CMSC 426 - P2

Gudjon Einar Magnusson

October 16, 2016

1 Capture Images

For this project I used 5 sets of images, with 3 to 8 images in each set, to test my code. All the images are shown in figure 1.

2 Feature Correspondence

2.1 ANMS

The ANMS algorithm is implemented in the the function *anms*. It takes two arguments, a matrix of corner metrics C and the max number of corners to return, n. It returns two vectors, x and y with the coordinates of the points.

The initial set of points is picked by finding peaks in corner metric matrix using the *imregionalmax* function. The points are then ranked based on their distance form other points and the top n points are returned. This results in a set of points that are roughly evenly spread across the image. In my implementation I generally use $n = 300$. I tried higher values but got little benefit from it.

2.2 Feature Descriptor

Feature descriptors are used to identify each corner across images. To do this I implemented the function *fdescript*. I takes three arguments, an image I and two vectors r and c with the index of the points of interest.

For each point I take the surrounding 40×40 pixel region and apply a Gaussian blur with sigma equal to 4. From that result I take the value

from every fifth row and every fifth column, resulting in 64 values. Those values are normalized by subtracting the mean and dividing by the standard deviation. The normalized values serve as the descriptor for that corner.

2.3 Feature Matching

Feature matching is implemented in the function *matchFeatures*. It takes four arguments, the corner coordinates and descriptor values for each of the two images we want to pair.

For every point it calculates the square difference from every other point by comparing the descriptor values. The point pair is accepted only if the difference is low and its significantly lower than the next best batch. In my implementation I only accepted the point pair if the lowest difference divided by the next lowest difference was lower than 0.3.

2.4 s

ectionRANSAC

The RANSAC algorithm is implemented in the function *ransac*. It takes three arguments, two sets of points, $M1$ and $M2$, and the maximum number of iterations, $iter$.

3 Output

To put together a set of images I, I first estimate the homography from each image to the first image, the set of homographies is called H. The first homography H_1 is simply the identity matrix, H_2 is the projection from I_2 to I_1, H_3 is the projection from I_3 to I_1, and so on. To do this I assume that each image in the set can be matched with the one before it. Then I can project each image to the first one multiplying H_n with H_{n-1}.

This makes it easy to project one image after another as long as the images are given in the correct order. The problem wit this approach is that as we move further from the first image the distortion gets worse. Long panoramas look good on one end but are distorted at the other. A better way is to project all the images to the center image. To do that I multiply each homography with the inverse of the center homography. Figure 2 shows how the choice of a center image affects the panorama.

To transform a set of images once I have the homography I implemented the function *transformSet*. It takes two arguments, a set of images I and a set of homographies H. It returns a set of transformed images and a set of binary masks that shows what pixels the images occupy.

The transformation is implemented using the built in Matlab functions *maketform* and *imtransform*.

4 Cylindrical Projection

Cylindrical projection is implemented in the function *cylindrical*. It takes two arguments, an image I and focal length f, and return another image C that is the same size as I. When mapping pixels from the source image to the their destination confidante I simply use floor of x' and y'. Figure 3 shows and example of the output of from *cylindrical*.

Lower values of f increase the amount of warping in the image, kinda like lowering the radius of the cylinder that the image is projected on.

5 Blending

The simplest method I could come up with to combine the image was to pick the maximum value of each pixel. This produces a fairly good result for the landscape image, except for a slight difference in brightness. For scenes closer to the camera there is a big problem, objects that move or don't match perfectly show up as ghosts. Figure 4 shows an example of the ghosting effect.

I'm currently implementing a technique that borrows from my seam carving implementation to find a low energy seam through the image intersection and uses that to get a better merging line. That might fix the issue of hard lines passing through objects. It would not fix the brightness issue though. I'll try to get to that.

6 Results

Turns out images very close to the camera are very difficult to get right.

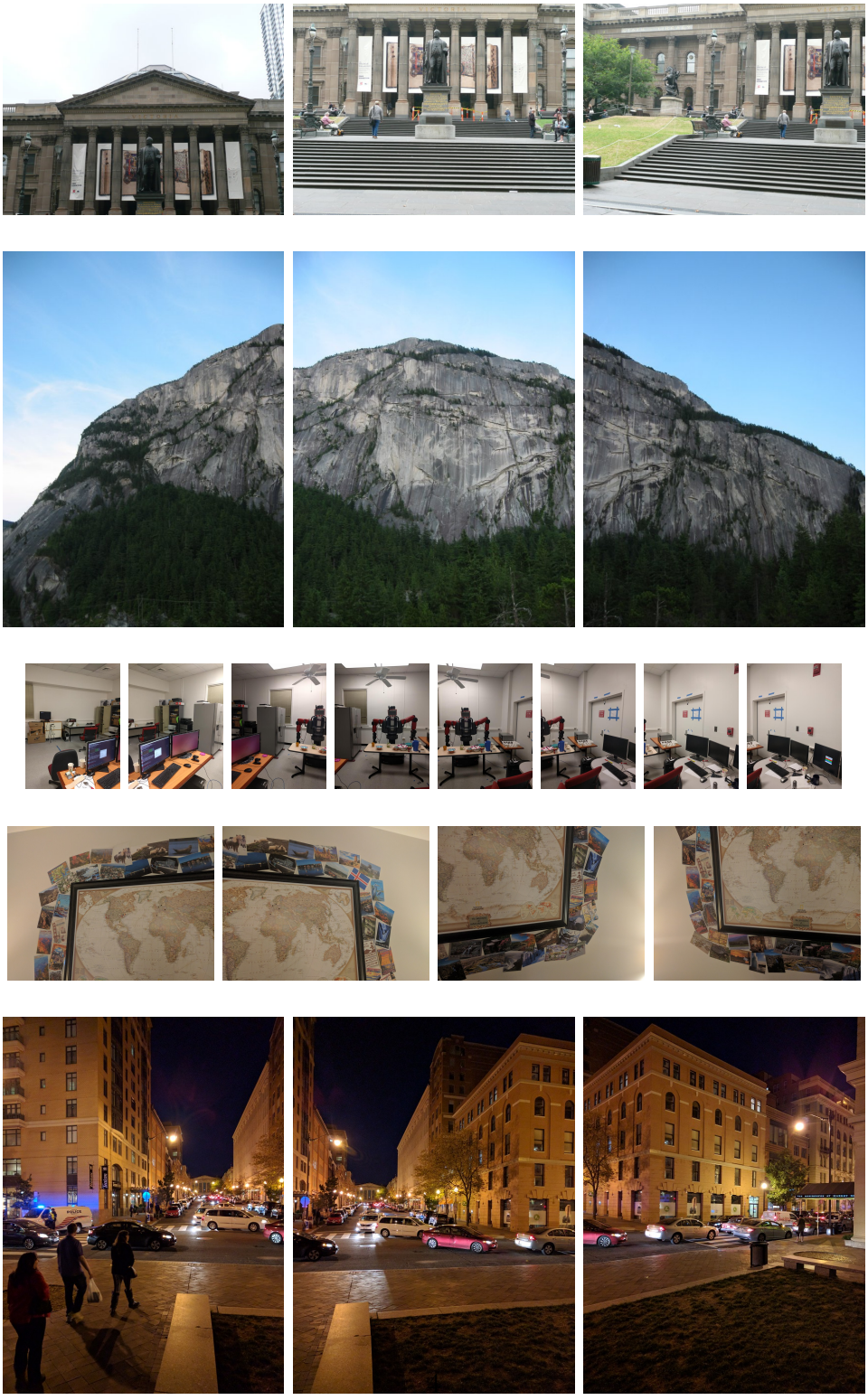
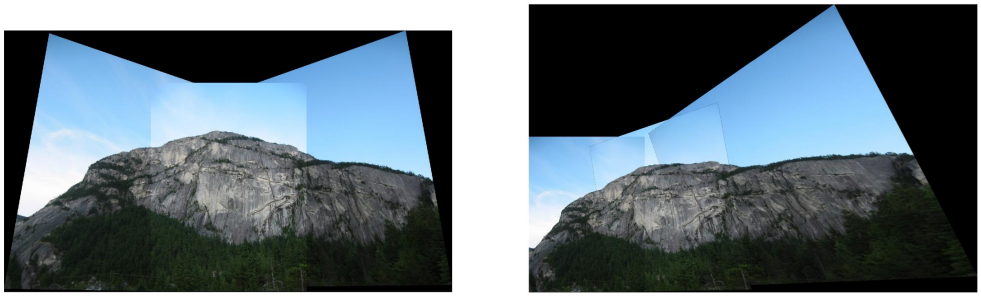


Figure 1: 5 image sets used for testing



(a) Panorama with 3 images projected to the center image

(b) Panorama with 3 images projected to the first image

Figure 2: Picking a good center makes the panorama much better

(a) Cylindrical projection with $f = 500$

(b) Cylindrical projection with $f = 400$

Figure 3: Examples of cylindrical projection

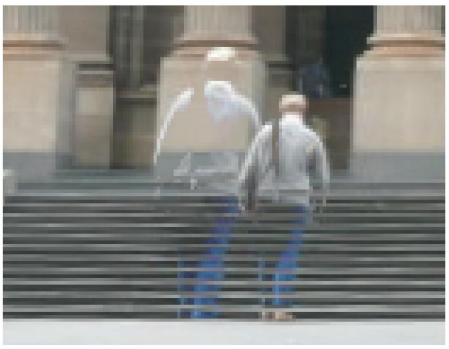


Figure 4: The problem with naive merging

Figure 5: Result from set 1

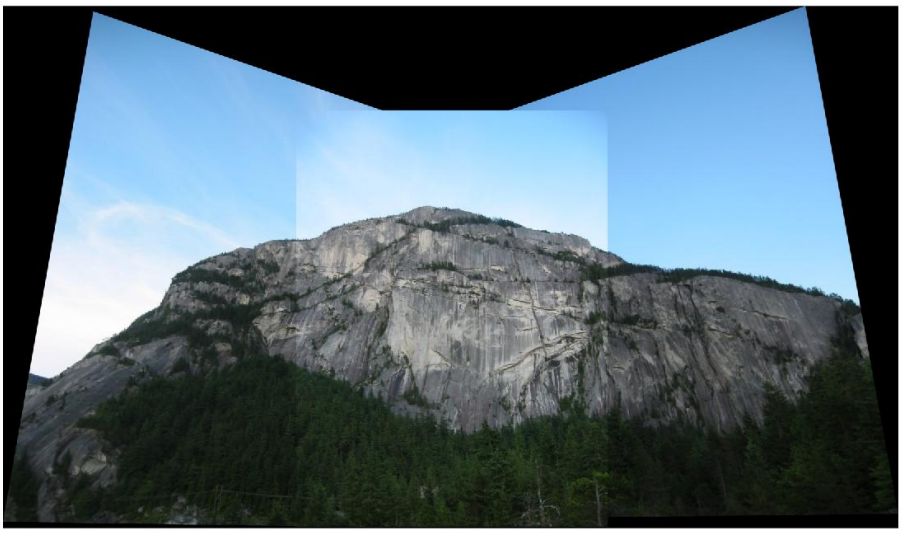


Figure 6: Result from set 2

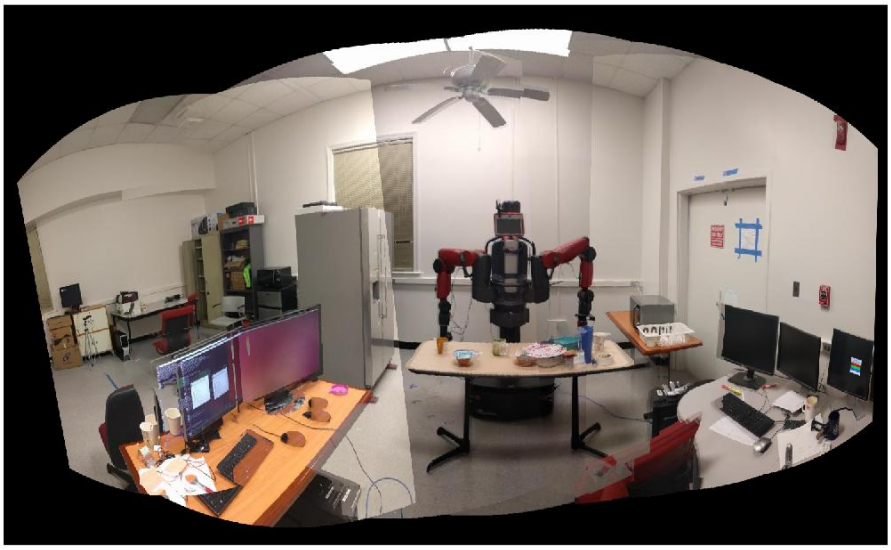


Figure 7: Result from set 3

Figure 8: Result from custom set 1

Figure 9: Result from custom set 2