B657 Assignment 1: Image Processing and Recognition Basics

Spring 2019
Due: Sunday February 23, 11:59PM
Late deadline: Tuesday February 25, 11:59PM (with 10% grade penalty)

In this assignment, you'll get experience with image operations and apply them to a recognition 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.

You may choose to use any general-purpose 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.sice.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 also 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. All that said, we recommend using either Python with the Pillow library, or C/C++ with the CImg library. No matter what language and library you use, make sure that your program obeys the input and output requirements below (e.g., takes the right command line parameters in the right order, and creates the right output files) since we use testing scripts that automatically test your program. If you have any questions about any of this, please ask the course staff.

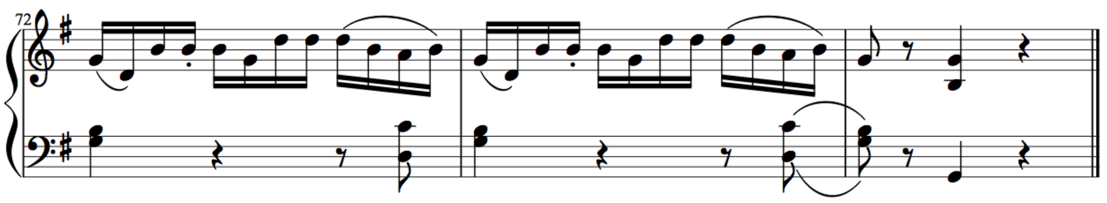
Academic integrity. You must follow the academic integrity requirements described on the course syllabus. In particular, you and your partners may discuss the assignment with other people at a high level, e.g. discussing general strategies to solve the problem, talking about C/C++ or Python syntax and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

The problem

At a major midwestern research university not far away, a hypothetical mildly-successful professor of computer science has a dream of being an accomplished organist. After several semesters of part-time study at one of the world's top music schools, he is already on chapter 3 of the beginner textbook and his performances have received reviews including: "well, that was loud," "I never thought that piece could be played that way," and "you know, you would be a very talented page-turner."

(Meanwhile, other students who have been studying for a single semester are already playing Widor Toccatas, although it is not a fair comparison because they are half his age and also have actual talent. But we digress.)

This hypothetical professor's new goal is to write a program that can play pieces for him automatically. In class we briefly discussed the Optical Character Recognition (OCR) problem, where the goal is to convert


(a)

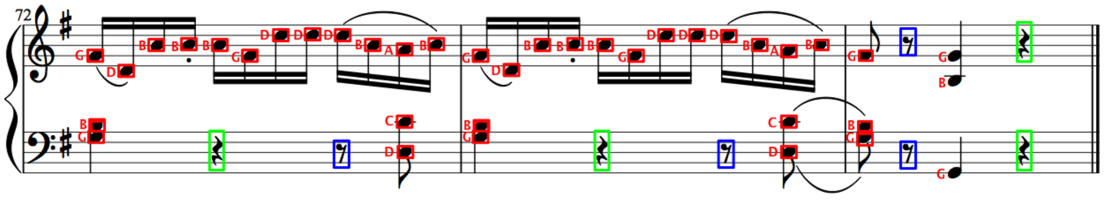

(b)

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.

scanned images of documents into textual representations. In this assignment, we'll consider optical *music* recognition. This problem, like many in vision, may seem easy at first but is actually quite challenging.

Some background

If you know how to read music already, great! If not, 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 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.

Your job is to write an automatic program that will 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).

What to do

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-sp2020`. Then in the yellow box to the right, you should see a repository called *userid1-userid2-userid3-userid4*-a1, where the other user ID(s) corresponds to your teammates.

2. While you may want to do your development work on a local machine (e.g. your laptop), remember that the code will be tested and thus must run on the CS Linux machines, e.g. `tank.sice.indiana.edu`. After logging on to that machine via ssh, clone the github repository:

   ```
   git clone https://github.iu.edu/cs-b657-sp2020/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.) This should fetch some test images and some sample code (see the Readme file in the repo for details).

3. 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 fancy tricks or Fourier Transforms, although you can if you want.) Make sure your code handles image boundaries in some reasonable way.

4. 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.

5. 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 algebra, this can performed using a convolution operation!

   Implement a routine to detect a given template by doing the convolution above.

6. 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 (not required for this assignment).

7. 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.

8. 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 program should be run like this:

   ```
   ./omr.py input.png
   ```

   and 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 two files:

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

(b) `detected.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. Explain your technique in detail in your report (see below).

Evaluation. Your program will almost certainly not work perfectly all the time, and that's okay. To make things fun, we will hold a competition in which we will evaluate the programs on a separate test dataset of unseen examples. 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. 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 Piazza shortly. Please present these results in your report (see below).

Report. An important part of developing as a graduate student is to practice explaining your work clearly and concisely, and to learn how to conduct experiments and present results. Thus an important part of this assignment is a report, to be submitted either 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 both quantitatively and qualitatively. 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? Note that even if your code performs very poorly, you can still write an interesting report that explains what you tried, what the advantages and disadvantages of that approach are, why you think it didn't work, etc. You can think of the report a bit like an argument for why you deserve a good grade on the assignment.

What to turn in

You should submit: (1) Your source code, and (2) Your report, as a Readme.md file in Github. To submit, simply put 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.