B657 Assignment 1: Image Processing and Recognition Basics

Spring 2016
Due: Tuesday February 9, 11:59PM
Late deadline: Thursday February 11, 11:59PM (with 10% grade penalty)

In this assignment, you'll implement several basic image operations, including image resampling, convolution, and edge detection, and then apply them to an object detection problem.

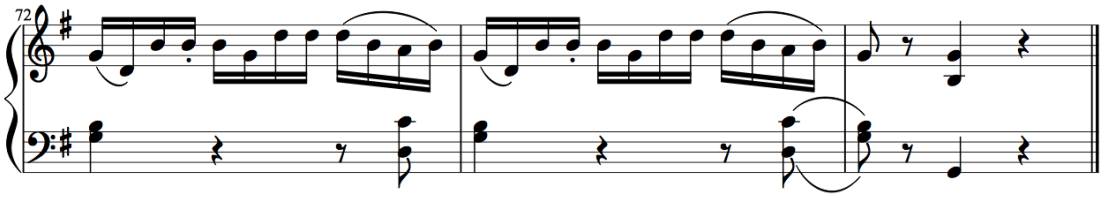
We've assigned you to a group of other students. You should only submit **one** copy of the assignment for your team, through GitHub, as described below. After the assignment, we will collect anonymous information from your teammates about everyone's contributions to the project. In general, however, 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 follow the instructions given here. Most importantly: please **start early,** and ask questions on the OnCourse Forum or in office hours.

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 when implemented in higher-level languages (like Matlab, Java, Python, etc.). However you may choose to use a different programming language, with the restriction that you must implement the image processing and computer vision operations yourself. For example, you may use Matlab, but you should implement your own convolution, edge detection, and resampling routines (instead of using Matlab's built-in functions). You do not have to re-implement image I/O routines (i.e. you may use Matlab's imread and imwrite functions). You may use built-in or third party libraries for routines not related to image processing (e.g. data structures, sorting algorithms, etc.). It is also acceptable to use multiple programming languages, as long as your code works as required below. If you have any questions about this policy, please ask the course staff.

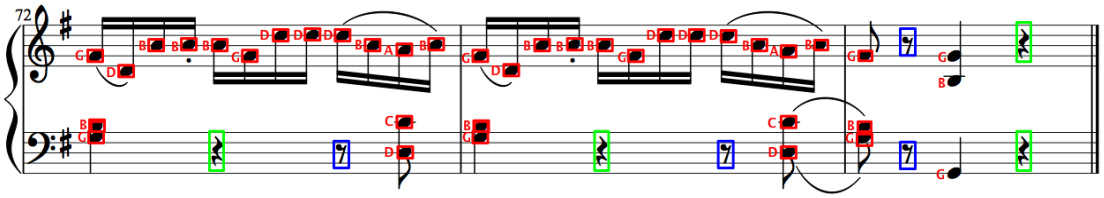
Academic integrity. You must follow the academic integrity requirements described on the course syllabus. In particular, you and your partners may discuss the assignment with other people at a high level, e.g. discussing general strategies to solve the problem, talking about C/C++ syntax and features, 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.

Background

In class we briefly discussed the Optical Character Recognition (OCR) problem, in which the goal is to convert scanned images of documents into textual representations. OCR on high-resolution scans of typeset English documents is considered a mostly solved problem, with commercial systems giving nearly (but never quite) perfect accuracy. In this assignment, we'll consider an unsolved problem that is similar in many ways to OCR: optical *music* recognition. This problem, like many in vision, may seem easy at first but is actually quite challenging. In this assignment, you'll make some progress towards a simplified version of OMR, and practice basic image processing and computer vision concepts in the process.



(a)



(b)

Figure 1: *(a):* A portion of a musical score. *(b):* Output from a simple OMR system, where notes are shown in red, quarter rests are shown in green, and eighth rests are shown in blue.

If you know how to read music already, great! If not, you don't need to know very much for this assignment — here's a quick tutorial. Figure 1(a) shows a small part of a piece written for piano. You'll see two sets of five horizontal lines. Each of these sets of five lines is called a *staff*. In the example, there are two staffs: the upper one is called the *treble staff* and holds higher-pitched notes, and the lower one is called the *bass staff* and holds lower-pitched notes. Musical notes are typically written with a *note head* (an oval) and a *stem*, which is a short vertical line that is connected to the head and may point up or down. The pitch of a note is annotated by its vertical position with respect to the staffs; higher pitched notes are placed further up on the staffs. The horizontal dimension is used to annotate time; the pianist plays the notes from left to right, just like reading English text. When multiple notes are written at the same position on the horizontal axis (but at different positions on the vertical axis), the pianist plays these notes simultaneously. The duration that a note should be played is annotated by the shape of the note head; note heads that are filled-in ovals are played with a shorter duration than notes that are not filled in. (The filled-in notes are typically eighth or quarter notes (or shorter), while the open ovals are typically half or whole notes; you do not need to know the details of timings for this assignment, although it might be helpful to read about them online.) A *rest* indicates when the pianist is supposed to play no notes; these have different durations as well which are annotated with symbols of different shapes (see the figure). A wide variety of additional symbols and text give other instructions to musicians, including properties like note speed, volume, touch, smoothness, etc. We'll largely ignore these symbols (although you can try recognizing them for extra credit).

The frequency or pitch of a musical note is typically denoted by a letter from A to G. Notes are placed either on staff lines or between them (or on *ledger lines*, which are very short staff lines above or below the main staff lines). For example, a notehead placed on the lowest bass staff line is a G. The position right above that, i.e. the space between the first and second lines, is an A. The line above that is a B, the next space is a C, and so on. For the treble staff, the bottom line is an E, the bottom space is an F, and so on.

The goal of OMR is to take an image of a page of music as input, and produce a symbolic representation of the music as output. Figure 1(b) shows what this output might look like on the image in Figure 1(a). This assignment will walk you through one possible approach to OMR.

What to do

0. For this project, we are assigning you to a team with another one or two students. We will let you change these teams in future assignments. 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 `B657`. Then in the yellow box to the right, you should see a repository called *userid1-userid2-userid3*-p1, where the other user ID(s) corresponds to your teammate.

   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:

   `git clone https://github.iu.edu/cs-b657/`*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 OnCourse wiki for help.)

1. Your directory should now contain some C/C++ skeleton code and test images. Compile and run the sample program:

   ```
   make
   ./omr image.png
   ```

   The code has been tested on the CS 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.

2. Implement a function that convolves a greyscale image I with an arbitrary two-dimensional kernel H. (You can use a brute-force implementation – no need to use Fourier Transforms, although you can if you want.) Make sure your code handles image boundaries in some reasonable way.

3. 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.

4. A main goal of OMR is to locate various musical symbols in the image. Suppose for each of these symbols, we have a black and white "template" – a small $m \times n$-pixel image containing just that symbol – with black pixels indicating the symbol and white pixels indicating background. Call this template T. Now we can consider each $m \times n$-pixel region in the image of a sheet of music, compute a score for how well each region matches the template, and then mark the highest-scoring ones as being the symbol of interest. In other words, we want to define some similarity function $f_T^I(i,j)$ that evaluates how similar the region around coordinates (i,j) in image I is to the template.

   One could define this function $f(\cdot)$ in many different ways. One simple way of doing this is to simply count the number of pixels that disagree between the image and the template – i.e. the Hamming distance between the two binary images,

   $$f_T^I(i,j) = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} I(i+k, j+l)T(k,l) + (1 - I(i+k, j+l))(1 - T(k,l))$$

   This function needs to be computed for each $m \times n$-pixel neighborhood of I. Fortunately, with a small amount of manipulation, this can performed using a convolution operation!

   Implement a routine to detect a given template by doing the convolution above. When a note is detected, it should also estimate the pitch of the note (i.e. letter between A and G). To help you get started, we've supplied an easy test image (music1.png) and three templates (template1.png, template2.png, template3.png). Your program should generate an output image similar to the one in Figure 1(b) showing the symbols that have been detected.

5. An alternative approach is to define the template matching scoring function using edge maps, which tend to be less sensitive to background clutter and more forgiving of small variations in symbol appearance. To do this, first run an edge detector on the template and the input image. You can use the Sobel operator and your separable convolution routine above to do this. Then, implement a version of template matching that uses the following scoring function:

$$f_T^I(i,j) = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} T(k,l) \min_{a \in [0,m)} \min_{b \in [0,n)} \gamma(I(i+a, j+b)) + \sqrt{(a-k)^2 + (b-l)^2}$$

where I and T here are assumed to be edge maps, having value 1 if a pixel is an edge and 0 otherwise, and $\gamma(\cdot)$ is a function that is 0 when its parameter is non-zero and is infinite when its parameter is 0.

Note that computing this scoring function for every pixel in the image can be quite slow if implemented naively. Each of the min's involves a nested loop, each summation involves a nested loop, so computing the score for every pixel (i, j) requires a sextuply-nested loop! However, we can once again use a simple instance of dynamic programming to speed up this calculation. Notice that the above equation can be re-written as

$$f_T^I(i,j) = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} T(k,l) D(i+k, j+l),$$

where

$$D(i,j) = \min_{a \in [0,M)} \min_{b \in [0,N)} \gamma(I(a,b)) + \sqrt{(i-a)^2 + (j-b)^2},$$

and $M \times N$ are the dimensions of I. Notice that $D(i, j)$ has an intuitive meaning: for every pixel (i, j), it tells you the distance (in pixels) to the *closest edge pixel* in I. More importantly, notice that re-writing the equations in this way reduces the number of nested loops needed to compute f_T^I from six to four, because D can be pre-computed. Computing D for all pixels requires four nested loops if implemented naively, but requires only quadratic time if you're clever.

6. The sample image and template we provided above were carefully designed so that the size of the template exactly matched the size of the objects appearing in the image. In practice, we won't know the scale ahead of time and we'll have to infer it from an image. Fortunately, if we can find the staff lines, then we can estimate the note head size, since the distance between staff lines is approximately the height of a note head. To find the staves, one could first find horizontal lines using Hough transforms and then try to find groups of five equally-spaced lines, but this two-step approach introduces the possibility of failure: if a line is not detected properly, an entire staff might not be found. A better approach is to apply the Hough transform to find the groups of five lines directly.

   Implement a Hough transform to do this. Assume that the lines in the staves are perfectly horizontal, perfectly parallel, and evenly spaced (but we do not know the spacing ahead of time). Then the Hough voting space has two dimensions: the row-coordinate of the first line of the staff, and the spacing distance between the staff lines. Each pixel in the image then "votes" for a set of row-coordinates and spacing parameters. Each peak in this voting space then corresponds to the row-coordinate and spacing of a staff line, which in turn tells us where each of the five lines of the staff is located.

7. Now combine the above techniques together to implement a simple OMR system. In this assignment, we'll focus on just detecting the staves and the three symbols shown in Figure 1(a): filled-in note heads, quarter rests, and eighth rests. In particular, your assignment should do the following:

   (a) Load in a specified music image.

   (b) Detect all of the staves using step 6 above. In addition to giving you the staves, this also gives an estimate of the scale of the image – i.e. the size of the note heads – since the space between staff lines is approximately the height of a notehead.

(c) Rescale the image so that the note head size in the image agrees with the size of your note head templates. (Alternatively, you can rescale the note templates so that they agree with the image, you can have a library of pre-defined templates of different scales and select the appropriate one dynamically.)

(d) Detect the notes and eighth and quarter rests in the image, using the approach of step 4, step 5, some combination, or a new technique of your own invention. The goal is to correctly find as many symbols as possible, with few false positives.

Your code should output several files (which are useful both for your debugging purposes and our grading purposes):

(a) `scores4.png`: The result of the convolution in step 4.

(b) `detected4.png`: The visualization of the detected symbols after step 4 (as in Fig 1(b)).

(c) `edges.png`: The image edge map produced in step 5.

(d) `detected5.png`: The visualization of the detected symbols after step 5 (as in Fig 1(b)).

(e) `staves.png`: The original image, with detected staves marked with blue lines.

(f) `detected7.png`: Visualization of which notes were detected (as in Fig 1(b)).

(g) `detected7.txt`: A text file indicating the detection results. The text file should have one line per detected symbol, with the following format for each line:

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

where row and col are the coordinates of the upper-left corner of the bounding box, height and width are the dimensions of the bounding box, symbol_type is one of `filled_note`, `eighth_rest`, or `quarter_rest`, pitch is the note letter (A through G) of the note or is an underscore (_) if the symbol is a rest, and confidence is a number that should be high if the program is relatively certain about the detected symbol and low if it is not too sure.

This assignment is purposely open-ended, with many details left unspecified. For example, 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 note from being detected multiple times. As is usually the case in computer vision, it may not be possible to achieve 100% accuracy on the test images.

Evaluation. Your goal in step 7 is to make an OMR system that works as accurately as possible. To help you, 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 OnCourse 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. We may also give extra credit for additional work beyond that required by the assignment.

What to turn in

Turn in at least two files, your source code and a PDF containing your report, by 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. 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?