HOMEWORK 2 WRITEUP

Felipe Alamos

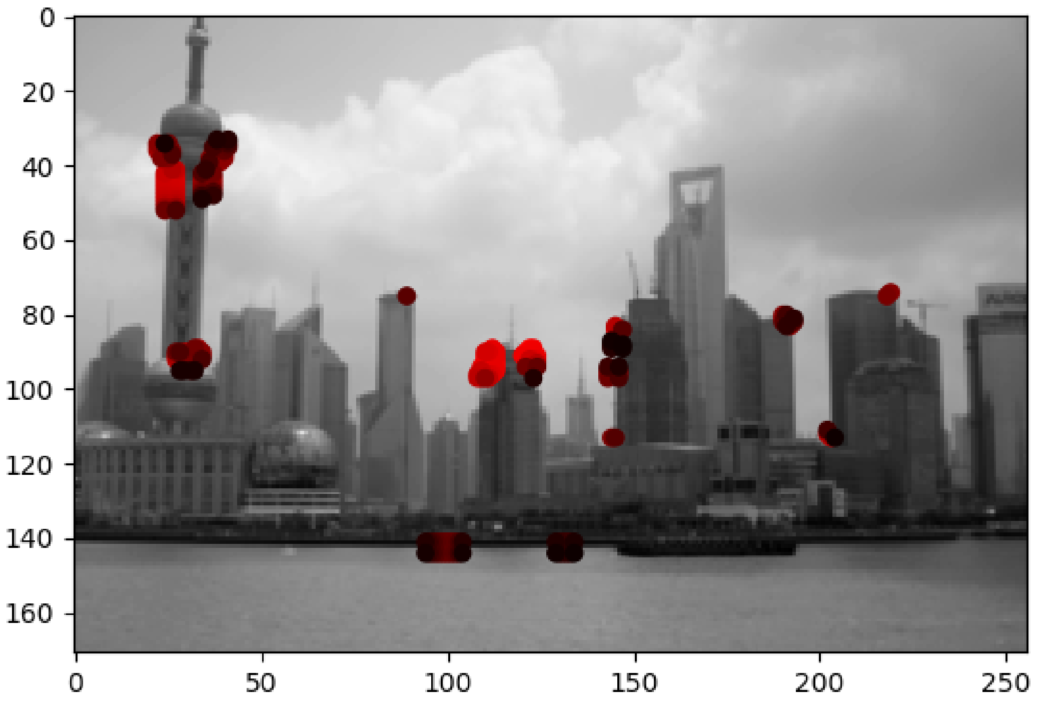
02/08/2020

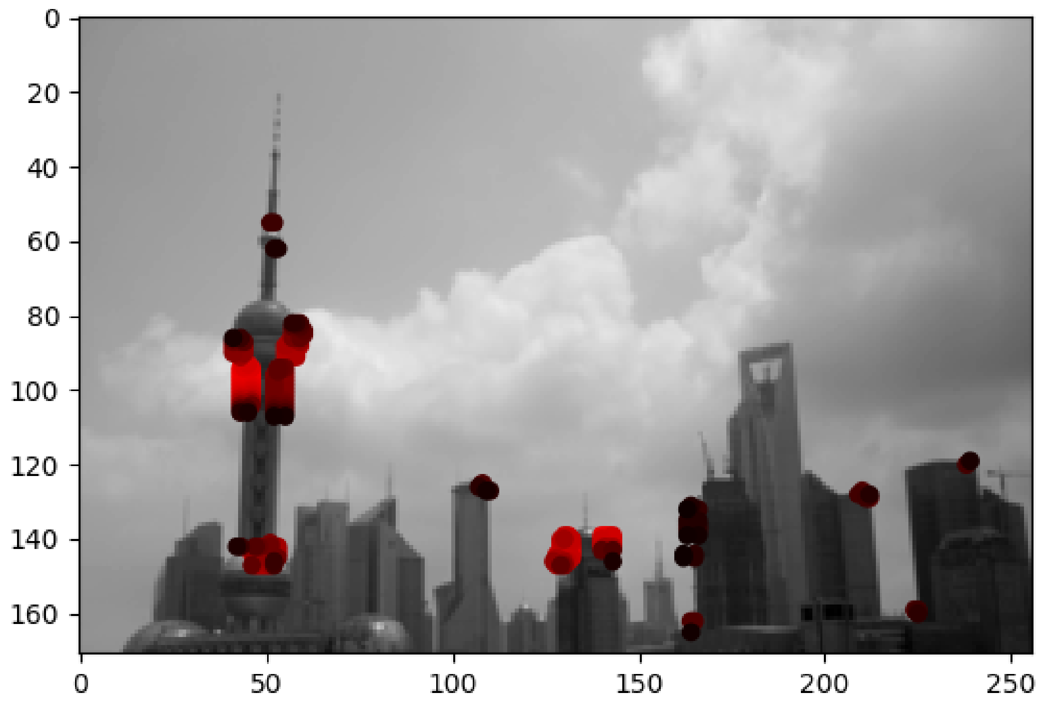
INTEREST POINT OPERATOR

I used the Harris corner detector, which is a well known algorithm for detecting with interest points.

To compute the matrix of second derivatives needed in the Harris algorithm, I convolve the image with horizontal and vertical derivatives of a Gaussian (sigma = 1). This is a modern variant of the algorithm introduced by Schmid, Mohr, and Bauckhage 2000.

Once our matrix of corners R is built, I use nonmaxsupression and a gaussian filter to eliminate corner detections that are not strong enough compared to neighbors.





FEATURE DESCRIPTOR

My feature descriptor is a histogram of gradients directions, for the surrounding areas (windows) of each feature descriptor.

For every point of interest, I looked at the 3x3 spacial grid (9 windows) around it, each of the windows of width 3. For each window, I created a length 8 histogram, one for each of pi/4 theta segments. Hence, in each of the bins of the histogram, I counted how many gradients in the window fall under the given theta segment. This way, I have a length 8 feature descriptor vector, for each of the 9 windows around the interest point. Concatenating these histograms, I end with a length 72 feature descriptor.

```
 ____ ____ ____
|    |    |    |
|    |    |    |
|____|____|____|
|    |    |    |
|    | () |    |
|____|____|____|
|    |    |    |
|    |    |    |
|____|____|____|

     |----|
 width = 3 px
```

I also used mirror padding in our image, so as to take into account that some points could be in borders or corners of our images, and would have been unfeasible to capture the 9 windows around it.

FEATURE MATCHING

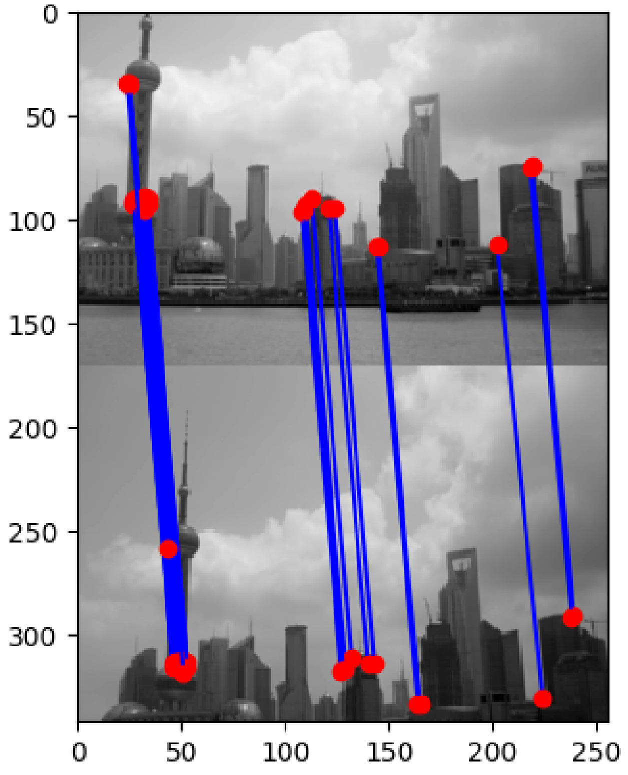
Navie

For naive mode, my approach is, for each interest point in feats0, loop over all in interest points in feats1, calculating the distance between the interest points. Then, obtain the two points in image 1 that have the closest distance to each interest point in feats0. I use euclidean distance between feature points, and results behave properly.

lsh

First of all, I created an *LSH* class that implements all necessary methods to work with lsh (*compute_hash*, *generate_hash_table*, *find_closest_hash*, *search_hash_table*, *search_feature_nn*). Instances of LSH have a hash_table dictionary (which contains the buckets with the features), and a seed used in computing the hash codes.

The amount of hash_tables used can me modified in line 480 of method *match_features*. This enables us to increase robustness. I first instantiate the hash_tables desired, with different seeds, saving in the buckets the features from feats1. Then, for each feature point in feats0, I get all its closest points as the union of closest points obtained from the different hash_tables. Finally, for that set of candidate points, I calculate the distance of our feature point with each of them and capture the two closest ones.



Efficiency comparison

Feature matching: After a series of iterations, testing performance (sim scores and runtime), I find that the ideal setting is to use 2 hash tables, each of them with hash_codes of length 20.

Under that setting, the results are as follows:

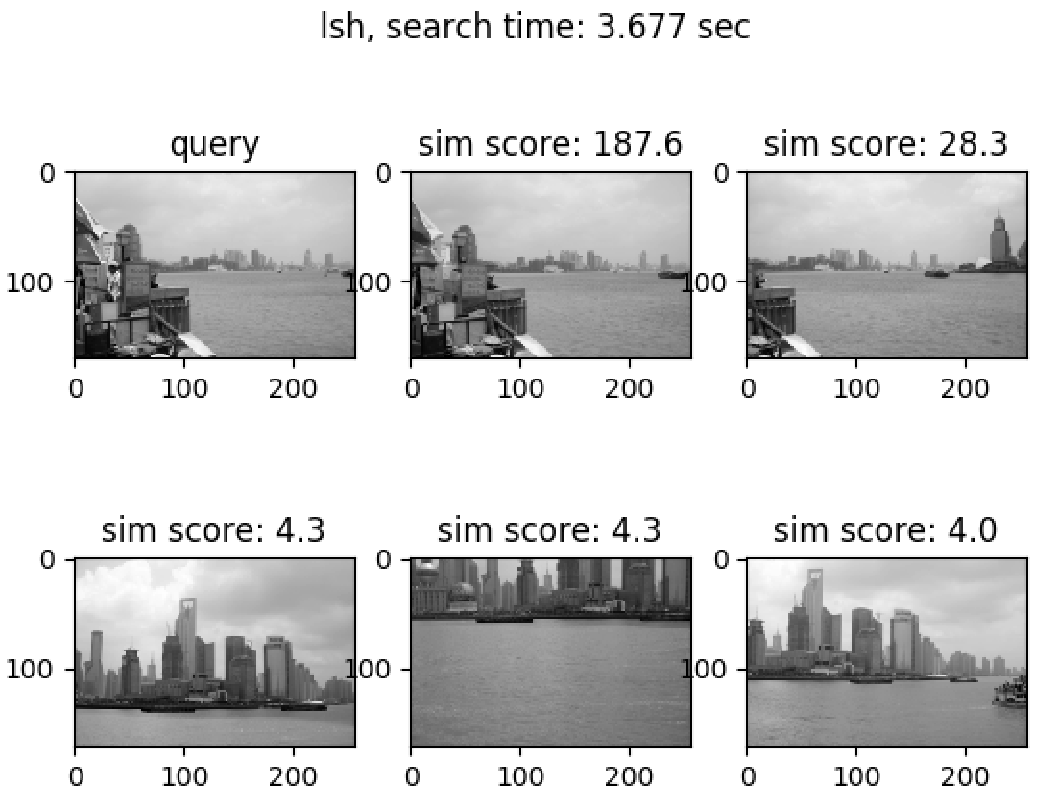
naive: Search time: 5.347. Sim scores: [180.4, 30.4, 5.5, 3.3, 3.3] lsh with 2 hash tables and hash codes length 30: Runtime: 3.677. Sim scores: [187.6, 28.3, 4.3, 4.3. 4.0]

For an lsh setting with 1 hash table, and hash codes of 20, runtime decreases as expected, but simscores also decrease: lsh with 1 hash tables and hash codes length 30: Runtime: 1.753. Sim scores [122.1, 23.0, 7.5, 6.5, 6.3]

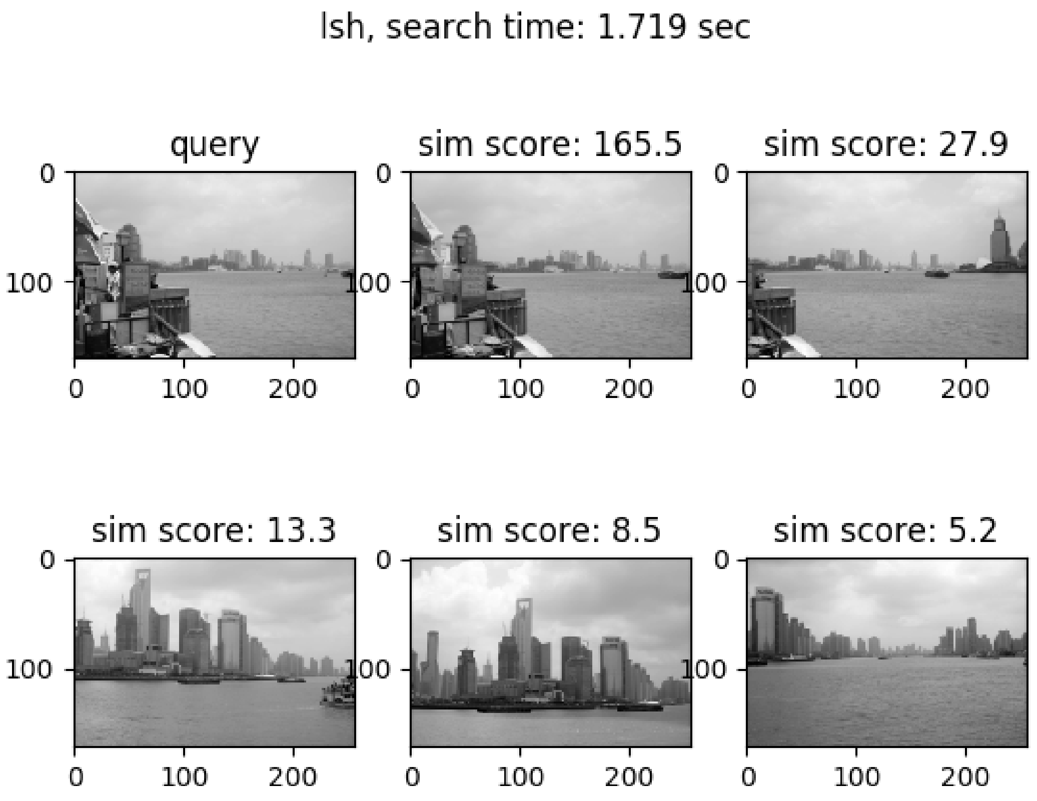
Lastly, if we vary the length of hash codes, we know both runtime and sim scores should be will be affected. The longer the hashcodes, the faster the process (cause there will be less close neighbors to compare with), but sim scores should decrease. For example:

lsh with 1 hash tables and hash codes length 20: Runtime: 1.719. Simscores: [165.5 27.9 13.3 8.5 5.2] lsh with 1 hash tables and hash codes length 50: Runtime: 1.250. Simscores: [125.8 29.5 10.6 7.0 6.5]

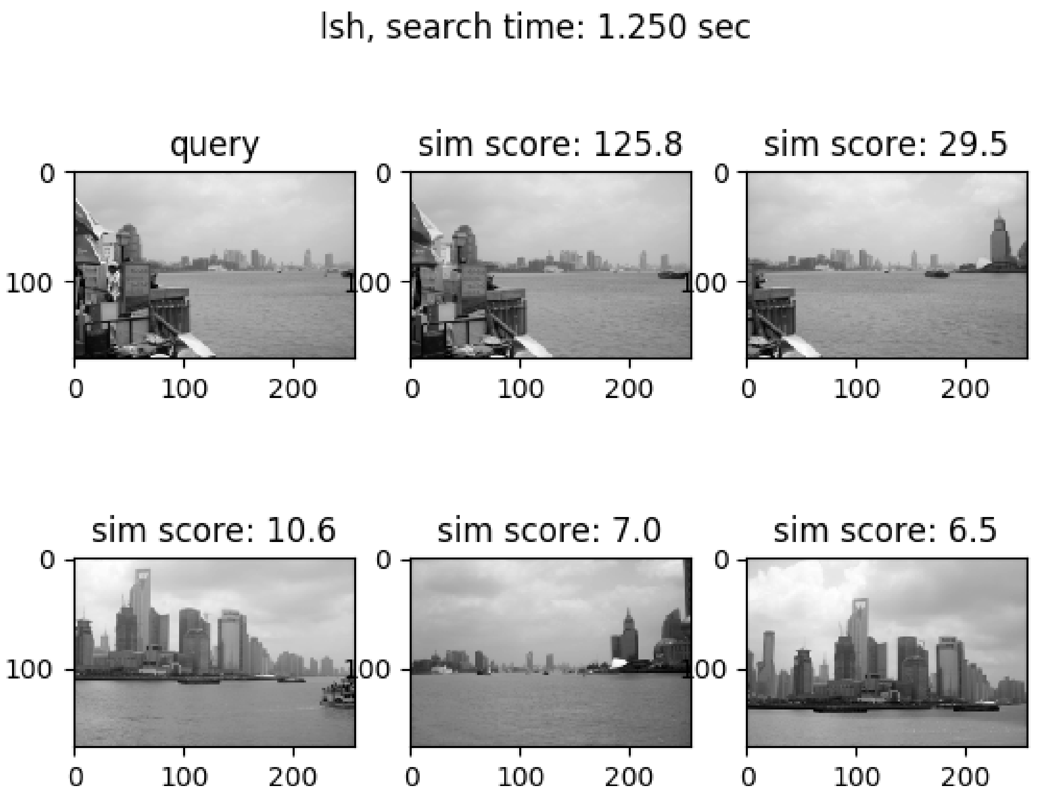
2 hash tables, hash code length 30:



1 hash tables, hash code length 20:



1 hash tables, hash code length 50:



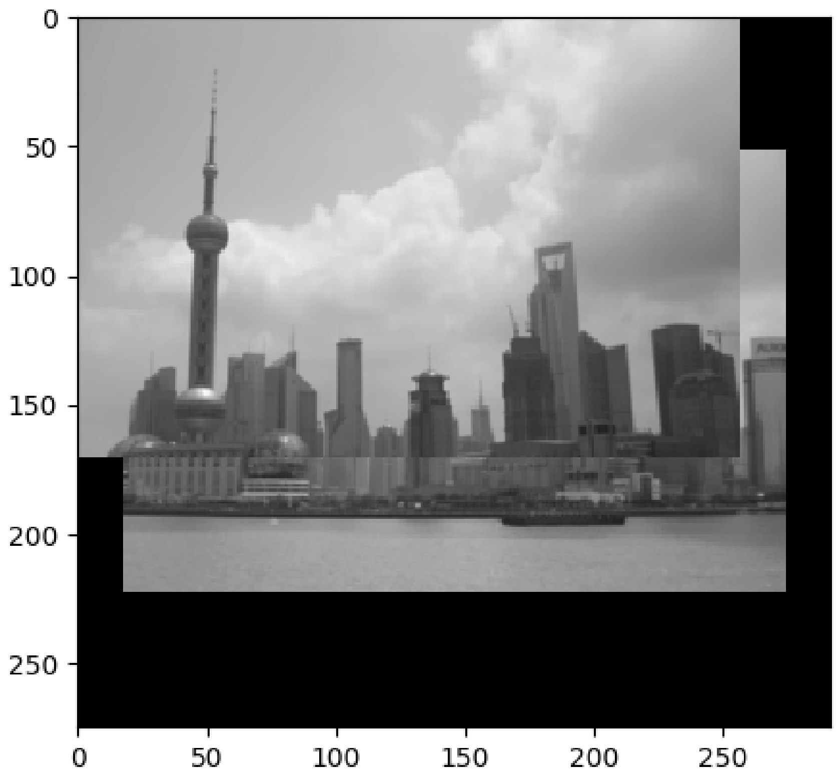
HOUGH TRANSFORM

The goal of this method is to find the 'most representative' translational vector.

Given that there are many different translational vectors between two images, the approach used consists in building a grid of translational vectors (similar translational vectors will end up in the same cell in the grid), and each cell will consist of the sum of the scores of the translational vectors on it.

In the x-axis, the grid contains all possible values of the x-component of the translational vectors. Analogous is done for y-axis.

I found that an optimal size of each cell in the grid was a 3x3 pixels.


0
50
100
150
200
250
0
50
100
150
200
250