\{255, 0, 0\}$ is red, and so on.

You'll find out very soon, if you don't know already, that pictures take lots of room. A little arithmetic: a relatively small, 24-bit-color image of 256x256 pixels takes 256*256*3 = 196608 bytes, or about 197 Kbytes. If you read that in and store each pixel color value in three different floats because you want to do arithmetic, that multiplies the storage by 4, because a float in C (usually) takes 4 bytes. So it's very easy to get up to a Mbyte for a small picture, before you even do anything.

The moral is clear. If you want to stay within your disk quota, you're going to have to avoid wasting space. In particular, you can't afford to make two copies of an image when one will do. Usually, you can process one color at a time in the same place. Enough said.

Using the Fourier Transform and the FFT

Frequency domain ideas are important in many application areas, including, but not restricted to, audio and image processing. For example, spectrum analysis is widely used to analyze speech, compress images, and search for periodicities in a wide variety of data in economics, biology, physics, and so forth. The class of JPEG compression algorithms, which are widely used and very effective, use a version of the Fast Fourier Transform called the Discrete Cosine Transform (DCT). See [1], for example, for lots of details about JPEG and the DCT. Almost all practical applications of spectrum analysis use some version of the FFT, an algorithm that takes the Fourier transform of an array of n points in $O(n \log n)$ time. In fact, use of the FFT is so widespread that it is almost synonymous with the Fourier transform, but technically the FFT is an *algorithm* and the Fourier transform is the *mathematical transformation* that it computes.

Wavelets are in a sense a generalization of the idea of the Fourier transform and have similar applications. For much more about wavelets, see the COS 598D home page (spring 1997).

One of the goals of this assignment is to get acquainted with the inner workings of the FFT, and its limitations and pitfalls. Be warned that there are unobvious traps for the unwary, and, as with many numerical algorithms, misleading results can be obtained by blind application of packaged programs. See [2] for an introductory level description of what the FFT does and how it works.

C code for the FFT is provided in the same place as the code for image I/O, in the form of a small main test program and the FFT function. The test is an 8-point transform of a complex sinusoid. After you compile and run the test, you should be able to turn off the printing and use the FFT for images.

As we'll describe in class, the Fourier transform of a two-dimensional array like an image can be performed by first transforming each row, and then transforming each column of the resultant array.

Assignment: Basic Part

There's lots to do here, and some of you will get farther than others. I'm not sure everyone will get the two-dimensional FFT going and debugged, but give it a try.

1. To get started, write a C program that just reads in an image in PPM raw format, reports its width and height in pixels, and writes out another copy of the same image. Then make a program that creates a small test image of your own with some simple shape in your favorite color.

2. Start with the program you wrote above, and add the feature of further saturating or desaturating each color. This might be useful for making pastel versions of images for web-page backgrounds. This program is like the "color" control in your TV, except there are three "knobs," one for red, green, and blue. Try displaying some images with only one color on at a time. Try also with only one color off at a time. Try some pastels. Interpret your results.

   If you have time and the inclination, experiment with other point-by-point operations on the pixel values. Try nonlinear transformations of the pixel values: nonlinear maps of the range $[0, 255]$ to the same set. Can you emulate the effect of the "contrast" control on your TV? How about the transformation that maps the pixel value u to $255 - u$? There are lots of possibilities here for interesting effects, including what is called *solarization*. Russ's handbook [3] contains a wealth of information about this and many other image-processing topics, and has a tremendous collections of examples. I recommend at least thumbing through it, especially if you're looking for a project in this general area.

3. One common reason to process an image is to "sharpen" it, in order to make it look better in some subjective sense, or to make it easier to locate the boundaries of an object. This falls under the rubric of "image enhancement." A very simple technique is to use some form of a differentiator, or gradient operator [4]. For example, if an image is represented by the pixel values $u(x, y)$, the gradient can be approximated in the simplest way with first differences. The derivative in the x direction is approximated by

$$\Delta_x(x, y) = u(x + 1, y) - u(x, y) \tag{1}$$

   and that in the y direction by

$$\Delta_y(x, y) = u(x, y + 1) - u(x, y) \tag{2}$$

   The magnitude of the vector with these x and y components,

$$G(x, y) = \sqrt{{\Delta_x}^2 + {\Delta_y}^2} \tag{3}$$

   can be used to generate a new image, which is in some sense the derivative of the original. Note that we really have three images to deal with, a red, a green, and a blue one. Computing three of these derivatives, one for each color, will give as a derivative that preserves color information. That is, red in the derivative will indicate a large gradient in the red part of the image.

   Another possible simple way to approximate an image gradient is to use first differences in the diagonal directions, which is called the *Roberts gradient* [4, 3]:

$$\Delta_1(x, y) = u(x + 1, y + 1) - u(x, y) \tag{4}$$

$$\Delta_2(x, y) = u(x + 1, y) - u(x, y + 1) \tag{5}$$

Try at least one of these gradients on an image with sharp boundaries. If you have time to try both, compare their performance. There are many other more sophisticated methods for image sharpening covered in [3].

Hint: You need to keep the pixel values in the range $[0, 255]$, which usually means renormalizing the values once you do calculations with them.

4. For the final part of the basic part, implement the FFT of an image. Again, there are really three images in one color image, so take the transform of all three. (But don't be profligate in your use of storage.)

   Display the magnitude of the FFT (the result is complex) for an image with some periodicity to illustrate the fact that the transform is a frequency-domain representation. Hint: as pointed out in [4], the high-frequency components of the Fourier transform typically die off quickly and are hard to see in an intensity plot. They therefore recommend plotting

   $$\log(1 + |F|) \tag{6}$$

   instead of $|F|$, where $|F|$ is the magnitude of the complex transform F. Note that you need three of these, one for each color. Explain what this transformation of intensities does. You might want to try it on the original image.

   Another hint on getting a display that's easy to interpret: Part of this exercise is figuring out just where the high and low frequencies are in the transform image. Once you do that, it might be convenient to display the transform so the zero-frequency point for both the horizontal and vertical frequency axes is in the center.

   Finally, take the FFT of the FFT to check your program. You should get back to the original image – but with one difference. What is that difference? How can you fix that easily in the the frequency domain?

Assignment: Extra-credit Part

1. You can implement a lowpass filter in the frequency domain first transforming, then masking out the high frequencies, and then transforming back to the spatial domain. Try this out on an image with some detail, and observe the expected blurring. What percentage of the transform image can you mask out without introducing "unacceptable" (to you) blurring? This is a crude form of image compression.

2. Try masking out low frequencies instead of high frequencies. What effect do you obtain? Compare to the results using the gradient above.

3. Look at the transform of the third sample image, the biker. You should see lines and blobs in the frequency domain that correspond to periodicities in the image, either the artifacts due to screening or periodicities in the image itself. Experiment with masking them out. Find other examples of periodicities in images showing up clearly in the Fourier transform, and see what happens when you try to remove them.

References

[1] W. Kou. *Digital Image Compression: Algorithms and Standards.* Kluwer, Boston, 1995.

[2] K. Steiglitz. *A DSP Primer, with Applications to Digital Audio and Computer Music.* Addison Wesley, Menlo Park, Ca., 1996.

[3] J. C. Russ. *The Image Processing Handbook.* CRC Press, Boca Raton, Fla., 1992.

[4] R. C. Gonzalez and P. Wintz. *Digital Image Processing.* Addison Wesly, Menlo Park, Ca., second edition, 1987.