B657 Assignment 1: Image Processing and Recognition Basics

Spring 2018
Due: Tuesday February 6, 11:59PM
Late deadline: Thursday February 8, 11:59PM (with 10% grade penalty)

In this assignment, you'll get experience with frequency domain analysis, implement several basic image operations, and apply them to an object detection problem.

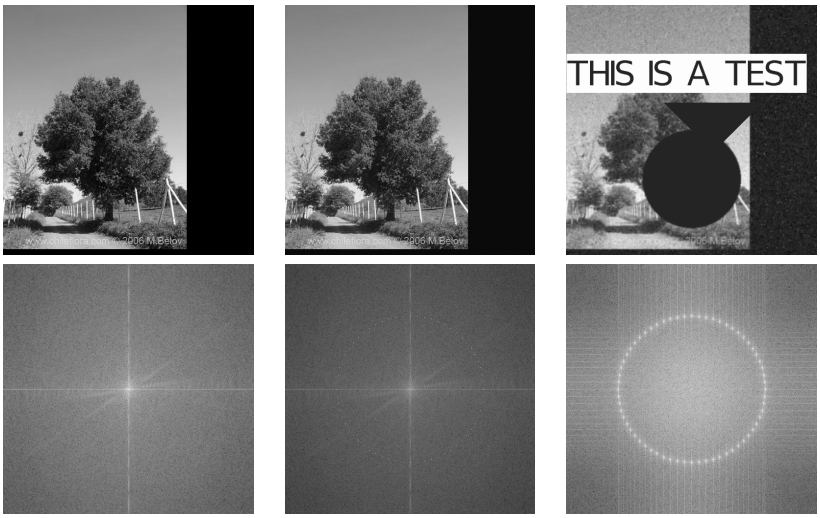
We've assigned you to a group of other students according to your stated preferences. You should only submit **one** copy of the assignment for your team, through GitHub, as described below. After the assignment, we will collect anonymous information from your teammates about everyone's contributions to the project. In general, however, all people on the team will receive the same grade on the assignment. Please read the instructions below carefully; we cannot accept any submissions that do not follow the instructions given here. Most importantly, **start early,** and ask questions on Piazza.

We recommend using C/C++ for this assignment, and we have prepared some skeleton code that may help you get you started (see details below). We recommend using C/C++ because computer vision algorithms tend to be compute-intensive, and your code may be frustratingly slow in higher-level languages (like Matlab, Java, etc.). You may choose to use a different programming language, with the restriction that you must implement the image processing and computer vision operations yourself, **and** your code must run on the SICE Linux servers (e.g. tank.soic.indiana.edu). For example, you may use Python, but you should implement your own convolution and edge detection (and not use some existing Python libraries). You do not have to re-implement image I/O routines (i.e. you may use Python libraries for reading and writing images). You may use libraries for routines not related to image processing (e.g. data structures, sorting algorithms, matrix operations, etc.). It is also acceptable to use multiple programming languages, as long as your code works as required below. You do not have to use our skeleton code, but if you make your own, make sure that your program can be invoked in the same way (i.e., takes the same command line parameters and creates the same output files in the same format). If you have any questions about this policy, please ask the course staff.

Academic integrity. You must follow the academic integrity requirements described on the course syllabus. In particular, you and your partners 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 your group submits must be your own work, which you personally designed and wrote. You should not use code that you find online or in other resources; the point of this assignment is for you to write code, not for you to recycle the code from others. If you do use any code, you must explicitly indicate the source of the code (e.g. URL, book citation, etc.), and indicate exactly which part(s) of your program were borrowed from other sources (by mentioning in the project report, and then explicitly labeling the line(s) of code using comments in the source files). You may not share written code with any other students except your own partners, nor may you possess code written by another student who is not your partner, either in whole or in part, regardless of format.

Part 0: Getting started

1. You can find your assigned teammate(s) by logging into IU Github, at `http://github.iu.edu/`. In the upper left hand corner of the screen, you should see a pull-down menu. Select `cs-b657-sp2018`. Then in the yellow box to the right, you should see a repository called *userid1-userid2-userid3*-a1, where the other user ID(s) corresponds to your teammate.
2. We recommend using the CS Linux machines for your work, e.g. `tank.soic.indiana.edu`. After logging on to that machine via ssh, clone the github repository:



(a) (b) (c)

Figure 1: Watermarking process: image (a) is injected with a watermark, creating image (b) that looks nearly identical but has an identifiable pattern on its spectrogram (a circle – you may have to zoom in). Even after extensive editing (c), the watermark is still visible in the spectrogram and can be detected. (We have exaggerated the spectrograms so that the watermark shows up more clearly.)

`git clone https://github.iu.edu/cs-b657-sp2018/`*your-repo-name*`-a1`

where *your-repo-name* is the one you found on the GitHub website above. (If this command doesn't work, you probably need to set up IU GitHub ssh keys. See Piazza for help.)

3. This should fetch some C/C++ skeleton code and test images. Compile and run the sample programs:

```
make
./watermark image.png
./detect image.png
```

The code has been tested on the SICE Linux machines; you may use another development platform (Windows, MacOS, etc.), but you may have to modify the skeleton code to get it to compile. It doesn't do very much at the moment, simply producing some placeholder output images.

Part 1: Fourier domain

Let's say you want to embed a digital watermark in an image – some data (e.g. the name of the photographer or copyright information) that is nearly invisible and very difficult to remove. Of course, there are simple ways to watermark (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 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! Even our simple version will be remarkably robust – see Figure 1.

Our skeleton code includes a function that computes the DFT for you. Given an image, it returns two SDoublePlane 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 the inverse DFT (IDFT), which takes a real and imaginary buffer pair and converts back to a single image.

To simplify the problem, assume the watermark we want to inject is an integer N (e.g. 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, compute the fourier transform, which produces a 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). 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 of a real (not complex-valued) images is that its real plane 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 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.

Now complete the following steps.

1. To understand the DFT better, it's useful to look at visualizations. 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:

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

   You can then save the resulting image buffer to an image file and view it as if it were an image.
2. In your git repo, `noise1.png` has suffered a 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. Now 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 2: Detecting objects

Let's start by implementing some basic convolution and edge detection routines.

1. Implement a function that convolves a greyscale image I with an arbitrary two-dimensional kernel H. Make sure your code handles image boundaries in some reasonable way.
2. Implement a function that convolves a greyscale image I with a separable kernel H. Recall that a separable kernel is one such that $H = h_x^T h_y$, where h_x and h_y are both column vectors. Implement efficient convolution by using two convolution passes (with h_x and then h_y), as we discussed in class. Make sure your code handles image boundaries in some reasonable way.
3. Use your convolution code to implement an edge detector, using for instance the Sobel operator followed by a threshold on the gradient magnitude. The result from each of the above convolution functions should be nearly exactly the same.

Now, let's use convolutions and edge detection to try to find objects in images. Let's consider a manufacturing inspection scenario, where the goal is take photos of populated printed circuit boards (PCBs) and place bounding boxes around all of the integrated circuits (ICs), as in Figure 2. ICs are usually rectangular, dark colored, have pins along two or more sides, and often have silk-screened labeling on top. We've provided PCB images in your repository. Examine these images and develop a strategy for locating ICs and marking them with bounding boxes. The goal is to correctly find as many cars as possible, with few false positives. As is usually the case in computer vision, it may not be possible to achieve 100% accuracy on the test images – your goal is just to find a strategy that works as well as possible on as many of the images as possible. (There very well may be objects that are ambiguous, depending on the exact definition of IC. Unfortunately, this is a fact of life in computer vision, where competitions are sometimes won or lost based on whether you define a motorbike to be the same as a motorcycle, or whether a painting of a bird should count as an actual bird, etc.)

This problem is purposely open-ended. How should you go about it? It's up to you, but here are a few ideas. You could use edge detection and Hough transforms to find lines and rectangles, filtering out rectangles that do not have reasonable size or aspect ratio, or that are not dark in color. You could use a "sliding window" approach, where you crop out a few IC images manually to use as car "templates," and then use cross correlation to find local image regions that are similar to one or more of the templates. You could clip out small regions of the image at random, compute a Fourier Transform for each one, and see if the frequency domain features are similar to known IC reference models. For all of these techniques, you'll need to do some experimentation to find parameters and thresholds that work well for this task. You may need additional heuristics, like non-maximal suppression to prevent the same car from being detected multiple times.

Your code should output at least three files (useful both for your debugging and our grading purposes):

1. `edges.png`: The result of the edge detector of step 3 above.
2. `detected.png`: Visualization of which ICs were detected (as in Fig 2(b)).
3. `detected.txt`: A text file indicating the detection results. The text file should have one line per detected IC, with the following format for each line:

```
<row> <col> <height> <width> <confidence>
```

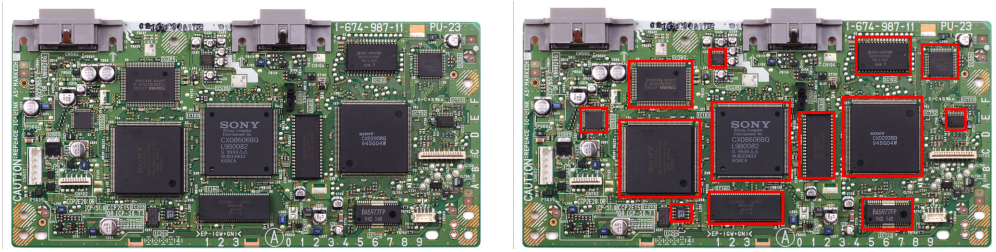


Figure 2: (a) Sample PCB image, and (b) sample output of the IC detector.

where row and col are the coordinates of the upper-left corner of the bounding box, height and width are its dimensions, and confidence is a number that is high if the program is relatively certain about the detected symbol and low if it is not too sure (which again you can compute however you'd like).

Evaluation. To help you evaluate your code, we'll provide some test images and also an evaluation program that will compare your output to our ground truth. Information about this will be posted to Piazza shortly. Please present these results in your report (see below). A small portion of your grade will be based on how well your system works compared to the systems developed by others in the class. Show some results on sample images. We may also give extra credit for additional work beyond that required by the assignment.

What to turn in

You should submit:

1. Your source code.
2. A short report, as either a PDF, or as a Readme.md file in GitHub (which allows you to embed images and other formatting using MarkDown). Your report should explain how to run your code and any design decisions or other assumptions you made. Report on the accuracy of your program on the test images. When does it work well, and when does it fail? Give credit to any source of assistance (students with whom you discussed your assignments, instructors, books, online sources, etc.). How could it be improved in the future?

To submit, simply putting the finished version in your GitHub repository (remember to `add`, `commit`, `push`) — we'll grade whatever version you've put there as of 11:59PM on the due date. To make sure that the latest version of your work has been accepted by GitHub, you can log into the github.iu.edu website and browse the code online.