# CMSC498D Project 1 Report

William Lin

March 29, 2025

# Introduction

This report will outline the use of basic homographies to rotate, translate, and shrink images and also use SIFT and RANSAC to achieve a panoramic stich of three different perspective images. An annoying problem that was eventually fixed with just one function was that opencv represented its imported images in BGR order whereas `plt.imshow` would treat the pixel values in RGB order. The quick fix was to change this order write after importing in the images. No functions from opencv were affected by this change in order.

# Intro to Homographies

First, I went over the slides shown in the slides from class and found the 3x3 homographies for:
Rotation:

$$\begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Translation:

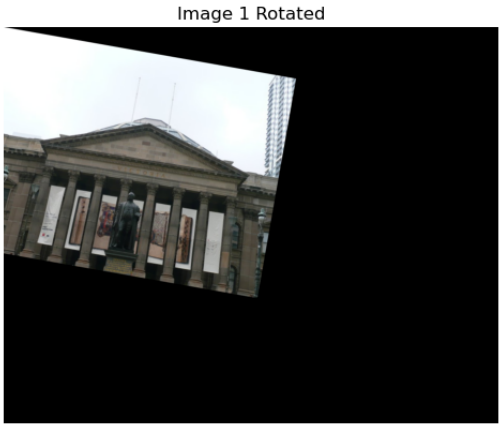
$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

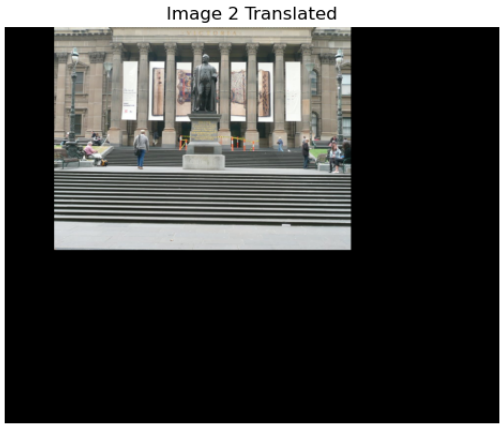
Scale:

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

After that, I looked at the documentation for `cv2.warpPerspective`. At first, I was confused since the documentations used `cv.getPerspectiveTransform` with `cv2.warpPerspective` but I realized applying the homography to an image was as simple as calling `cv2.warpPerspective` with the image, homography, and output shape.

Here were the results:



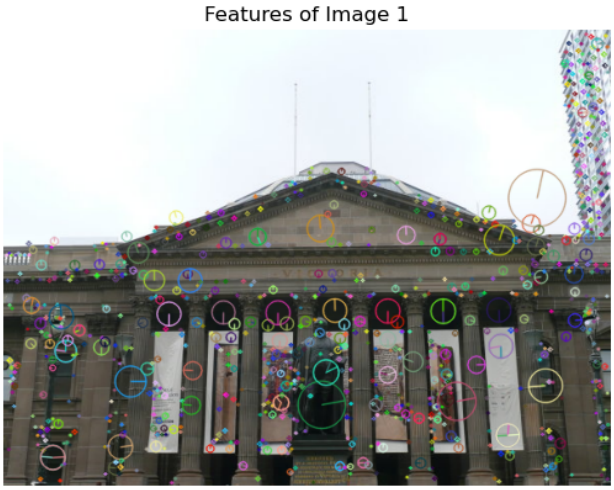

# Panoramic Stitching

For this section, I struggled a bit as I understood the concepts of SIFT and RANSAC from class but had no idea how I would actually implement them. Luckily after looking over Szeliski and OpenCV's Python feature detection documentations, it was actually pretty easy since the library includes everything.

## Compute SIFT features

This part involved creating a SIFT object from cv2, creating a black and white version of the images, and then calling `cv2.detectAndCompute`. I didn't know what the outputs of `cv2.detectAndCompute` were at first since they were only labeled as `kp` and `des`. After looking at documentations and printing the actual values, I learned that `kp` were keypoint objects which simply stores the coordinates of important features on the image and the weight of the features. `des` is a special *numberof keypoints* x 128 matrix which is useful when calculating the distance matrix between two images.

The results when I draw the keypoints on the images are:

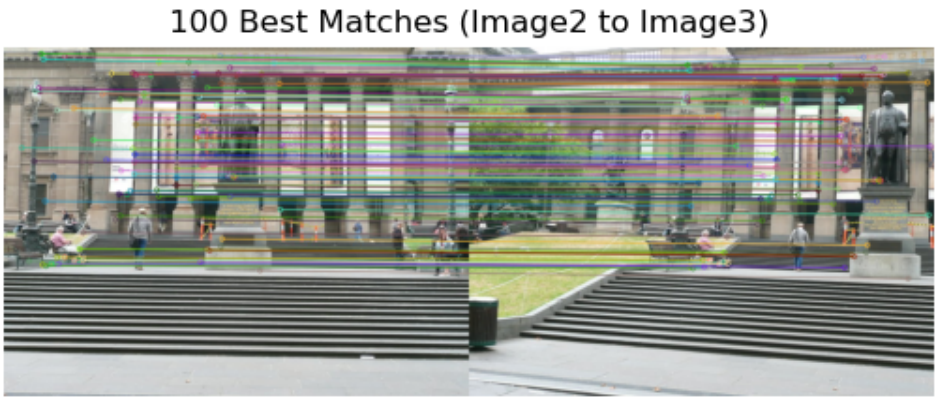





## Match features

This part was shown in the documentations that were referenced above. Using a brute force matching algorithm, I was able to find the 100 best matches between image 2 and image 1/3. First using the `scipy.spatial.distance_matrix` function and passing in the descriptors created earlier, we got a matrix of the $l2$ distance between each feature between image 2 and image1/3. We can then manually find the 100 best matches. However, since there are no limitations for using the opencv library, I used the `BFMatcher` class' `match` function to find all the matches between image 2 and image 1/3. After that, we can simply sort and extract the 100 best images.

Here are the results:


Here are clearer images if we only look at the 5 best matches to really see that the matches are authentic:



## Estimate the homographies

After looking at the documentation for `cv2.findHomography` and feature detection, I saw that it needed to be given the points from our source (image 2) and destination (image 1/3). These were easy to extract from the

keypoints since they were the trainIdx and queryIdx fields respectively. After reshaping, the estimation ran effortlessly by passing the points, the method (RANSAC), and threshold of 2. The function outputted the 3x3 homography and a mask which wasn't useful in our case.

The estimated homographies were:

$$H_{1\text{to}2} = \begin{bmatrix} 1.12039948 & 0.08471577 & -39.2280603 \\ 0.01822429 & 1.14793726 & -268.315104 \\ 0.00001364 & 0.00033560 & 1.00000000 \end{bmatrix}$$

$$H_{3\text{to}2} = \begin{bmatrix} 1.24009077 & 0.00328655 & -297.852987 \\ 0.07414743 & 1.16483208 & -38.8472409 \\ 0.00038257 & 0.00001256 & 1.00000000 \end{bmatrix}$$

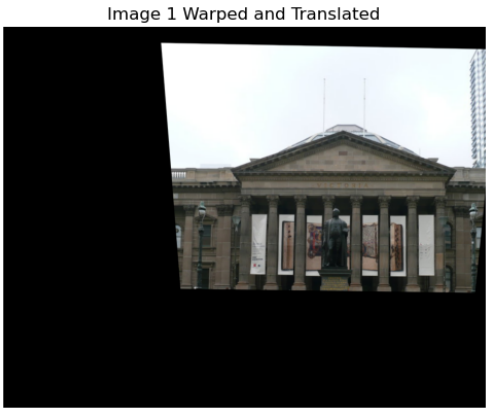
## Warp and translate images

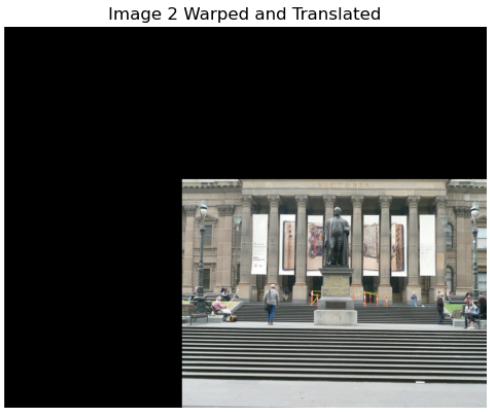
Translating worked the same as part 2 except now we translate 350 pixels right and 300 pixels down which looks like

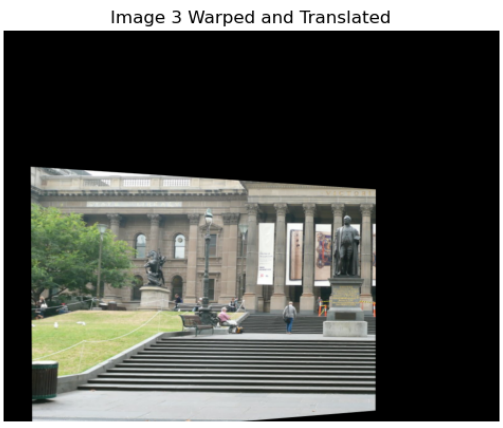
$$H_{\text{translate}} = \begin{bmatrix} 1 & 0 & 350 \\ 0 & 1 & 300 \\ 0 & 0 & 1 \end{bmatrix}$$

At first, I wasn't sure how I would translate and then warp and so I tried to first call `cv2.warpPerspective` with $H_{\text{translate}}$ and then $H_{1 \text{ to } 2}/H_{3 \text{ to } 2}$. However, this didn't work. But then I remembered back in the slides that performing a matrix multiplication would achieve a composite homography matrix that did what we needed. There was also the problem of having a larger image as a result of the stiching which was fixed by adding the extra width of 350 from the translation and the extra height of 300 from the translation. After that, the images were ready to be warped and stiched.

The separate pieces before the stich looks like this:




The final panorama was this:

Panorama