CSC420H1 L0101, Fall 2018
Assignment 2

Name: JingXi, Hao
Student Number: 1000654188

Due Date: 12 October, 2018 11:59pm

1. **Interest Point Detection**

   (a) Write two functions for computing the Harris corner metric using Harris (R) and Brown (harmonic mean) methods. Display your results for the attached image **building.jpg** showing your cornerness metric output. Compare the results corresponding to two different methods. For Harris you can use the code provided in Tutorial C.

   Solution:
   Please see the code implementation for this question, which can be found in the file, ***a2.py***. Based on my code, the output images obtained by computing the Harris corner metric, using Harris and Brown methods are shown below.

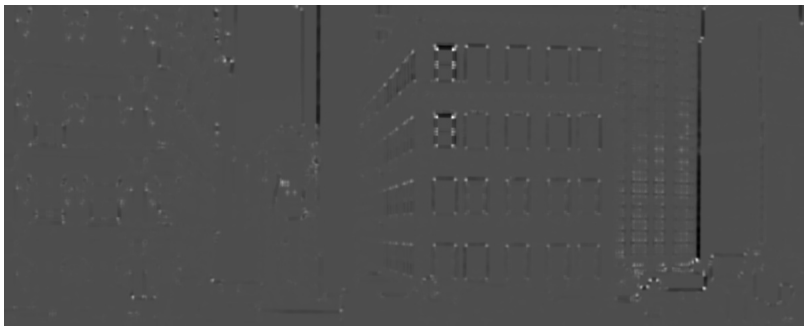

Figure 1: Result Obtained From Harris Method

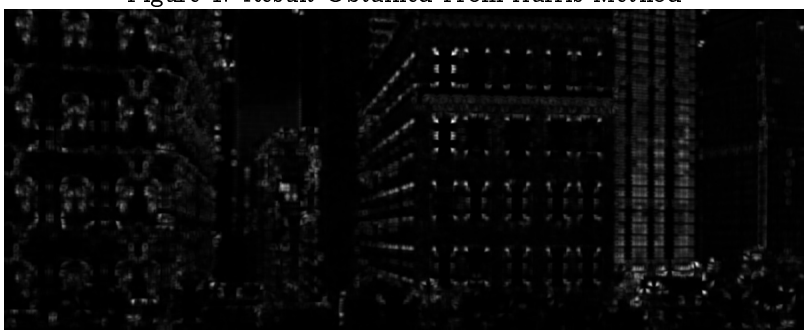

Figure 2: Result Obtained From Brown Method

Based on the result images shown above, we are able to see that Brown method (*Figure 2*) shows more clear corner points than Harris method (*Figure 1*) does, which implies that Brown method works better to reduce the noise and find the corner points.

(b) Write your own function to perform non-maximal suppression using your own functions of choice. Use a circular element, and experiment with varying radii r as a parameter for the output of harmonic mean method. Explain why/how the results change with r. MATLAB users may want to use functions ***ordfilt2()*** , however it can be easily implemented.

Solution:

Please see the code implementation for this question, which can be found in the file, ***a2.py***. The output images for using distinct choice of r are shown below.

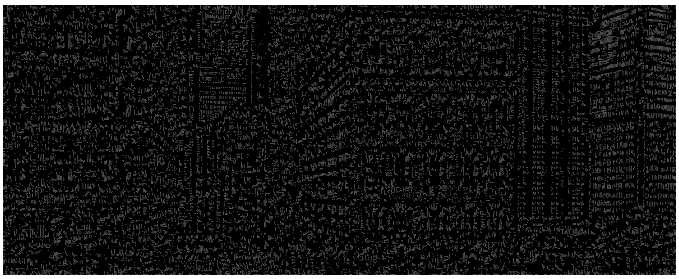

Figure 3: Result Obtained With $r = 1$

Figure 4: Result Obtained With $r = 3$

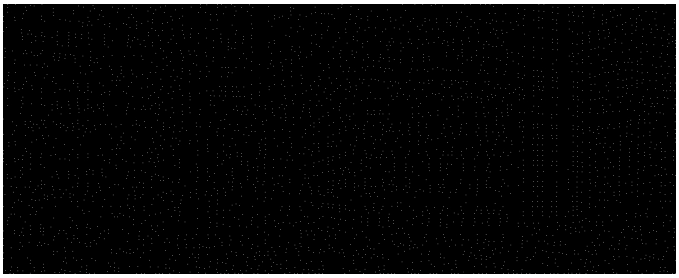

Figure 5: Result Obtained With $r = 5$

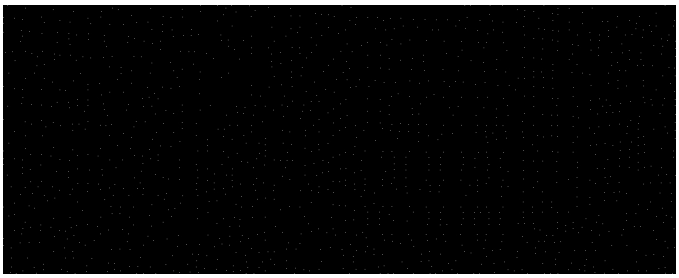

Figure 6: Result Obtained With $r = 9$

Based on the output images shown above, we can conclude that the number of interest points decreases when the value for r increases. Since patch size increases with the increasing of r, thus the probability for the intensity of each pixel to become the local maxima would reduce. Therefore, by increasing the magnitude of r, we gonna end with selecting less number of local maxima, which indicates that we obtain less number of interest points.

(c) Write code to search the image for scale-invariant interest point (i.e. blob) detection using the Laplacian of Gaussian and checking a pixels local neighbourhood as in SIFT. You must find extrema in both location and scale. Find the appropriate parameter settings, and display your keypoints for **synthetic.png** using harmonic mean metric . *Hint: Only investigate pixels with LoG above or below a threshold.*

Solution:

Please see the code implementation for this question, which can be found in the file, ***a2.py***. The output image, showing keypoints for **synthetic.png** by applying scale-invariant interest point detection, is displayed below.

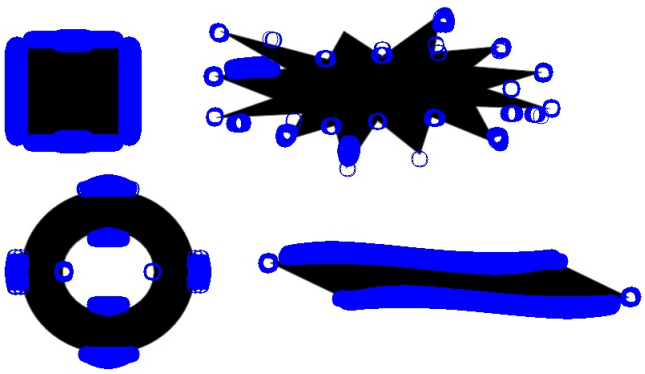

Figure 7: Result Obtained From Applying Scale-Invariant Interest Point Detection

(d) Use open-source implementation of another local feature descriptor that is not covered in the

class, and show the output keypoints on **synthetic.png** and **building.jpg**. Describe the main ideas of your algorithm of choice in a few sentences. You may want to look at Slide 7 in Lecture **8-A** for a list of existing methods.

Solution:

Please see the code implementation for this question, which can be found in the file, ***a2.py***. Based on my code, the output images are shown below by applying *FAST* local feature descriptor to both **synthetic.png** and **building.png**.

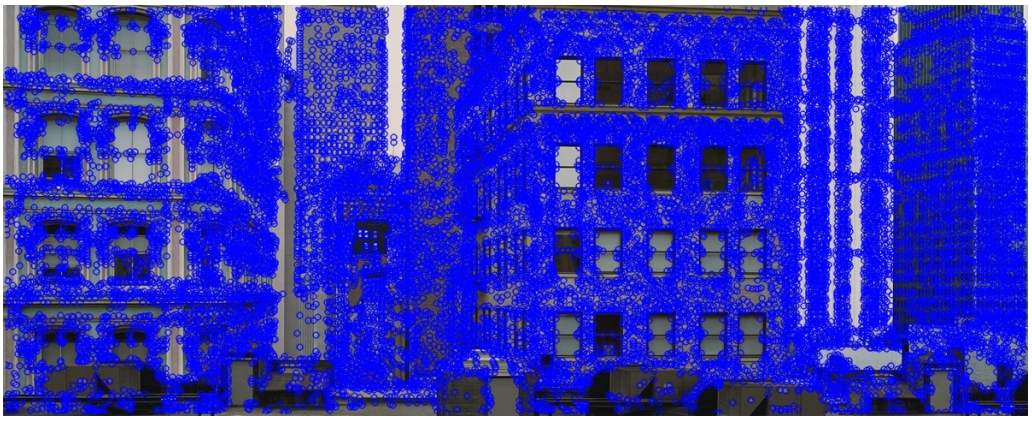

Figure 8: Result Obtained From Applying *FAST* on **building.jpg**

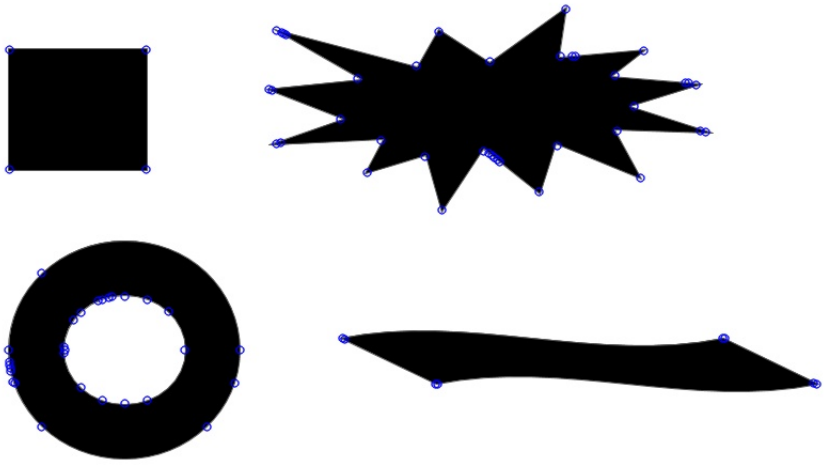

Figure 9: Result Obtained From Applying *FAST* on **synthetic.png**

The reason for choosing *FAST* corner detection algorithm is due to its computational efficiency. This algorithm first picks a pixel in the image which is to be identified as an interest point or not. Then defines a appropriate threshold value. Based on the threshold defined, the method would execute a high-speed test, which only tests 4 pixels in a circle (patch) where the center is

the pixel we currently detect in order to determine whether the pixel chosen is a corner point. Therefore, this algorithm is able to produce faster detection process for finding corner points.

2. **SIFT Matching** (For this question you will use interest point detection for matching using SIFT. You may use a SIFT implementation (e.g. http://www.vlfeat.org/), or another, but specify what you use)

   (a) Extract SIFT keypoints and features for **book.jpg** and **findBook.jpg**.

   Solution:
   Please see the code implementation for this question, which can be found in the file, ***a2.py***.

   (b) Write your own matching algorithm to establish feature correspondence between the two images using the reliability ratio on Lecture 8 . You can use any function for computing the distance, but you must find the matches yourself. Plot the percentage of true matches as a function of threshold. Also, after experimenting with different thresholds, report the best value.

   Solution:
   Please see the code implementation for this question, which can be found in the file, ***a2.py***. Based on my code, the output plot showing the number of matches as a function of threshold is displayed below.

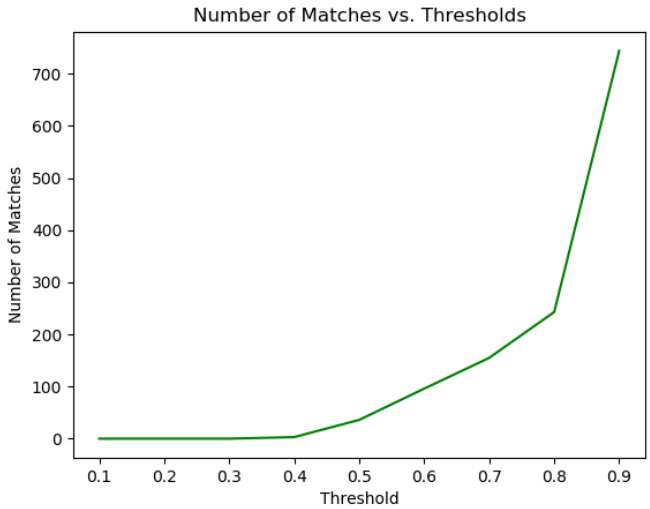

Figure 10: Plot For Number of Matches vs. Thresholds

After experimenting with various values of thresholds (let thresholds be $0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9$), the plot for the function of number of matches in terms of threshold is shown as above. Based on experiment results, the best value for threshold seems to be 0.6 for my case.

(c) Use the top k correspondences from part (b) to solve for the affine transformation between the features in the two images via least squares using the Moore-Penrose pseudo inverse. What is the minimum k required for solving the transformation? Demonstrate your results for various k . Use only basic linear algebra libraries.

Solution:

Please see the code implementation for this question, which can be found in the file, ***a2.py***. I have tried with $k = 3, 5, 7, 10$ to solve for the affine transformation between the features in the two images via least squares using the Moore-Penrose pseudo inverse. Then, the results for affine matrices are shown below with k equals to 3, 5, 7, and 10 respectively.

```
k = 3, [[  3.95330171e-01]
 [ -1.05261398e-01]
 [  9.38219369e-02]
 [  2.67450197e-01]
 [  4.94543450e+02]
 [  1.22385241e+02]]
k = 5, [[  3.82714854e-01]
 [ -9.37124578e-02]
 [  7.93108190e-02]
 [  2.76879351e-01]
 [  4.93245907e+02]
 [  1.22577627e+02]]
k = 7, [[  3.84764864e-01]
 [ -9.51281077e-02]
 [  8.27473672e-02]
 [  2.74561140e-01]
 [  4.92999221e+02]
 [  1.22154745e+02]]
k = 10, [[  3.82793709e-01]
 [ -9.37422544e-02]
 [  7.73036968e-02]
 [  2.77830238e-01]
 [  4.92848099e+02]
 [  1.21941567e+02]]
```

Figure 11: Results Obtained From Solving For Affine Transformation With $k = 3, 5, 7, 10$

The minimum k required for solving the affine transformation should be 3. Since the formula for computing affine transformation, $\mathbf{a}$, is

$$\underbrace{\begin{bmatrix} & \vdots & & & & \\ x_i & y_i & 0 & 0 & 1 & 0 \\ 0 & 0 & x_i & y_i & 0 & 1 \\ & \vdots & & & & \end{bmatrix}}_{\mathbf{P}} \underbrace{\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} \vdots \\ x_i' \\ y_i' \\ \vdots \end{bmatrix}}_{\mathbf{P'}}$$

, therefore, we need to compute 6 unknowns in order to find the affine transformation. Thus, we need to have at least 6 equations for solving 6 unknowns, which implies that at least 6 rows should be in the matrix P. Since for each match, we have 2 more equations. Hence, we need at least 3 matches, which indicates that the minimum k required should be 3.

(d) Visualize the affine transformation. Do this visualization by taking the four corners of the reference image, transforming them via the computed affine transformation to the points in the second image, and plotting those transformed points. Please also plot the edges between the points to indicate the parallelogram. If you are unsure what the instruction is, please look at Figure 12 of [Lowe, 2004].

Solution:

Please see the code implementation for this question, which can be found in the file, ***a2.py***. After applying affine transformation, the output plot for the transformed points of four corners of the reference image, including parallelogram formed by connecting edges between points, is shown below.

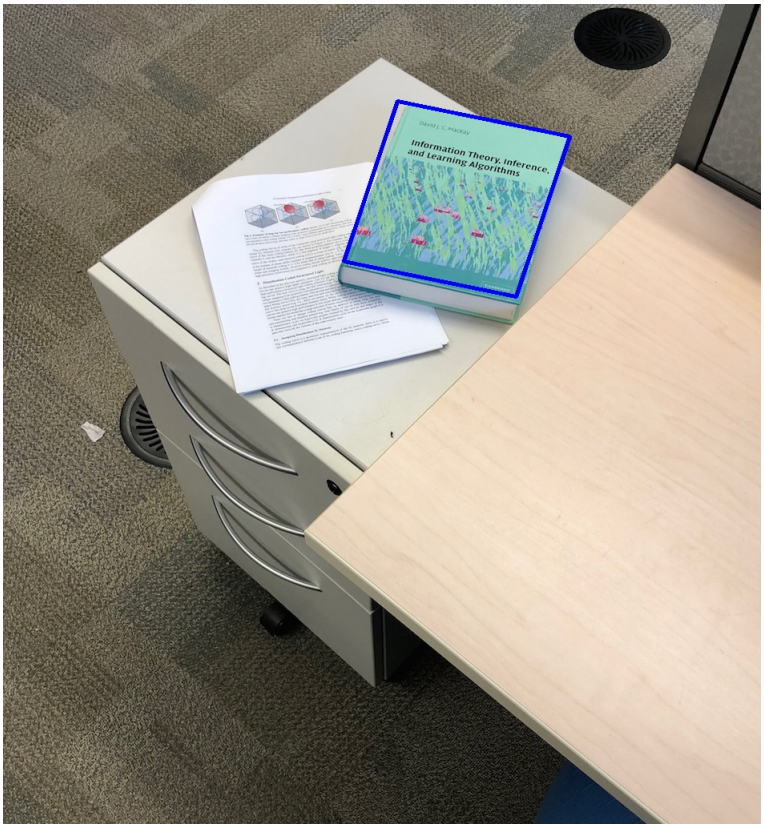

Figure 12: Result Parallelogram Formed by 4 Transformed Corner Points Marked In Blue

(e) Write code to perform matching that takes the colour in the images into account during SIFT feature calculation and matching. Explain the rationale behind your approach. Use **colourTemplate.png** and **colourSearch.png**, display your matches with the approach described in part (d).

Solution:

Please see the code implementation for this question, which can be found in the file, ***a2.py***. The output image displaying the matches between **colourTemplate.png** and **colourSearch.png** with the approach described in part (d) are shown below.

Based on the image shown above, the match results between **colourTemplate.png** and **colourSearch.png** are marked in yellow. For this approach, we first employ $SIFT$ to compute keypoints and descriptors for both images. Then, we use the descriptors obtained from applying $SIFT$ to match the features for two images. In this case, we need to take color into consideration, therefore, we not

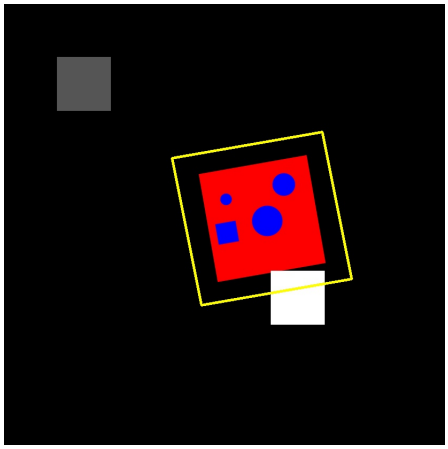

Figure 13: Match Results Between **colourTemplate.png** And **colourSearch.png**

only need to compare the computed value with the threshold but also need to check whether the color for the pixels are matched (should ensure all RGB values are same). If both these conditions are met, then we are able to conclude that we find a match between two images. After finding all matches, we compute the affine transformation matrix based on the matches found between two images. Then, we are able to do transformation for each corner points in **colourTemplate.png** and marked it in **colourSearch.png** with paralellogram to show the matches.

3. **RANSAC**

(a) Assuming a fixed percentage of inliers $p = 0.7$, plot the required number of RANSAC iterations (S) to recover the correct model with a higher than 99% chance (P), as a function of k (1:20), the minimum number of sample points used to form a hypothesis.

Solution:

Please see the code implementation for this question, which can be found in the file, ***a2.py***. Also, the plot for the required number of iterations, S, as a function of k is shown on the next page.

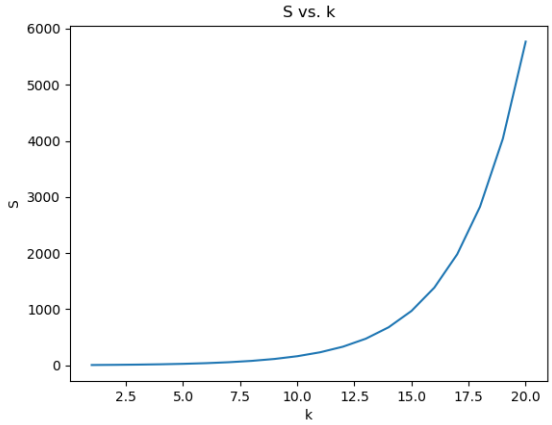

Figure 14: Plot For S vs. k

(b) Assuming a fixed number of sample points required ($k = 5$), plot S as a function of the percentage of inliers p $(0.1 : 0.5)$

Solution:

Please see the code implementation for this question, which can be found in the file, ***a2.py***. Also, the plot for the required number of iterations, S, as a function of the percentage of inliers p is shown below.

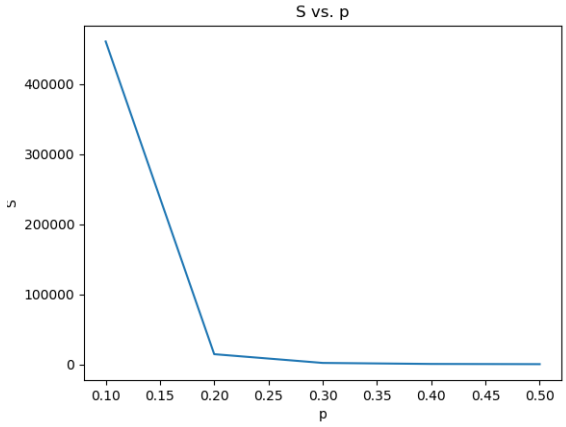

Figure 15: Plot For S vs. p

(c) If $k = 5$ and the initial estimate on the percentage of inliers is $p = 0.2$, what is the the required number of iterations to recover the correct model with $P \geq 0.99$ chance? Assume that you have implemented this and there are 1500 matches in total. In iteration #15, 450 points agree with the current hypothesis (i.e. their error is within a preselected threshold), would the number of required iterations change? explain how and why.

Solution:

When $k = 5$ and the initial estimate on the percentage of inliers is $p = 0.2$, so, based on the formula, we are able to compute the required number of iterations to recover the correct model with $P \geq 0.99$, which is $S = \frac{log(1-P)}{log(1-p^k)} = \frac{log(1-0.99)}{log(1-(0.2)^5)} \approx 14389$ iterations. In addition, in iteration #15, given that there are 450 points agree with the current hypothesis, then we can compute the $p = \frac{450}{1500} = 0.3$. Then, based on the formula, we have that $S = \frac{log(1-P)}{log(1-p^k)} = \frac{log(1-0.99)}{log(1-(0.3)^5)} = 1893$ iterations. Thus, we are able to see that the number of required iterations change. The number of requited iterations should change, shown from *Figure* 12, we are able to see that S decreases when p value change from 0.2 to 0.3 with $k = 5$.