Carl Xiong

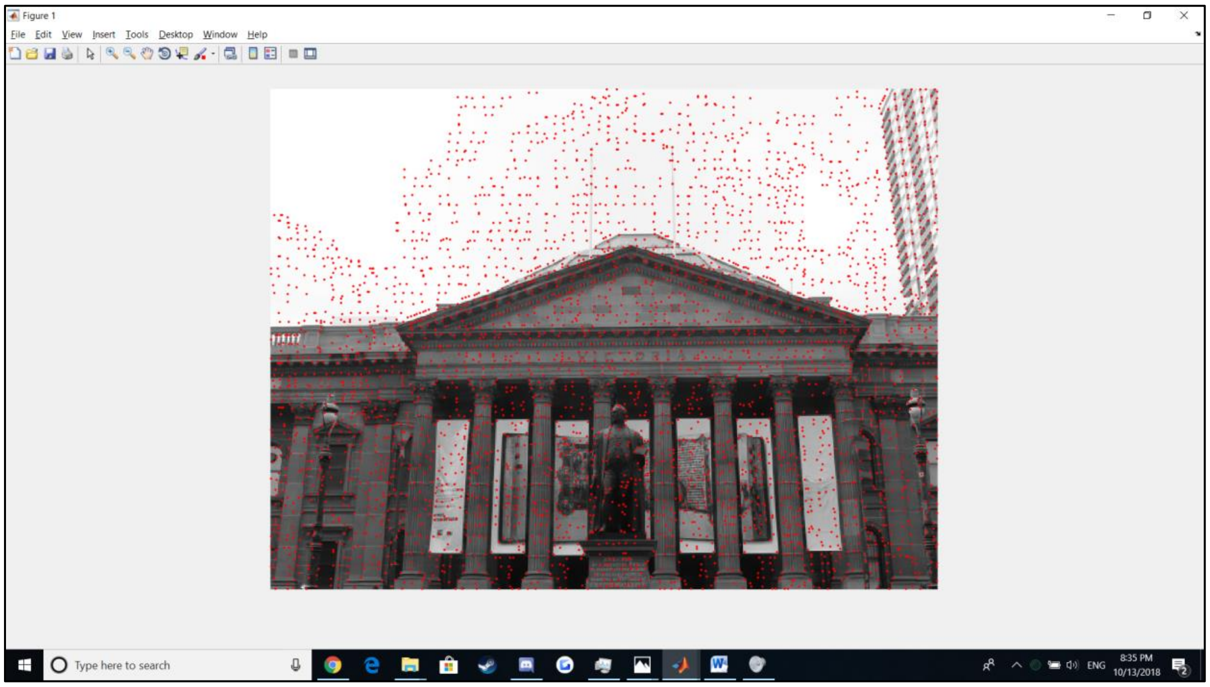
CMSC426

Professor Aloimonos Yiannis

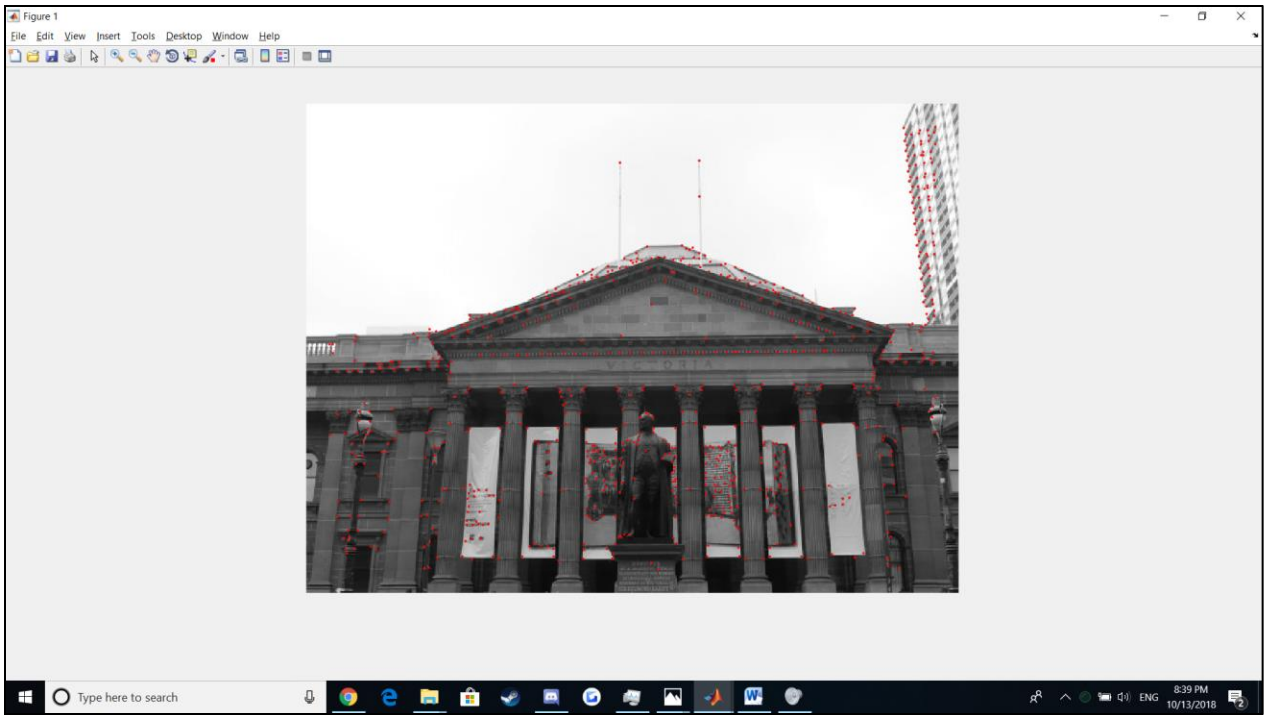
October 13, 2018

CMSC426 – Panorama stitching

The objective of this project is to take in several input images and stitch them accordingly to produce a panorama. This is done with a combination of corner detection, feature matching (RANSAC), and homography warping. The output of the MATLAB code will produce a single panorama image. For the nature of this report, I will be using Image Set 1, the Victoria building. The first step is loading in the image and detecting the corners. Detecting the corners will be accomplished using the built-in "cornermetic()" function. "Cornermetric" will return a matrix of the same image size. It assigns every pixel a corner score. We can use imregionalmax() to find the local maximums of each "corner" to find out which specific pixel is the "center" of the corner. The output of this step is displayed below.



This image is a plot of all the corners the cornermetric() function detected. We need to refine this image, getting only the strongest corners and making sure that they are evenly distributed throughout the image. This step is done with ANMS. ANMS picks the strongest points and selects points that are not close to each other. This is done with a simple distance calculation using the pixel's X and Y coordinate. If the distance is above the threshold, then the pixel is added as a good corner. The output of this step is displayed below.

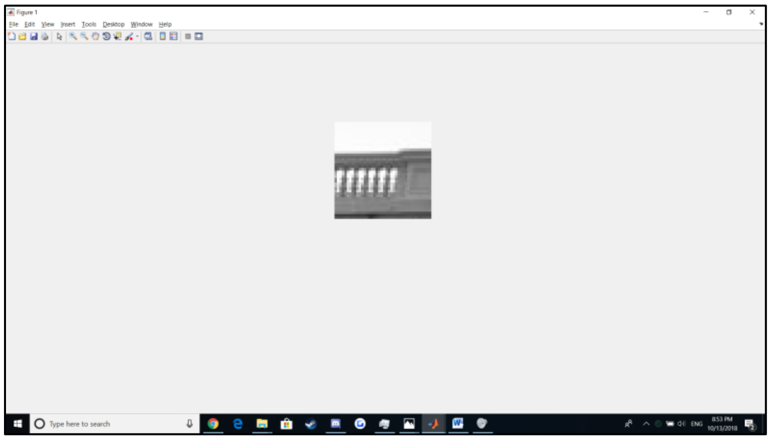


The next part addresses recording the corner's information as a feature descriptor. The steps for creating a feature descriptor for a particular pixel are as follows:

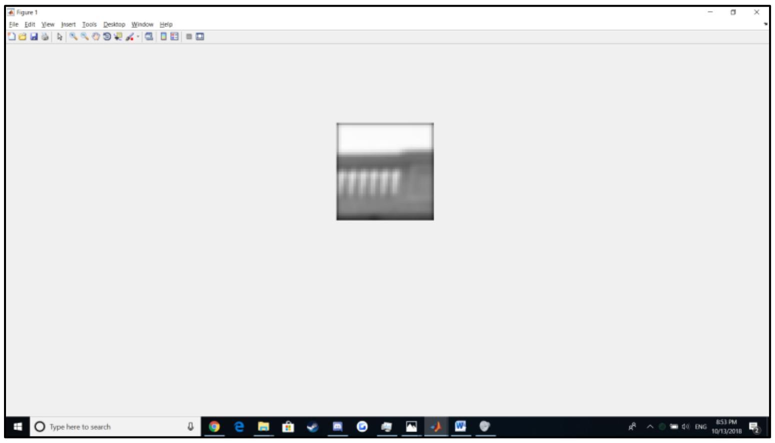
- Take a 40x40 patch of pixels around the designated corner.
- Apply a Gaussian blur to the patch. (This helps reduce effects of different lighting conditions)
- Down sample the blurred patch into a 8x8 matrix
- Reshape the 8x8 matrix into a 64x1 vector

- Standardize the vector by subtracting all the elements by the mean and dividing by the standard deviation.

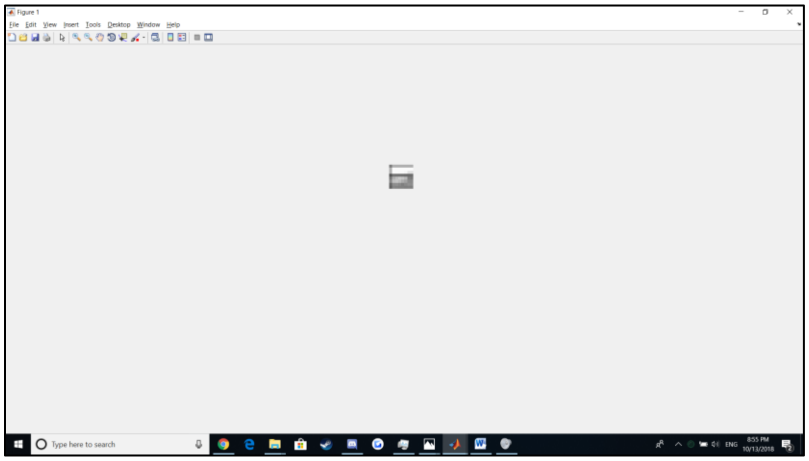
This makes is easier to determine if a corner on an image matches another corner on a separate image. We use a sum of squared differences technique to determine the likeliness of a corner. We test the ratio between the best match and the second best match to determine whether or not the corner is the same. These steps output the following images. (The corner pixel was randomly selected to demonstrate the effect)



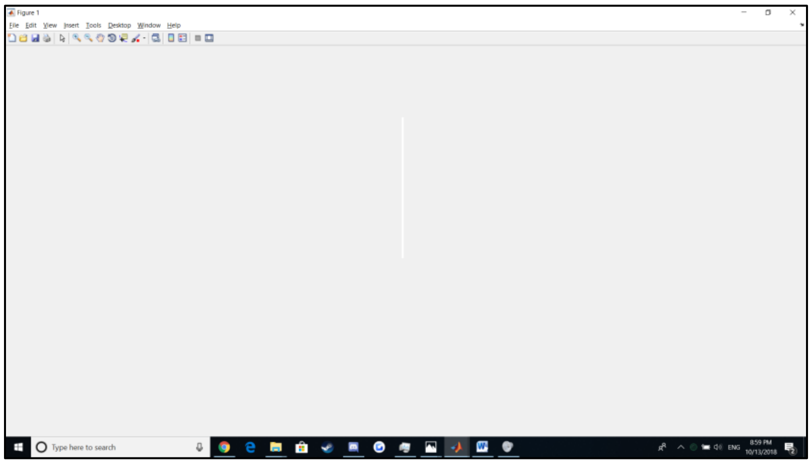
Selecting 40x40 patch



Applying Gaussian blur



Down sampling into 8x8

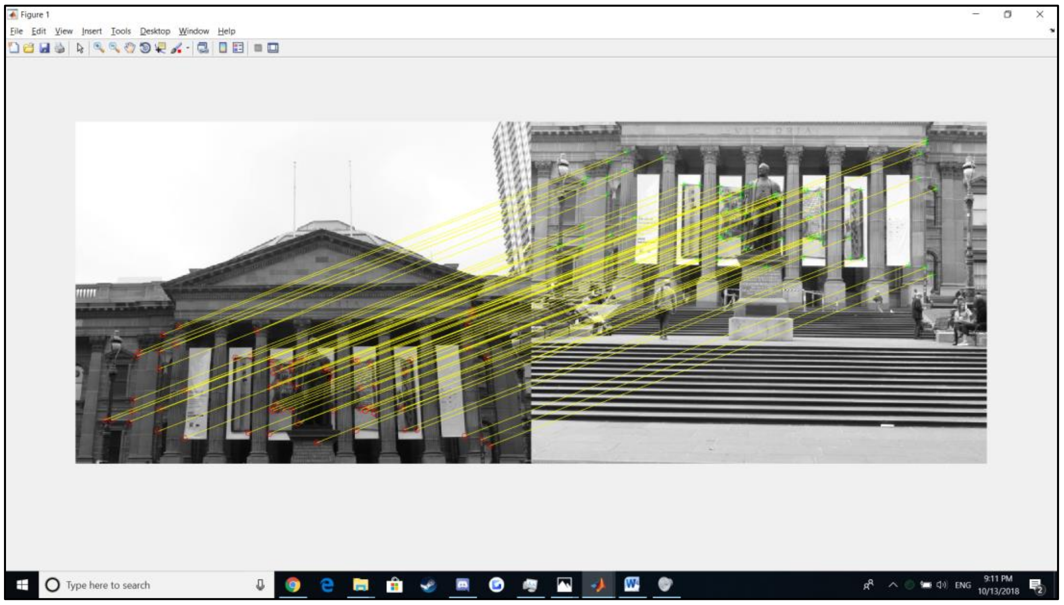


Reshaping into 64x1

(The values are not accurately displayed however all of the 64 pixels are of varying values between 0-255)

The final step is standardizing the 64x1 matrix. The output is not shown but it is similar to the image displayed above. However, since it is standardized, the mean of all the values are centered on 0 and the standard deviation is 1. (E.g. [-0.87, 0.17, 0.15, -1.70, -0.97, -0.62 …])

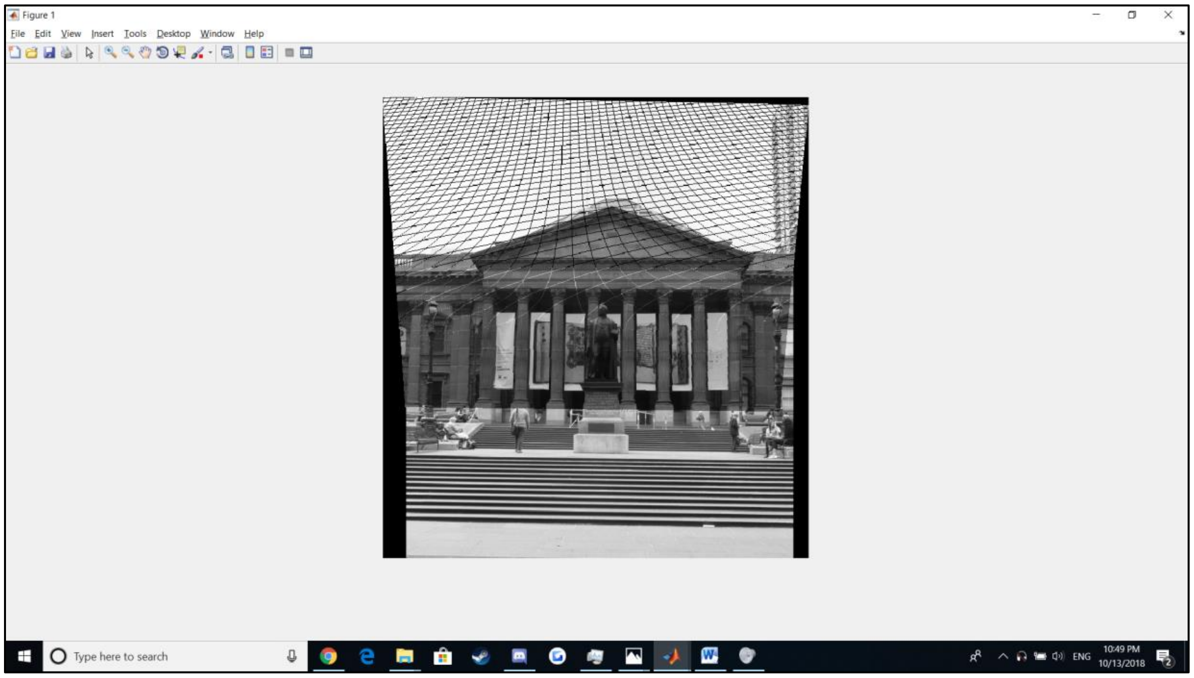
Using this feature descriptor, we can easily calculate whether or not the opposing corner on a separate image is similar enough to be considered the “same” corner. A list of pixel corners on two images is then composed using the SSD (sum of squared differences) applied on the feature descriptors. If the SSD gives a value that is under a user defined threshold, that means the two corners are very similar. The two coordinates are then added to a list containing all the matching corners. The output of this step is displayed below.



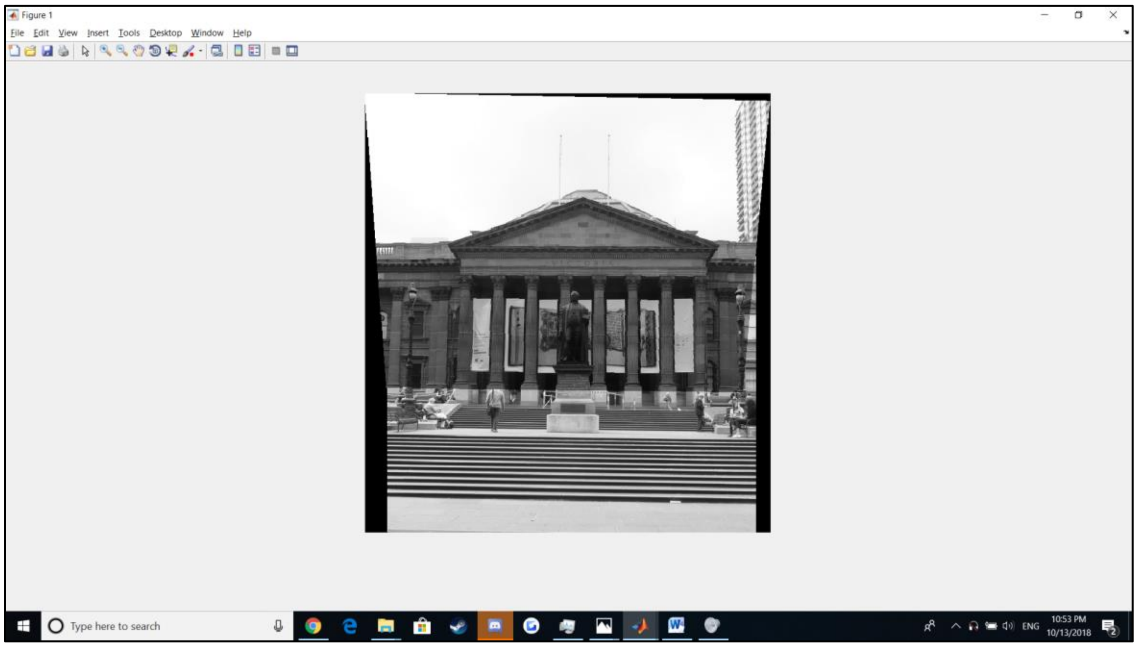
We can pass this list of coordinates into the provided est_homography() function to approximate a transformation matrix for the input images. We can find a more accurate homograhy matrix using the RANSAC technique. The dataset is iterated over continuously and averages out the inliers while throwing out the outliers. The process stops when either, the max number of iterations is finished, or the algorithm hits a threshold percentage for the data.

Using the estimated homography, we can map the pixels from the original image onto the new one, using the corners as an anchor to determine where the picture should exist in the new space. However, we cannot directly map the pixels over as some would go out of bounds. So, we have to create a new image space, a canvas. This canvas should be created by taking the size of the original image and padding it with enough space so that the mapped image can exist on top of the second image.

In order to find out how much padding we need to add, we need to find out “how much” out of bounds the mapped image will be. We can find this easily by mapping the four corners of the original image using the apply_homography() function. The four corners should yield the minimum and maximum coordinates that the mapped image should exist in. Logically speaking, if any of the new (X,Y) coordinates of the mapped corners is either less than 0, or greater than the image size of the new image, then we have to account for that offset. Afterwards we can paste the second image directly over onto the new canvas starting at (0,0) plus the left_offset and the up_offset. This gets the second image onto the canvas. To get the first image, we use the apply_homography() function. We iterate over every pixel in the first image and pass the coordinate with the apply_homography() function. This returns an (X,Y) coordinate pair to the new canvas space. We copy the old pixel value and insert at the designated space. The result is displayed below.



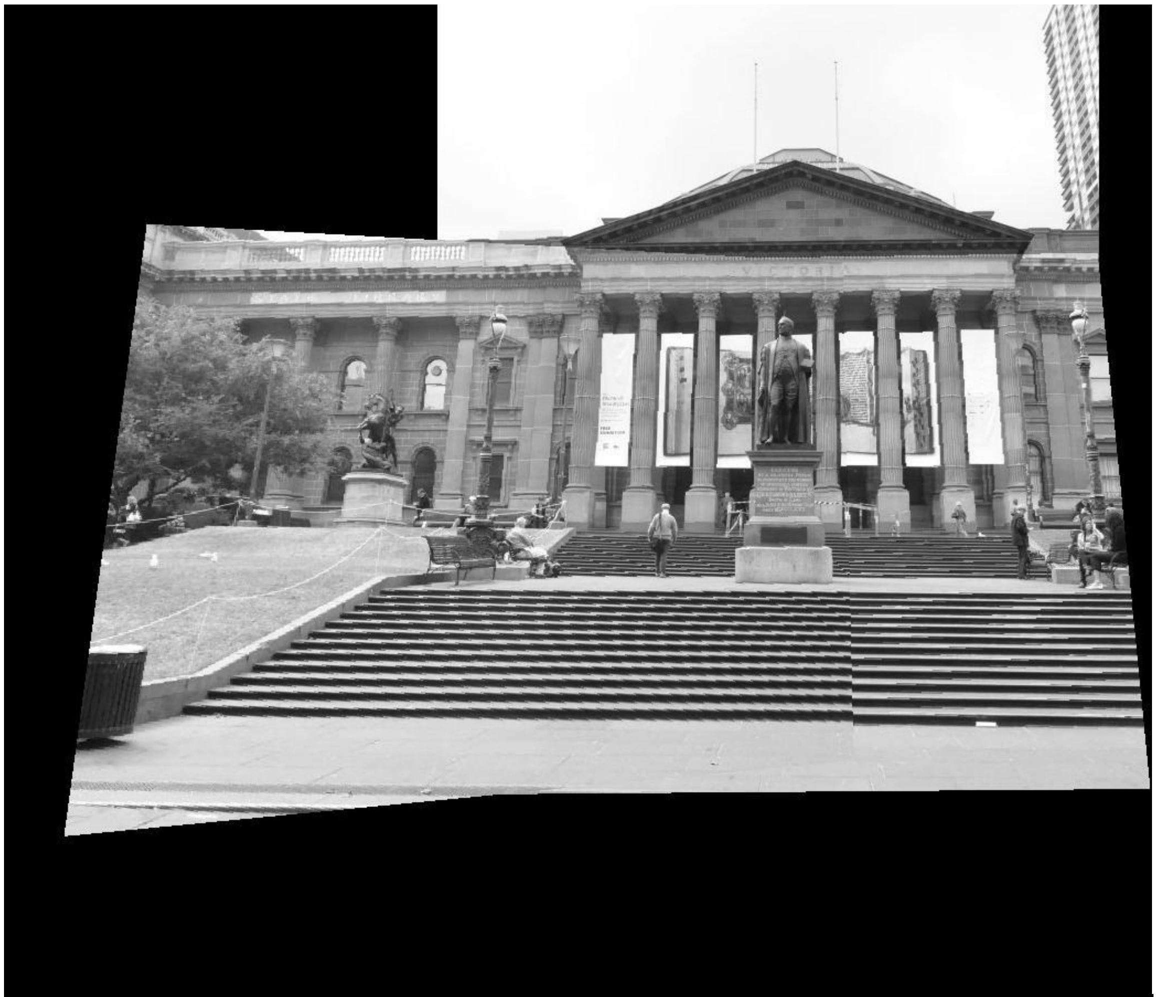
This produces the stitched panorama but there are un-appealing lines strewn though out the image. This is because when scaling an image up, there is not enough data to fill in the spaces. This can be somewhat alleviated by instead of iterating over the first image and finding its coordinates in the canvas, we can iterate over the canvas and find out what pixel value should belong in each pixel. In order to accomplish this, we take the inverse of the estimated homography and apply it to every pixel in the canvas. This will return a (X,Y) coordinate that should exist in the first image. If the (X,Y) is out of bounds in the first image, then we leave the pixel black. The output of this technique is displayed below.

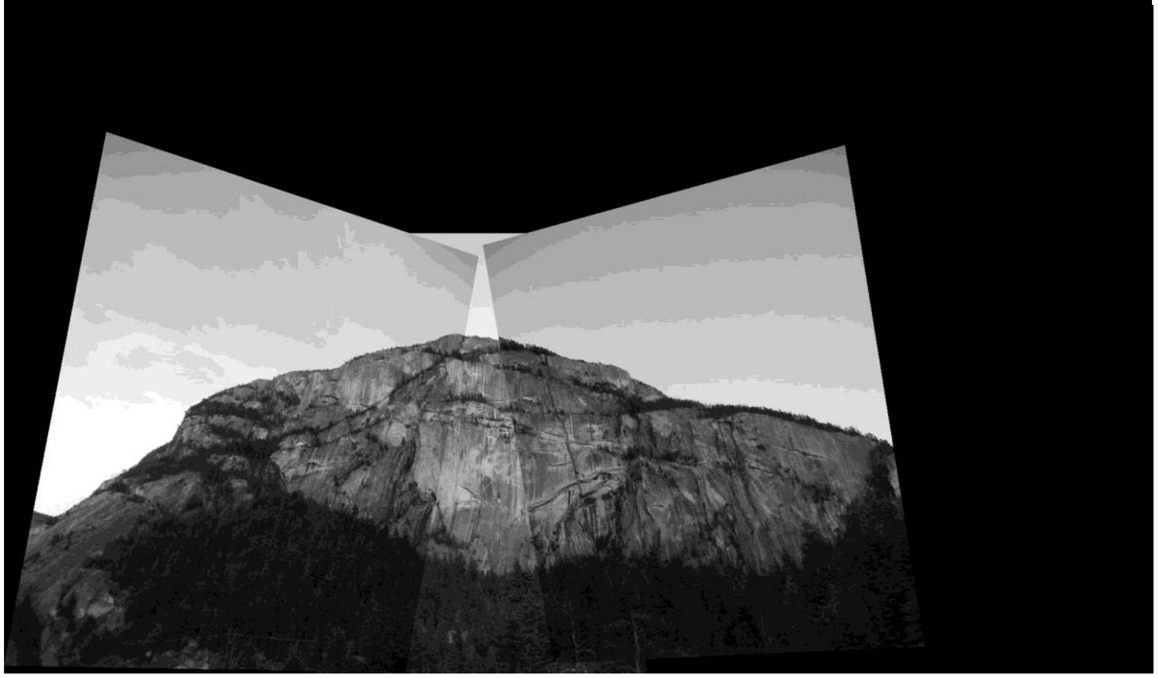


The artifacts resulting from the homography warp are no longer apparent. Now we take this image as one of the inputs and feed it through the program again until we eventually have a complete panorama. All the outputs of the complete panoramas are displayed below.

(NOTE: For some reason, the classroom images just did not work. I tried tweaking all the parameters and thresholds but the result would always be an insanely warped image.)

(NOTE 2: I did not implement the iteration over all the files in the input/folder. In order to create the panoramas, I took the resulting panorama created by images 1 and 2 and saved the matrix/image as a jpg. I then re-ran the program but instead of taking image 1 and 2, I took in the new canvas and image 3. I manually “concatenated” all the images together instead of using a loop to iterate over all the files in the folder.)


VICTORIA



HAMARKSHR.
5000MILUR