Report

Hongxi Xiang

1 Detection

1.1 Image gradients

We need to compute the gradients I_x and I_y for each pixel.
According to the formula:

$$I_x(i,j) = \frac{I(i,j+1) - I(i,j-1)}{2}$$

We need two neighbors (left and right one) of each pixel to compute gradient I_x. Since convolution is a flipped correlation, the filter to compute I_x should be:

```
IxFilter = np.array([[0,0,0],[0.5,0,-0.5],[0,0,0]])
```

Similarly, the filter to compute I_y should be:

```
IyFilter = np.array([[0,0.5,0],[0,0,0],[0,-0.5,0]])
```

Notice that if we exchange the order of the element "0.5" and "-0.5", the gradient we compute would change sign ("I_x" would become $-I_x$, "I_y" would become $-I_y$). In our case, this would not change the result, since when we compute the auto-correlation matrix M, the minus signs would cancel out. After that, we can use `scipy.signal.convolve2d` function to compute gradients for each pixel.

1.2 Local auto-correlation matrix

According to the formula:

$$M_p = \sum_{p' \in \mathrm{N}(p)} \omega_{p'} \begin{bmatrix} I_x(p')^2 & I_x(p')I_y(p') \\ I_y(p')I_x(p') & I_y(p')^2 \end{bmatrix}$$

First compute 3 matrices to store $I_x(p')^2$, $I_y(p')^2$ and $I_y(p')I_x(p')$ for each pixel:

```
Ixx = np.square(Ix)
Iyy = np.square(Iy)
Ixy = np.multiply(Ix,Iy)
```

Then average these 3 matrices above according to the weight $\omega_{p'}$ given by a Gaussian filter, then we get 3 new matrices `sumofIxx`, `sumofIyy`, and `sumofIxy`, which respectively store the elements of Local auto-correlation matrix M_p for each pixel.

1.3 Harris response function

After having computed Local auto-correlation matrix M_p for each pixel, we can compute the Harris response for each pixel according to the formula:

$$C(i,j) = \det(M_{i,j}) - k\,\mathrm{Tr}^2(M_{i,j})$$

Corresponding code is:

```
#Determinant of M
DetMatrix =  np.multiply(sumofIxx,sumofIyy) - np.square(sumofIxy)
#Trace of M
TraceMatrix = sumofIxx + sumofIyy
# Harris response per pixel
C = DetMatrix - k * np.square(TraceMatrix)
```

1.4 Detection criteria

The keypoints should satisfy two conditions: 1. Its Harris response is above a given threshold; 2. Its Harris response is the local maxima in its 3×3 neighbors.

I do this in 2 steps:

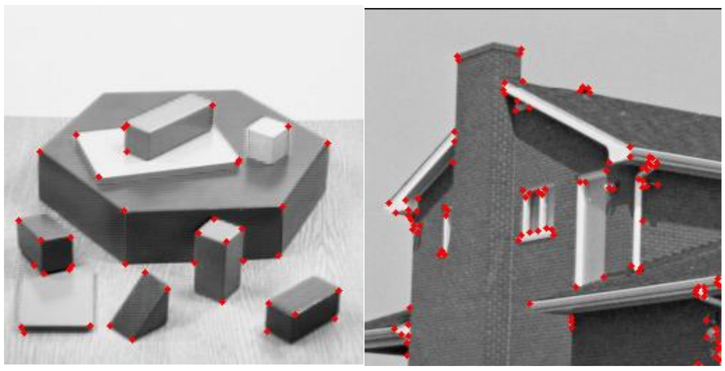
1. Verify whether a pixel is a local maxima: apply the function `ndimage.maximum_filter` to Harris response matrix `C` to get a new matrix `MaxFilteredC`. Then compare `MaxFilteredC` with `C` elementwise, if the elements are the same, then corresponding pixel is the local maxima.
2. Verify whether the local maxima is above threshold.

   Corresponding code is:

   ```
   MaxFilteredC = sc.ndimage.maximum_filter(C, size=3)
   corners = np.asarray(np.where((MaxFilteredC==C) & (C > thresh))).T
   corners =corners[:,[1,0]] # change x,y coordinates
   ```

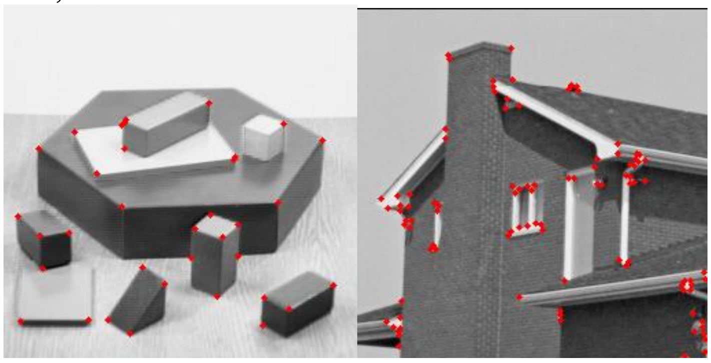
Change threshold T, constant k and standard deviation σ, the following results (some selected examples) are acquired:

1. $\sigma = 1.0$, T = 1e-5, k = 0.04



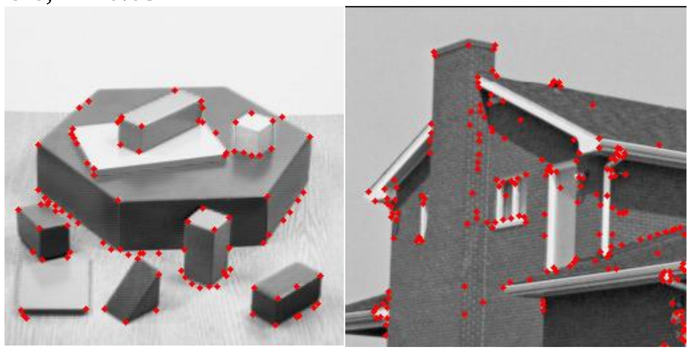
Number of keypoints are 41 and 106 respectively.

2. $\sigma = 1.0$, T = 1e-5, k = 0.05



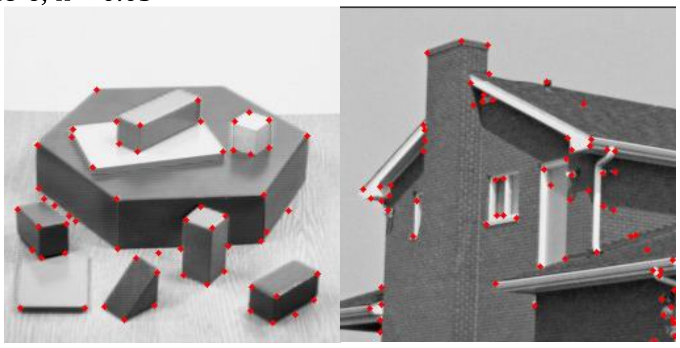
Number of keypoints are 37 and 104 respectively.

3. $\sigma = 1.0$, T = 1e-6, k = 0.05



Number of keypoints are 100 and 181 respectively.

4. $\sigma = 2.0$, T = 1e-6, k = 0.05

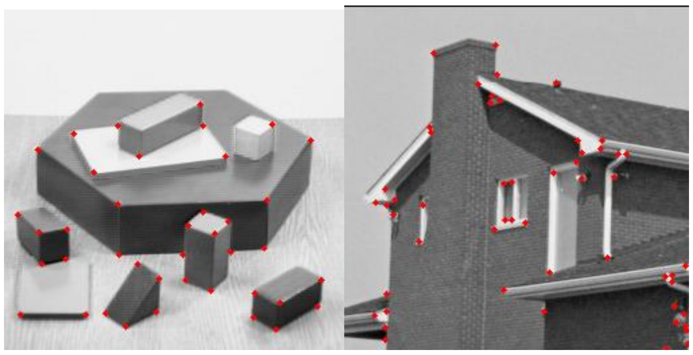


Number of keypoints are 60 and 71 respectively.

It can be seen that:

1. When we increase k, number of keypoints will decrease. This is because Harris response for each pixel will decrease and thus, less points have response above the threshold.
2. When we increase T, number of keypoints will decrease. This is because less points have response above the threshold.
3. When we increase σ, number of keypoints will decrease. This might because increase σ would have stronger averaging effect and thus reduce the number of local maxima in the image.

After tuning, I choose the final parameters to be: $\sigma = 2.0$, T = 9×1e-6, k = 0.04. The results are:



Number of keypoints are 40 and 53 respectively.

Notice that even with "most appropriate" parameter values, there are still some corners can't be detected.

2 Description & Matching

2.1 Local descriptors

We want to filter out the keypoints which are close to boundaries. That means we should delete keypoints whose 9 × 9 patches would be outside the image. We can compute the lower bound and upper bound of the x, y coordinates of the allowed keypoints using patch size and image size and delete the keypoints which are outside the range. The corresponding code is:

```
lowboundx = (patch_size-1)/2

upperboundx = img.shape[0]-(patch_size-1)/2-1

lowboundy = (patch_size-1)/2
```

```
upperboundy = img.shape[1]-(patch_size-1)/2-1


# check x
IndexToDelete =
np.where(((keypoints[:,0])<lowboundx)|((keypoints[:,0])>upperboundx))
keypoints = np.delete(keypoints, IndexToDelete, 0)


# check y
IndexToDelete =
np.where(((keypoints[:,1])<lowboundy)|((keypoints[:,1])>upperboundy))
keypoints = np.delete(keypoints, IndexToDelete, 0)
```

2.2 SSD one-way nearest neighbors matching

We first need to compute the SSD matrix, which would be a $q_1 \times q_2$ matrix, where q_1, q_2 are the number of remaining keypoints (with those close to boundaries already deleted) in image 1 and 2 respectively. The code is:

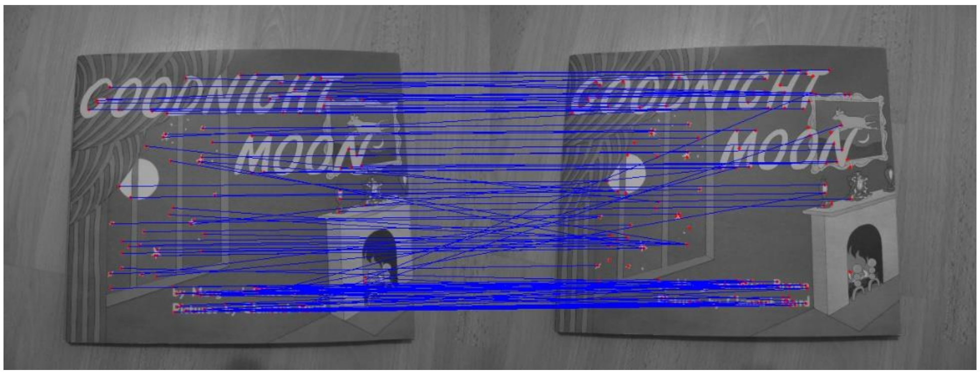
```
# for broadcasting
augmenteddesc1= desc1[:, np.newaxis]
# differenceMatrix: (q1,q2,feature_dim) numpy array
differenceMatrix = augmenteddesc1 - desc2
#distances: (q1, q2) numpy array
distances = np.sum(differenceMatrix**2, axis = 2)
```

Notice that in order to get corresponding subtraction by Python using broadcasting, we need to argument the dimension of `desc1` array to make sure the dimensions of `desc1` and `desc2` match well.

In one-way matching, we choose the nearest neighbour for each keypoint in image 1 and regard it as a valid matching. The code is:

```
Indicesindesc1 = np.array(range(q1))
Indicesindesc2 = np.argmin(distances,axis =1)
matches = np.vstack((Indicesindesc1,Indicesindesc2)).T
return matches
```

The result is:



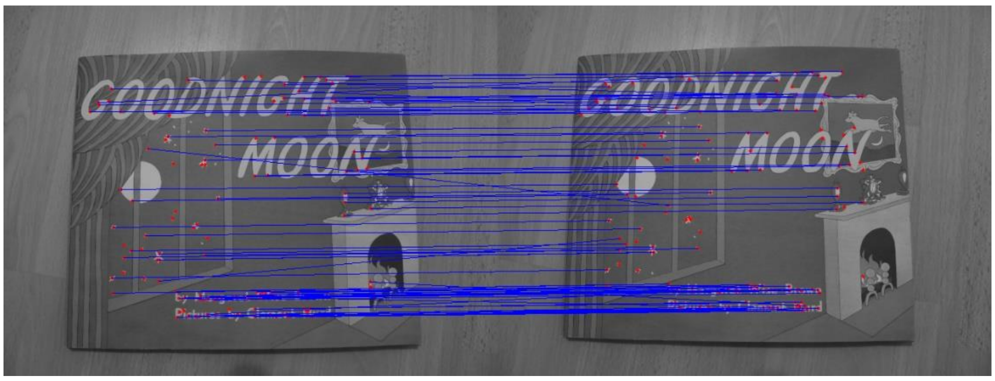
As can be seen, some of the matchings (those not parallel to horizontal axis) are not correct. Thus, one-way matching is not enough in some cases.

2.3 Mutual nearest neighbors / Ratio test

In mutual nearest neighbors, we use one-way matching in both side from image 1 to 2 and from image 2 to 1. Only the matchings are valid in both ways are kept. The code is:

```
# match from img1 to img2

matches_ow1 = match_descriptors(desc1, desc2, "one_way")

# match from img2 to img1

matches_ow2 = match_descriptors(desc2, desc1, "one_way")

matches_ow2 = matches_ow2[:,[1,0]]

# keep the matches which are valid in both ways, and delete the others

remaining = (matches_ow1[:,None] == matches_ow2).all(-1).any(1)

matches = matches_ow1[remaining]

return matches
```

The result is:

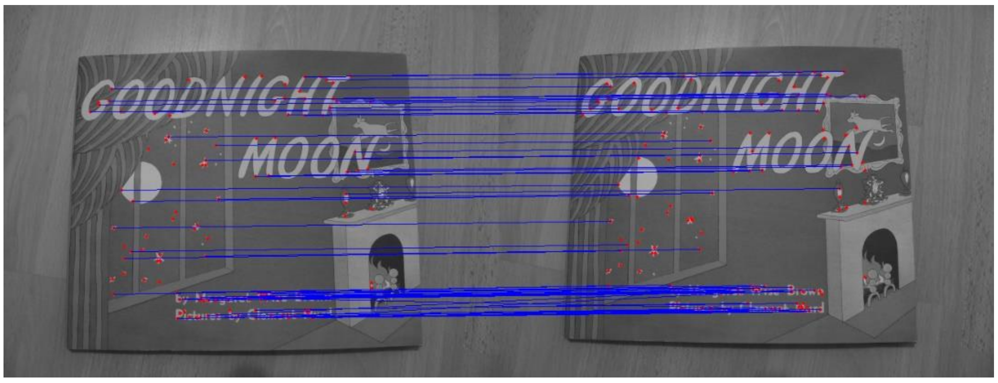


As can be seen, there are less matchings which are not correct (those not parallel to horizontal axis), this is because only matchings are valid in both ways are kept. Most of the improper matchings are deleted. Thus, mutual matching is more accurate than one-way matching. But unfortunately, there are still some matchings which are not correct.

In Ratio test, we essentially do one-way matching, but instead of choosing only the nearest neighbour, we compute the ratio between the first and the second nearest neighbor, only those below certain threshold (this means the matching with the first nearest neighbor is much more likely than the second one) are kept. The corresponding code is:

```
matches_ow = match_descriptors(desc1, desc2, "one_way")
#sort the smallest 2 elements
distances = np.partition(distances,(0,1))
remaining = np.where(((distances[:,0])/(distances[:,1])) < ratio_thresh )
matches = matches_ow[remaining]
return matches
```

The result is:



As can be seen, almost all the improper matchings have been deleted. Thus, in this case, one can say ratio test is even more accurate than the mutual matching.