CS6640 A2

Cade Parkison

9/15/2018

<u>Assignment 2</u>

Note: For all the code submitted in this assignment, the testing script can be used to test the functions created.

Problem 1: Texture Analysis

For the study of texture analysis, I took some frames from the video of a moving car. Using the Laws 7x7 texture filters, the function CS6640_Laws() creates 10 energy maps, which is an image where each pixel is associated with a vector of 10 texture attributes. I passed these "energy maps" along to Matlab's K-means clustering algorithm. In order to study the K-means algorithm, I tried running k-means with different k values. The k input to the Matlab function denotes the k clusters that the observations will be split into. In my experiments, if this number was too low, then the algorithm clustered many different looking parts of the image into one cluster. If this number was too high, then there were so many clusters that the data became meaningless. There was a sweet spot where the clustering split the pixel regions into meaningful data, where all pixels in one region roughly corresponded to semantic parts of the image. For example, with a k value of 4, one cluster accurately separates the white sidewalk from the black asphalt of the parking lot. This can be seen below in Figures 2 and 3. By combining different k-means clusters, I was able to get pixel regions to match very closely to specific textures in the image. For example, in Figure 6, I combined clusters 1 and 4. This clustered almost all of the black asphalt, while also clustering some of the darker parts of the tree leaves. Further image processing on that image could be done to separate these different regions. Since the asphalt region is fairly closed with gaps between it and the trees, various techniques could be used to separate these such as edge detection, or morphological operators like opening and closing.

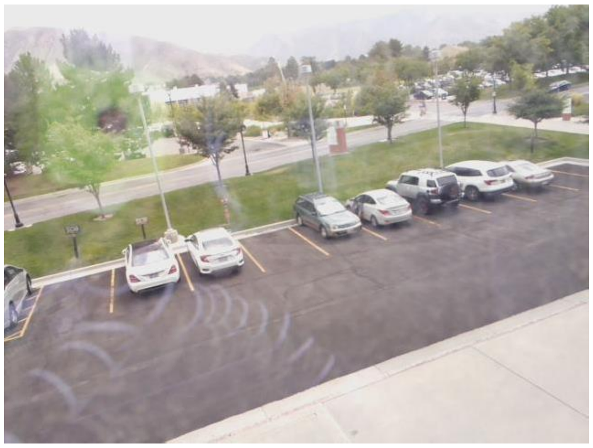

Figure 1: Original Image

K = 4

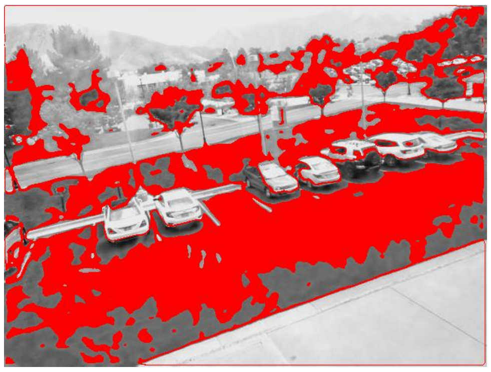

Figure 2: K-means Cluster 1

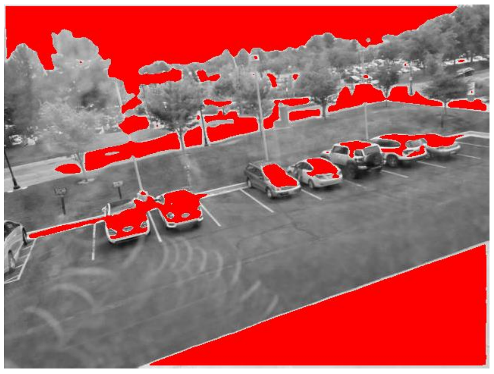

Figure 3: K-means Cluster 2

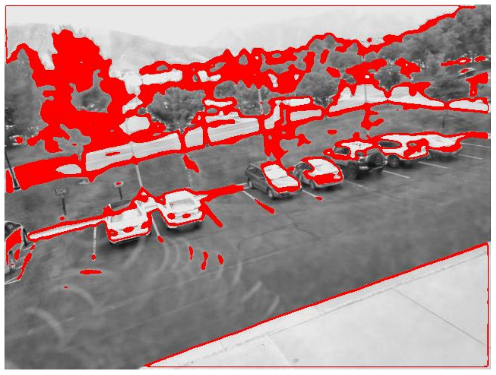

Figure 4: K-means Cluster 3

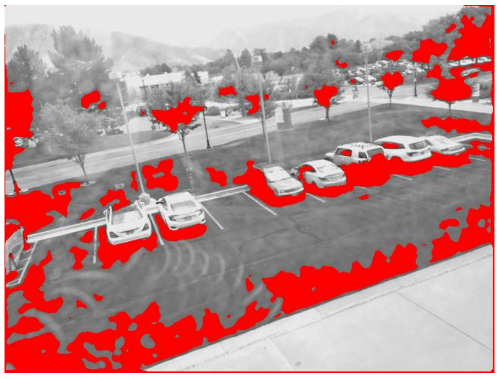

Figure 5: K-means Cluster 4

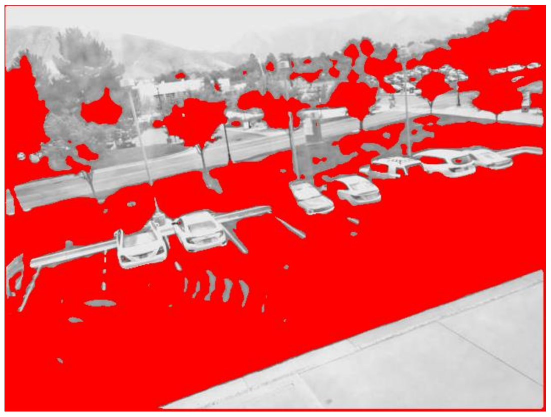

Figure 6: K-means Clusters 1 and 4 combined

To show the effects of the parameter k, below is clustering done with the same texture energy map. This time, a k-value of 9 was used for k-means clustering. Shown are only a few of the clusters, but an important thing to note is that this separated some visually similar regions into separate clusters. This could be valuable to detect smaller texture changes in an image, but it would require more post-processing to extract meaningful data.

K=9

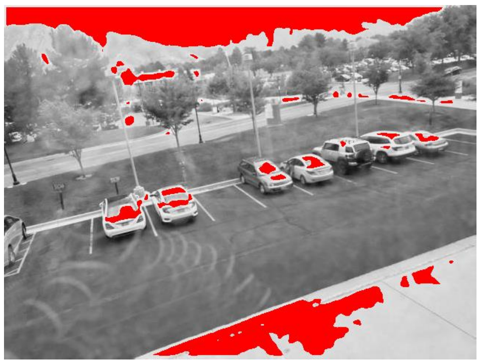

Figure 7: Cluster 1

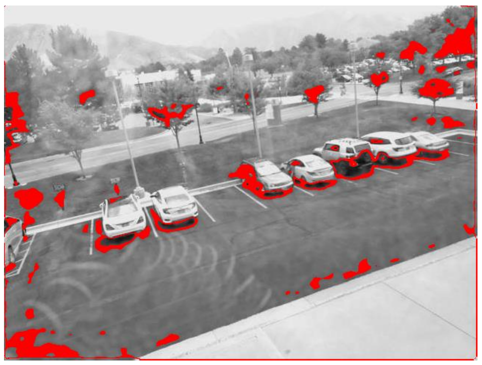

Figure 8: Cluster 2

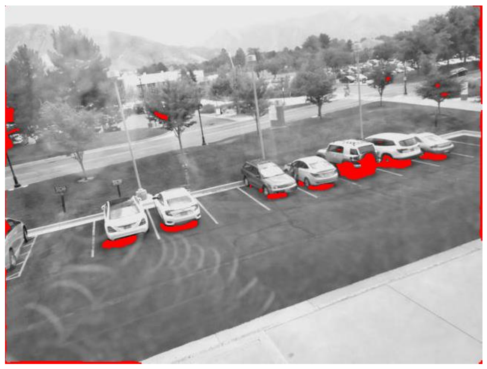

Figure 9: Cluster 3

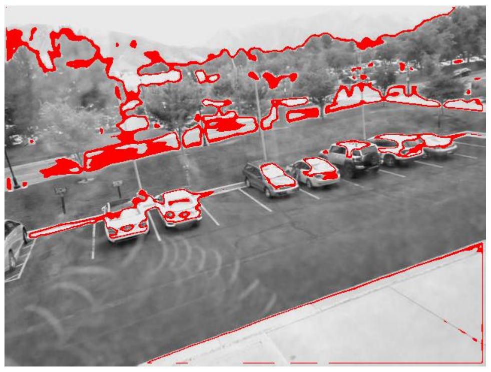

Figure 10: Cluster 7

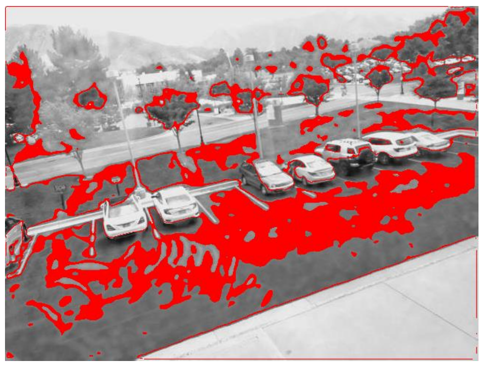

Figure 11: Cluster 9

Problem 2: Image Registration

To study image registration, I experimented with a few sets of images. First, to test the transformations and registration, I used a single image for both the moving and the fixed image. By selecting identical points in both images and passing these to the Register function, I expected to get an Identity matrix for the transformation matrix. Below are these results.

Figure 12: Original Image

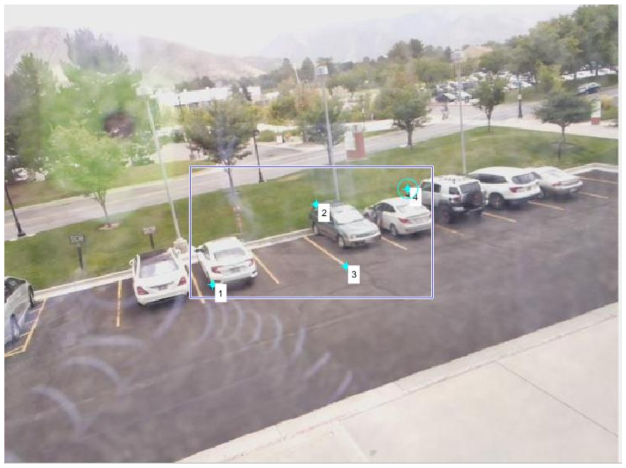

Figure 13: Control Points

The cpts, found using the cpselect() function, were the following:

Cpts = [218 295; 218 295; 325 211; 325 211; 356 276; 356 276; 432 195; 432 195]

Using the above corresponding points does indeed give the identity matrix. In the affine case, the following A matrix was returned:

```
A =

   [1.0000   -0.0000   -0.0000;

  -0.0000    1.0000    0.0000;

        0         0    1.0000]
```

Ignoring the negatives on the zeros, this is the Identity matrix and when applied to the original coordinates will produce transformed coordinates that are identical. This confirms the affine transformation in the register function is working correctly.

Next, I perturbed the original corresponding points by slightly shifting the points in the images. I used the following corresponding points for this:

Cpts = [218, 295; 220 293; 325 211; 324 210; 358 276; 356 274; 431 196; 432 195];

These sets of points used in the register function produced the following non-identity A matrix as expected:

A = [0.9866   -0.0149   8.1132; -0.0020   0.9856   2.6724; 0   0   1.0000]

This is to be expected because the coordinates in cpts do not correspond to the same location, thus a transformation is needed to get from one set to the other.

In the quadratic transform case to the above images, the following q vector was returned:

```
q =

     [ 0

   1.0000

        0

  -0.0000

  -0.0000

   0.0000

        0

   0.7130

        0

  -0.0007

  -0.0008

   0.0026]
```

Unfortunately, I am not happy with this result, as it should have returned something more similar to an Identity matrix. I believe this is due to the way the quadratic functions were set up. For some reason, the transformed coordinates were outside the bounds of the image. If I had more time, I would have liked to explore this area more.

One current problem with my registration function is that it does not handle interpolation between pixel values in the best way. If the transformed pixel location does not lie on a pixel in the other image, I round the coordinates to the nearest pixel value. A better method to use in the future would be some form of linear interpolation which looks at the surrounding pixel values to come up with the best estimate for the pixel value in the new location. Another issue that I ran into, was when the transformed pixel coordinate was outside the bounds of the image. This seems to happen when the two images require a large transformation between pixels. In that case, I limited the pixel coordinates to the maximum and minimum height and width of the image.

Next, I took two frames from the video of a moving car and attempted to use the CS6640_register() function to transform one image to the other. Finally, I tried using image registration on two aerial photos of a location in a city. These photos were taken from different camera locations and orientations, but the same buildings are in frame in each shot. The attempt here was to transform the second image in order to match the orientation of the buildings in the first image. This turned out to be a much better use case than using this algorithm on the pictures of the moving car. With the moving car images, the algorithm greatly distorted the region around the car when registering it with the first image.

In both cases, the Harris function was used to compute the Harris operator at each pixel. The Harris operator is a score for the “cornerness” of an image. In other words, it is a metric that determines how much a pixel looks like a corner because of it’s surrounding pixels. This is done by using the Hessian to look for sharp changes in pixel intensities in two perpendicular directions at each pixel. After computing this value for every pixel in an image, I took a threshold to only return the most corner-like pixels. Due to the invariance of the Hessian to rotation and orientation, the same location in two different images should have similar Harris values. The eigenvectors will point in different directions, but their values should be similar.

Case 1

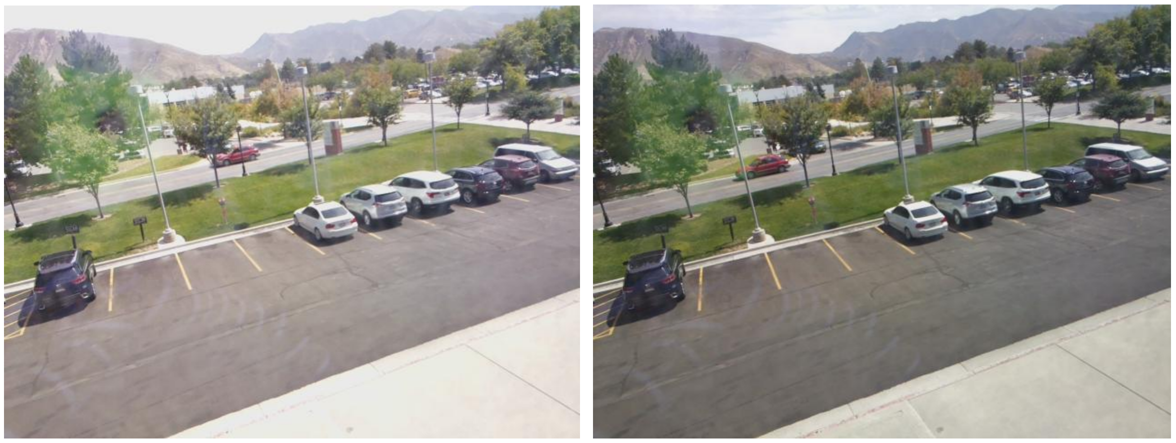

Figure 14: Case 1 Images

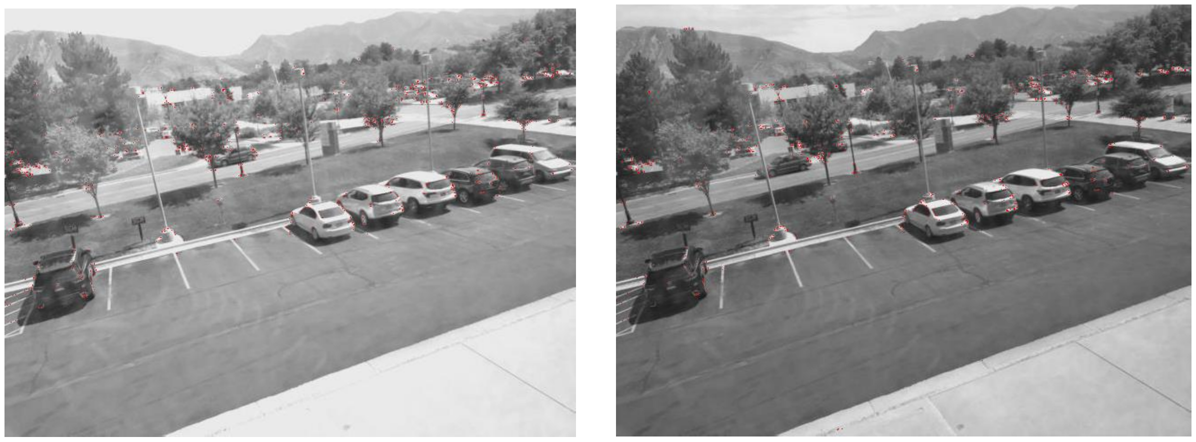

Figure 15: Harris operator with R>0.005 threshold

Above images show the Harris operator in combination with the original images with a threshold applied of R>0.05. As can be seen, most of the sharp corner of the cars, as well as the discontinuities between objects like the trees are highlighted. Interestingly, there are discrepancies between the two images even in the stationary points of the images. For these corresponding points to be useful in this application, you would want to have an set of points that all correspond to the same physical locations. This could be done by manually filtering the Harris operators that are above the threshold.

Case 2

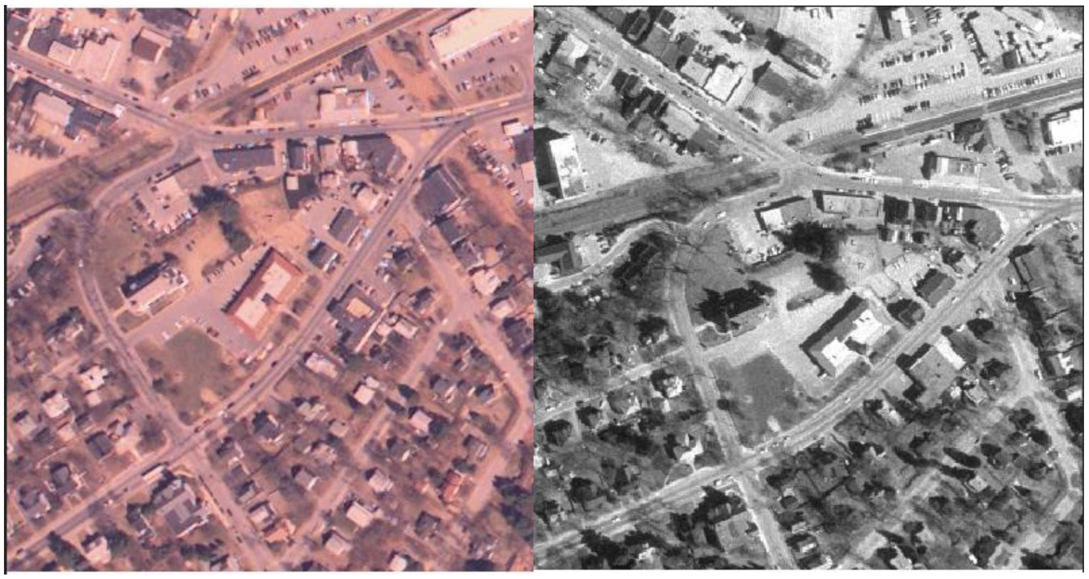

Figure 16: Case 2 images

After converting the above left image to grayscale, I ran the Harris function on both images to produce the following images showing the pixels that are most corner-like. To remove some noise and only study the most corner-like pixels, I did a threshold at a value of R= 0.01. As can be seen below, this marked similar building and street corners in both images, but also marked many locations that do not correspond to locations in the images. I believe this is due to the different lighting conditions and camera positions and orientations. Without manually selecting corners, this data will make the registration problem much more difficult. An application where the Harris operator would be better suited for image registration would be when the two images are more similar than these two.

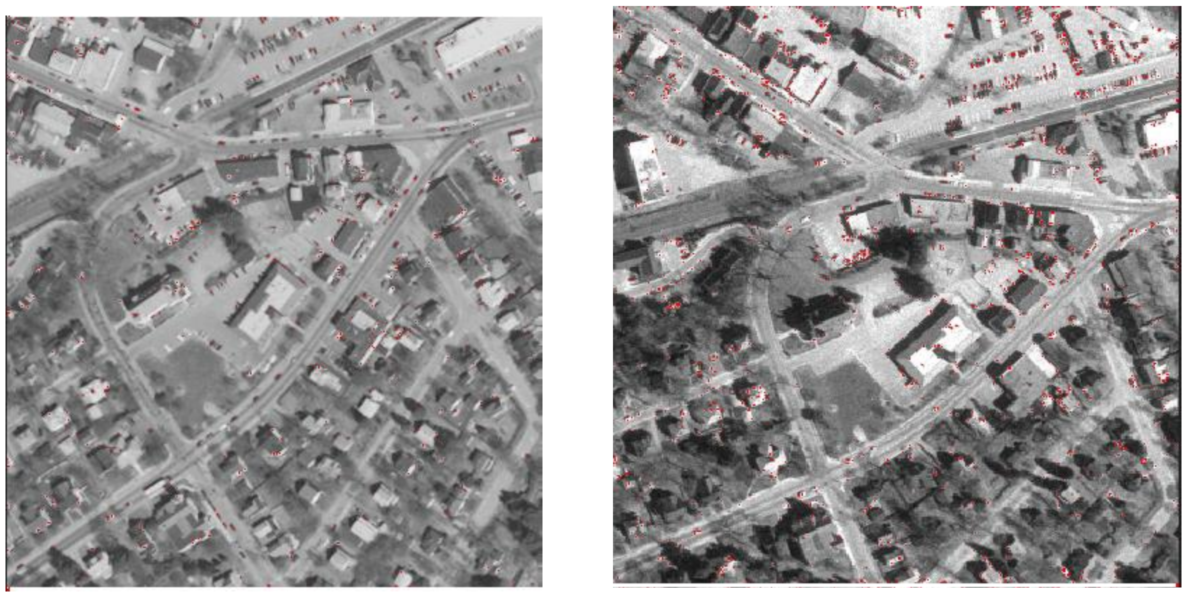

Figure 17: Harris points combo with original images

Next, to better test the Image Registration function, I manually picked pairs of corresponding control points in both images. To do this, I used Matlab's Control Point Selection tool, cpselect(). These are shown below.

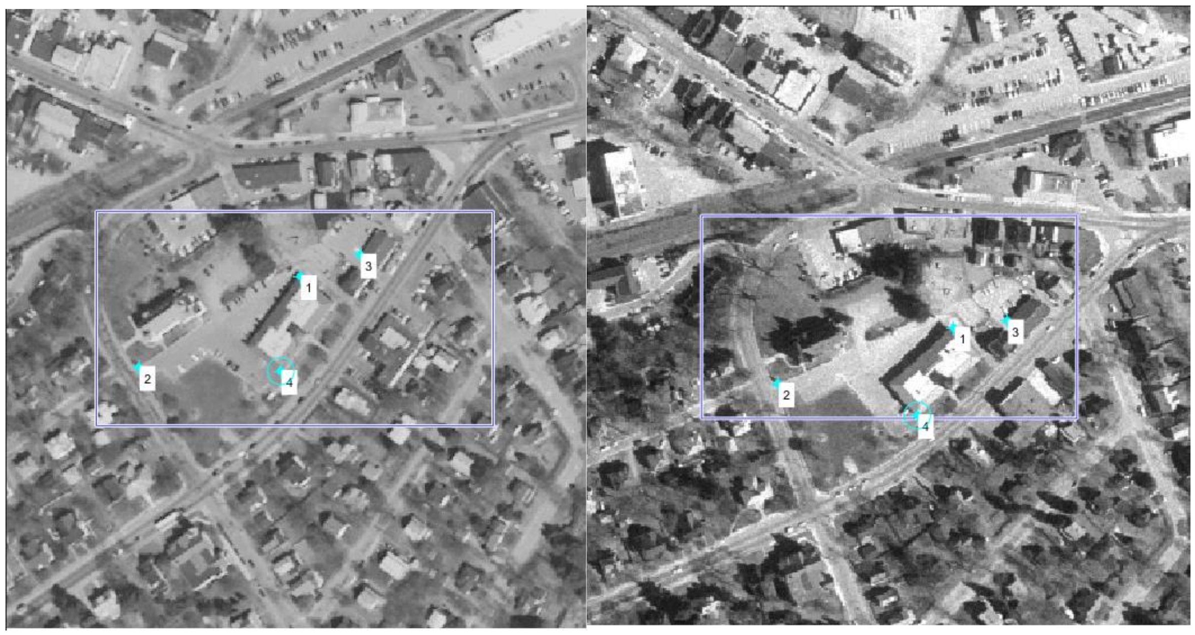

Figure 18: corresponding points using cpselect()

The corresponding points are the following, where each two rows represents corresponding x-y pairs.

```
Cpts =  [ 189.6250  172.6250;
           247.3750  217.3750;
           86.8750  230.3750;
          129.1250  254.8750;
          227.6250  158.8750;
          283.3750  212.3750;
          177.1250  232.8750;
          222.8750  276.1250]
```

Using the pixel values for the above point pairs along with the left image, I can run the CS6640_register function to produce a registered image from the right image. In theory, this should align the right image with the left, where the overlapping images would look identical.

Problem 3: Temporal Differential Operator

Out of the three problems in this assignment, I found this one to be the most interesting and also gave me the best results. The problem of motion tracking is a very useful problem to solve in robotics and many other applications. The temporal difference operator works by first extracting the stationary background from a video with a moving object, in this case the car moving across the field of view. Once the background image is extracted, the algorithm loops over every frame of the video and finds the difference image by subtracting the background frame from the current frame. This can be seen below in the bottom left image. Next, I took a threshold of that image to remove some noise that appeared along the road. This produced the binary image seen bellow on the bottom right.

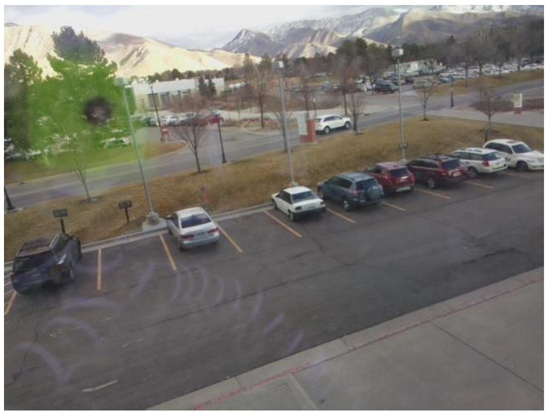

Figure 19: Original Test Image

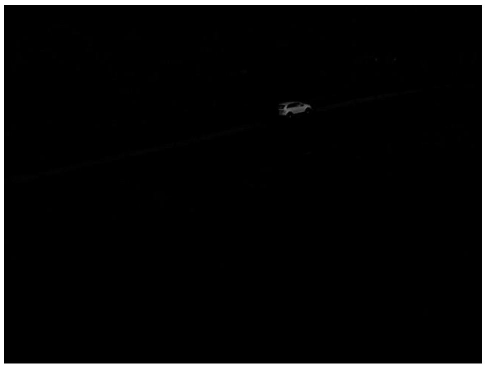

Figure 20: Difference Image

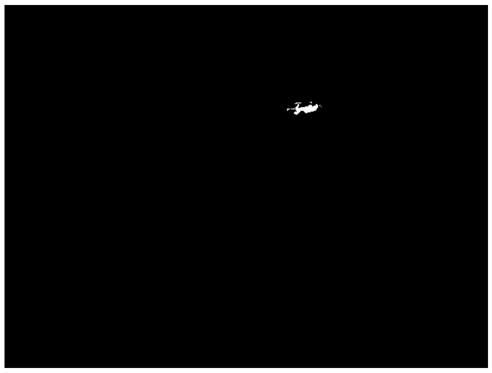

Figure 21: Thresholded difference image

This binary image above has white pixels where it believes the moving object is, and black pixels everywhere else. Then, the centroid of these white pixels is calculated by taking the mean of their x and y coordinates separately. These new coordinates very closely represent the centroid of the moving object. Once this is done for every image in the video, a track of the centroid location is created and used to overlay a marker on the original video.

Below are some images from my final video that display a red + marker on the centroid of the moving object. These images were made using a custom function titled displayTrack. This function creates a movie in the same manner as the track() function, only this time it uses the original video instead of the mask. As you can see, the estimated centroid is very accurate, although it’s exact location on the car varies by a few pixels. This is due to the way I calculated the centroid. When the car is partially obscured by objects, the apparent size of the car is smaller from the background subtraction. This shifts the mean of the pixel locations of the moving object slightly. In the first image below, where the car is just barely in frame, the calculated centroid is slightly in front of the car. I believe this is due to the thresholding I applied to the difference images as well.

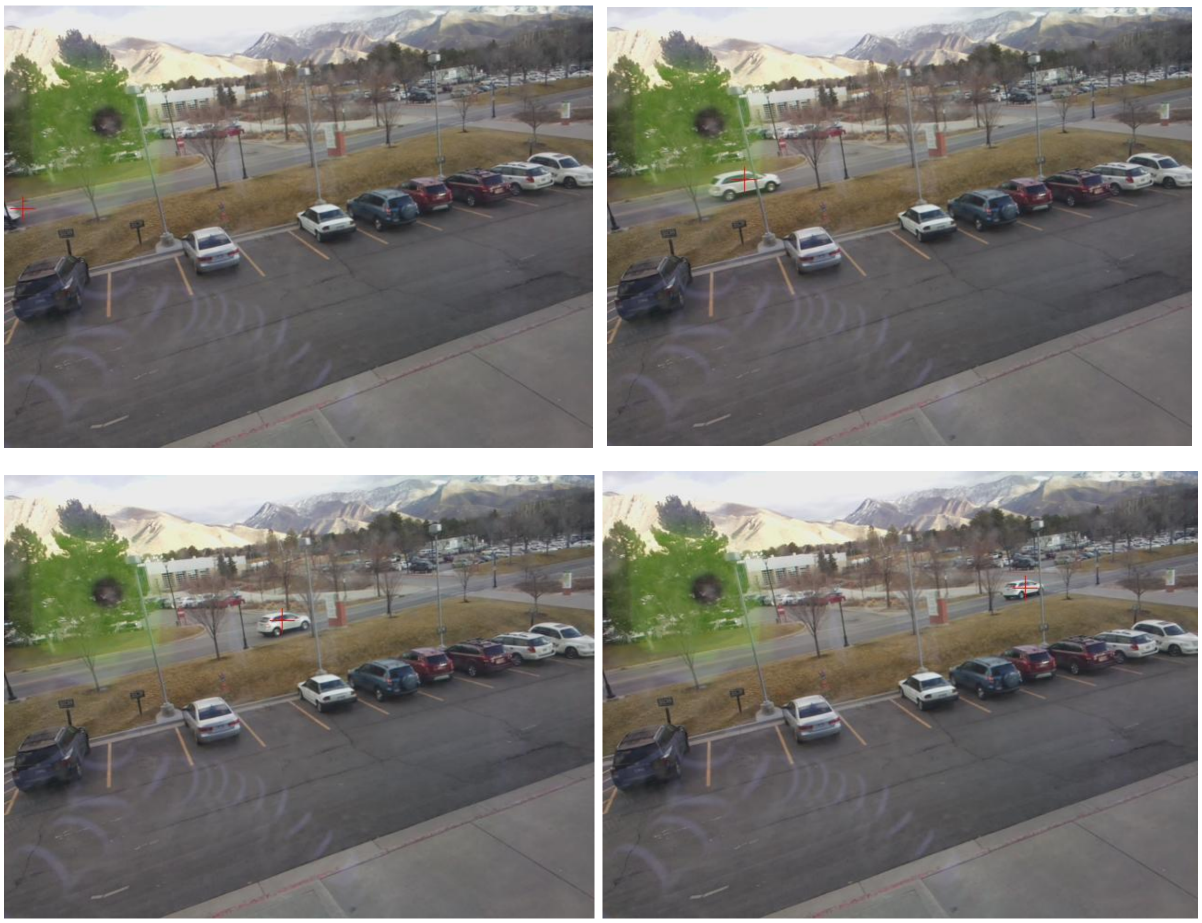

Figure 22: Select frames from video with centroid overlay

The output of the track() function on the video discussed above was a 209x2 array giving the pixel coordinates of the centroid of the moving object. Below is the first few lines of that array:

```
tr =

   55.8108  211.2973
    20.7273  221.1818
   20.9158  220.2263
   26.1759  219.1065
   32.2321  217.4732
   35.2134  215.7945
   35.8447  213.8350
   37.3027  212.3887
   38.8779  211.0433
   41.8593  209.7099
```

As you can see, these x-y pixel values correspond to the line that the road makes across the camera sensor. Interestingly, the x-coordinate jumps from 55 to 20 before linearly increasing in subsequent frames. This is due to the calculated centroid of the first frame before the car fully emerges into view. This is also noticeable in the output video, where the red + will jump backwards in the beginning.