**Course:** CVI620

**Task:** Final Assessment

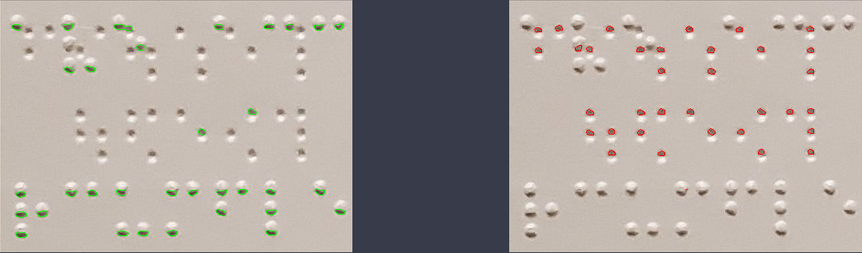
**Student:** Yathavan Parameshwaran

**Date:** April 19, 2020

**Professor:** Vida Movahedi

The program accepts unlimited arguments for image file names with the “.jpg” extension located in the **“project\_files\_dir/Project\_2\_1\_files/”** directory to perform braille character detection. Once detected, as specified in the instructions, it will circle each bump or hole (with a color, for easy viewing by the user) labelling it as such for the user to clearly see what it is.

Example:



The program starts off by initializing some cv::Mat objects to house the image files and binary images that we will have to output as the result of the braille character detection program. We then use imread to locate and read the image data and load it into the Mat object. We will read it as a 3 channel image, and then use cv::cvtColor to convert it into a greyscale image that will be better suited for our purposes. The Mat containers for the binary image can be initialized to the same “dimensions” (rows/cols) as the image we are opening. Set the type to cv::THRESH\_BINARY because… binary.

Once converted to greyscale image, applying some gaussian blurring to the images to reduce the noise. A kernel of 3x3 should be good, because we don’t want to over-blur and dilute the bumps or holes of the braille document, as this will render the program useless.

After blurring, run it through the cv::threshold function to convert it to a binary image. This will greatly help “simplify” the image into a format that will assist the Canny edge detection to identify the edges of the image. I used a thresh value of 150 with a max of 255. This is pretty standard values, so no need to change them.

The Canny edge detection method will be what actually detects the edges of the prominent features of the image. In this case, it is detecting the dark shadows of the bumps and the small “specs” that are the holes.

Now that we have an image with the edges recognized, we can identify the contours using the cv::findContours function. Next, each contour can then be evaluated individually to see if it is a bump or a hole.

The basic idea is pretty simple: each bump is the same and each hole is the same. Is it black or white, no grey area. There is a clear line that seperates the bumps from the holes and that is an advantage we have to writing an algorithm for the detection.

Now that we have the contours, we will loop through each contour using a for loop to push the area of each contour into a vector. Sort the vector by lowest to greatest to get a range, then find the median. The median will be the “splitting point” between the bumps and holes.

Once you have the median, loop through the contours, and push it into either the holes vector or the bumps vector, depending on how the area compares to the median. Once you have two vectors, use the drawContours function to write the data to the Mat arrays, then writing to the filesystem for the user to view.

**Concepts used in this program:**

* Mat arrays
* Thresholding
* Blurring
* Canny edge detection
* Finding contours

Note, for Part II: calculation of confusion matrix, please find it as a seperate code file confusion.cpp in the Part 2 folder on Onedrive.