Lewis Arnsten

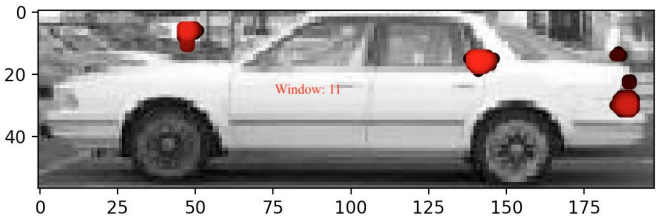
Please note: My feature extraction produces a weird error that I could only fix with the following change to visualize.py, plot_matches.

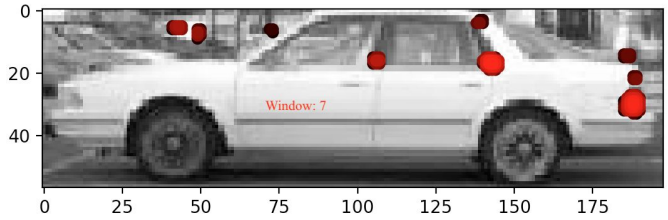
```
def plot_matches(image0, image1, xs0, ys0, xs1, ys1, matches, scores, th):
   assert image0.ndim == 2, 'image should be grayscale'
   assert image1.ndim == 2, 'image should be grayscale'
   # combine images
   #print(matches)
   sy0, sx0 = image0.shape
   sy1, sx1 = image1.shape
   sy = sy0 + sy1
   sx = max(sx0, sx1)
   image = np.zeros((sy, sx))
   image[0:sy0,0:sx0]       = image0;
   image[sy0:sy0+sy1,0:sx1] = image1;
   # get coordinates of matches
   xm = []
   ym = []
   for i in matches:
    xm.append(xs1[i])
    ym.append(ys1[i])
   #xm = xs1[matches]
   #ym = ys1[matches]
   # draw correspondence
   plt.figure()
   plt.imshow(image, cmap='gray')
   X = np.zeros((2))
   Y = np.zeros((2))
   N = matches.size
   for n in range(N):
      if (scores[n] > th):
         X[0] = xs0[n]
         X[1] = xm[n]
         Y[0] = ys0[n]
         Y[1] = ym[n] + sy0
         plt.plot(X,Y,'b-')
         plt.plot(X[0],Y[0],'ro')
         plt.plot(X[1],Y[1],'ro')
```

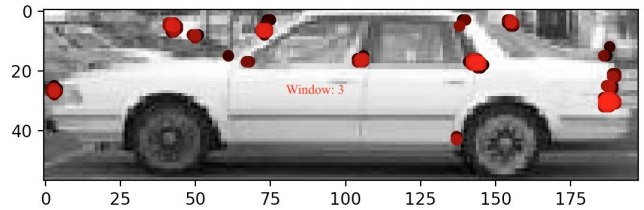
Interest point operator:

For the find_interest_points function, I chose to use the Harris Corner Detector as my operator. First, I used a gaussian to smooth out noise in the image. Then I implemented the Harris Corner Detector and visualized the results with window sizes 3, 7, 11, and 19.

Here are the results for window sizes 3, 7, and 11 to conserve space: (only showing top 100 or so points)


Window: 11
0
20
40
0
25
50
75
100
125
150
175


Window: 7
0
20
40
0
25
50
75
100
125
150
175


Window: 3
0
20
40
0
25
50
75
100
125
150
175

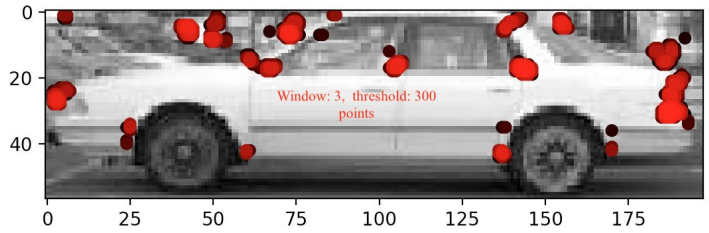

Window: 3
0
25
50
75
100
125
150
175
0
50
100
150
200
250

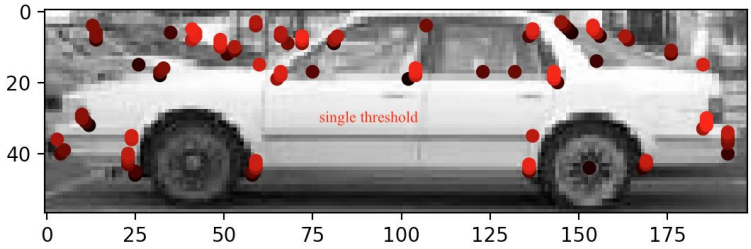
Given these results I chose to use a window size 3 as it finds more interest points, especially those on the tires of the car, which I anticipated would be easier to match between images. Thus, a window size of 3 seemed most likely to find a large number of reliable matches between different views.

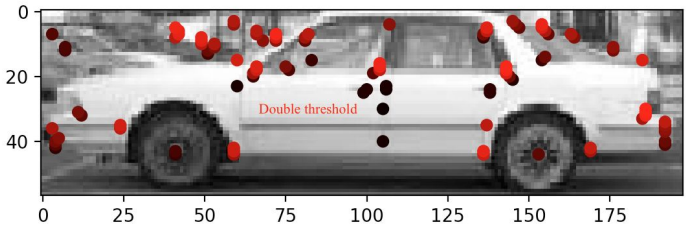
However, as I visualized these window sizes I also noticed that many of the pixels that were dubbed interest points came in patches. Thus thresholding (which I did using a sorted list of interest points) would not return very diverse locations. Additionally, after non max suppression I was left with very few interest points (this was less of a problem with window size 3, but nonetheless). Therefore, I decided to use a threshold significantly higher than the max_points, then do non max suppression and remove all the points that were suppressed from the list of possible interest points. This gave me a list of points from which harris corner detection would

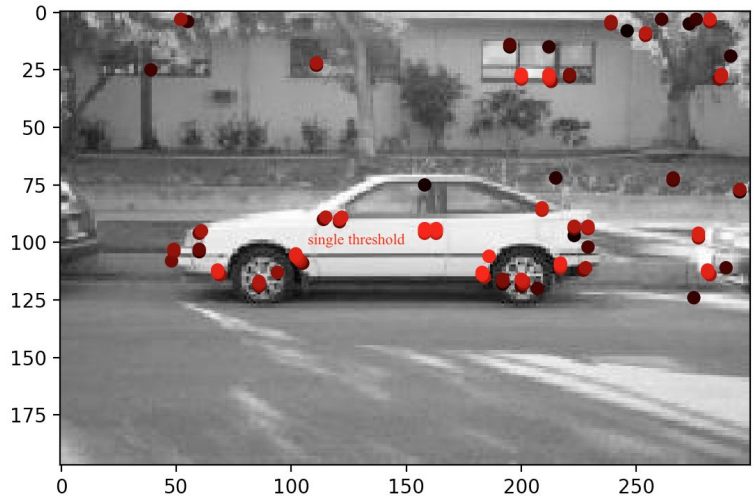
find less patchy interest points. I then did thresholding and non max suppression once again, and returned the top x interest points of highest value where x is 0.9 * max_points.

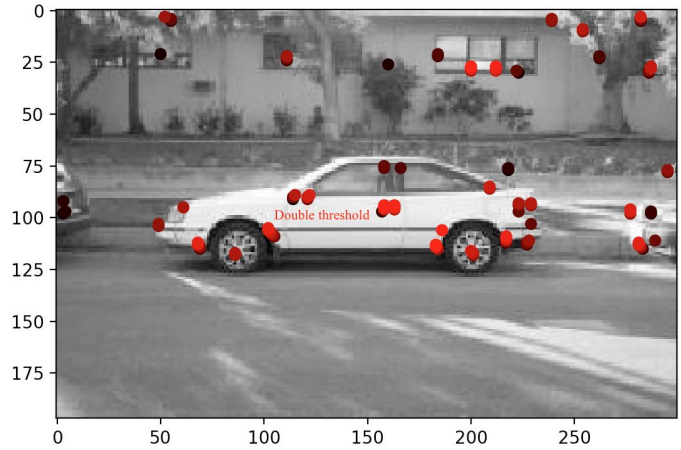
Here are the results of single thresholding versus my double thresholding (as well as images with a flat threshold of 300 with no non max suppression for reference):









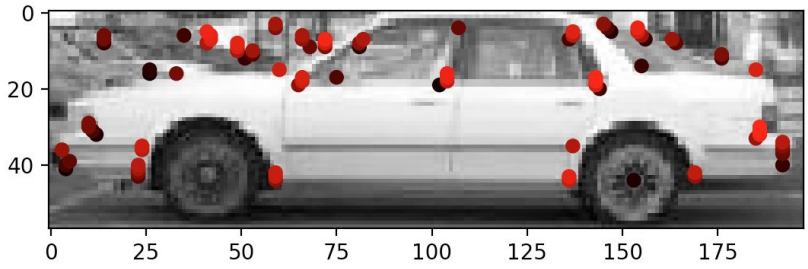


The multiple thresholds that I used ended up having very little of an effect, however I thought it was an interesting idea and decided to include it.

In total I used three thresholds, 4*max_points, 4*max_points, and 0.9*max_points, all of which were decided through experimentation.

For non max suppression, I took the suggested route and compared interest points to the maximum value over a certain region. For non max suppression I found 7 to be the optimal window size, as any window smaller would leave points that were clearly not edges, and any window larger would suppress too many points.
.
Additionally, I chose to use an alpha of 0.06, which was decided upon through experimentation.

Here is a visualization of the final results of my interest point operator:



Feature descriptor:

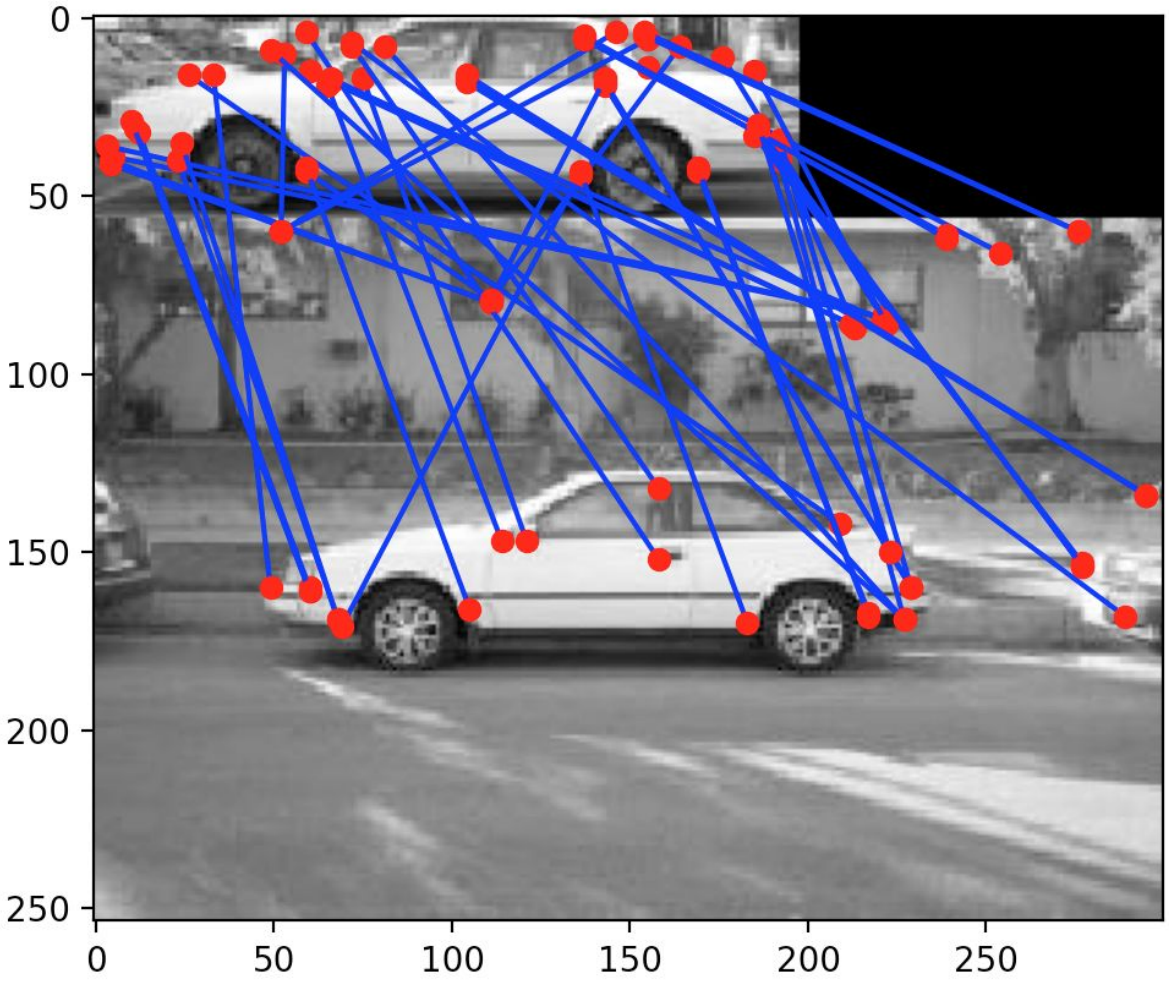
As suggested, I implemented a SIFT-like feature descriptor by binning orientation energy in spatial cells surrounding an interest point. To begin, I did canny edge detection without non max suppression. I decided to set a cell width of 10, which meant looking at windows of size 30. This was decided upon experimentally after testing the results of window's size 12, 18, 24, 30, and

36. By using a larger window size for feature descriptors I made it less likely that two uncorrelated interest points would have similar features. Finally, I normalized and returned the feature array.

Feature matching:

I implemented feature matching using the nearest neighbor distance ratio discussed in class. I then sorted by this ratio to get matches in order of confidence, and returned the first value--the match with the lowest distance. Each of these matches was given a score equal to the nearest neighbor distance ratio. Finally, I checked for overlapping between matches (two interest points in one image match with the same interest point in another). If two matches overlapped, then I checked their scores and set the score of the match with the lower score to zero.

Here is the result of my feature descriptor and feature matching:



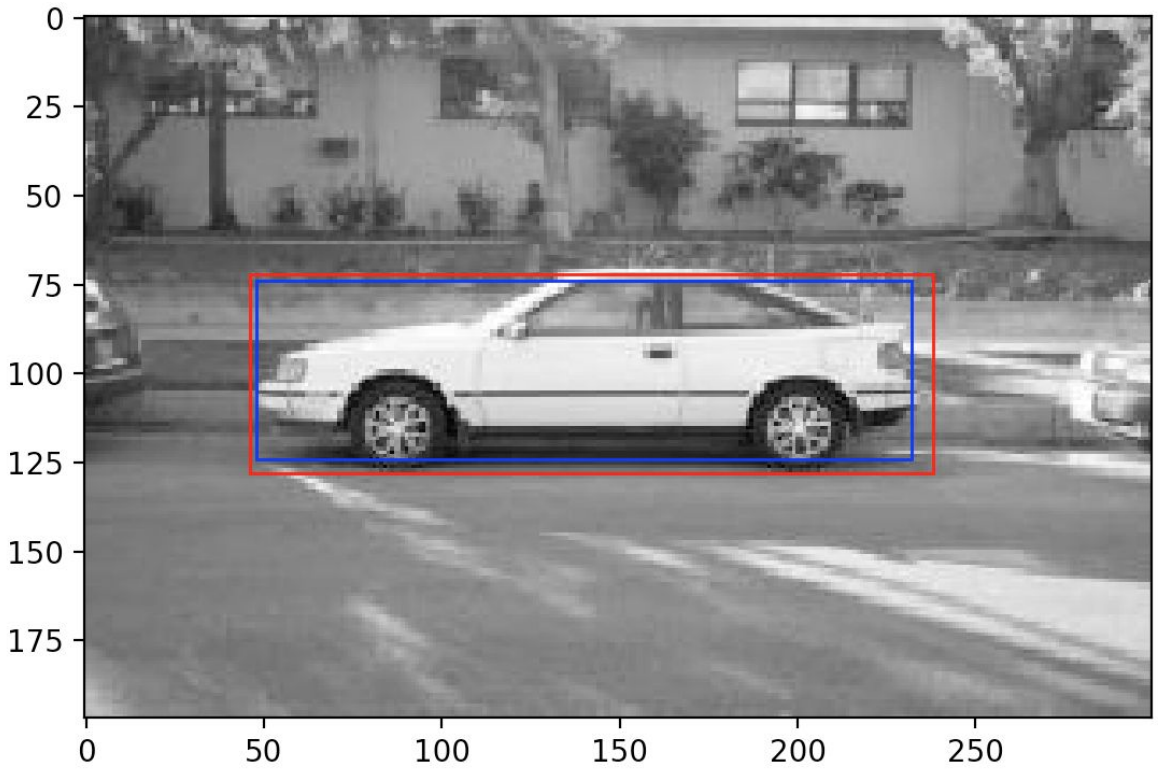
Hough transform:

My implementation of hough_votes followed the basic implementation steps discussed in class: I created two lists of parameter values (seperate for x and y transformations), let each point vote for a certain degree of transformation, and then created two weighted histograms to return the respective tx and ty values that got the most votes (taking into account score). Thus, votes was returned as a list containing the maximum of each of these histograms.

Object detection:

To implement object detection I combined all the steps discussed thus far. I found interest points and extracted features from the template images and test image. I used the masks to reduce background noise, and then used the maximum and minimum values of the x and y lists of interest points to find x_min, x_max, y_min, and y_max. I then matched features between each of the template images and the test image and used hough_votes to find possible transformations. After testing all template images, I chose the x and y transformations that received the most x and y votes across all images. My multi-scale strategy was to resize images and repeat my single-scale detection procedure over multiple image scales.

Here are my object detection results:



Before scaling was implemented:

```
class data_car, average IOU 0.7295784192102966, total running time 166.814837455
7495s
```

```
class data_cup, average IOU 0.3191338660855241, total running time 412.802744388
5803s
```

After scaling was implemented:

```
class data_car, average IOU 0.7529962339394539, total running time 448.585863351
8219s
class data_cup, average IOU 0.29358671846404416, total running time 1476.3048546
31424s
```

In retrospect, I believe difference of gaussian would have performed better than Harris corner detection.