

Computer Vision Exercise 2

Viviane Yang 16-944-530


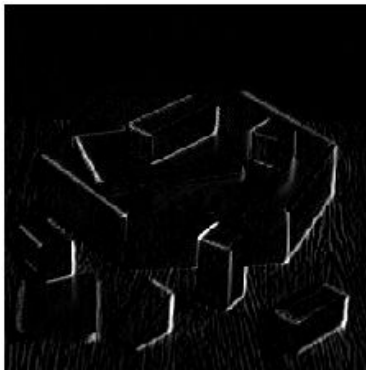

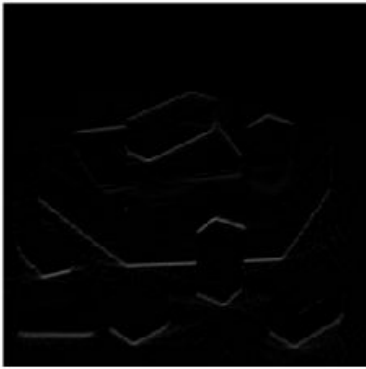
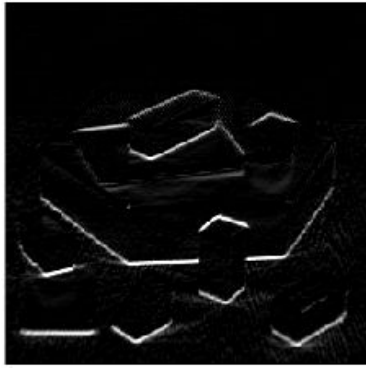
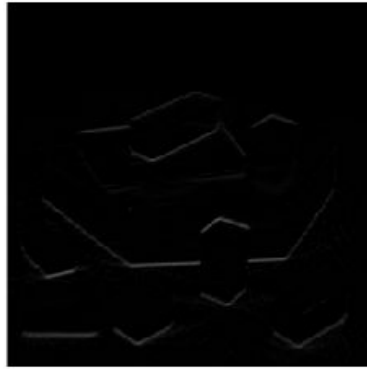
1 Detection

1.1 Image gradients

The values for I_x and I_y for each pixel can be computed by using the filters

$h_x = \frac{1}{2} \cdot [1 \ 0 \ -1]$ and $h_y = \frac{1}{2} \cdot [1 \ 0 \ -1]^T$ and convolve it over the intensity matrix of the image. I am using the `conv2` function in MATLAB with “**same**” option, to only get the convolution of the matrix in the interesting area, where the filter fully overlaps the image. Therefore, the returned matrices for I_x and I_y have the same dimension as the image, which is $n \times m$.

To see whether our gradient is implemented correctly, we can plot the image gradients using our method and the built-in function `imgradientxy` and compare their images.

 <p>I_x using <code>conv2</code></p>	 <p>I_x using <code>conv2</code> and scaled by 5</p>	 <p>I_x using <code>imgradientxy</code> with “central” option</p>
 <p>I_y using <code>conv2</code></p>	 <p>I_y using <code>conv2</code> and scaled by 5</p>	 <p>I_y using <code>imgradientxy</code> with “central” option</p>

By looking at the picture, our method look exactly like the image produced by `imggradientxy`. To make the gradient more visible on the picture, we can scale it with a scaling factor of 5 and you can see the contours in the picture.

```
filter_x = 0.5*[1 0 -1];
filter_y = 0.5*[1 0 -1]';

Ix = conv2(img, filter_x, "same");
Iy = conv2(img, filter_y, "same");
```

1.2 Local auto-correlation matrix

For the computation of the components of M , I computed the $I_x^2, I_y^2, I_y I_x$ matrices and ran a gaussian filter with standard deviation σ over the resulting matrices. This would give the summands of the components and the sum of the summands is computed using again by convoluting the matrices with a neighbor matrix N

```
N = [1 1 1; 1 0 1; 1 1 1]; %neighbor function

Ix2 = imgaussfilt(Ix.^2, sigma);
Iy2 = imgaussfilt(Iy.^2, sigma);
Ixy = imgaussfilt(Ix.*Iy, sigma);

Ix2_sum = conv2(Ix2, N, "same");
Iy2_sum = conv2(Iy2, N, "same");
Ixy_sum = conv2(Ixy, N, "same");
```

There is no need to compute the whole matrix $M \in 2n \times 2m$ since we are only using its components for computing the Harris response function C .

1.3 Harris response function

The Harris response matrix C consists of $\det(M_{ij})$, $Tr^2(M_{ij})$ and k , with the last one being a scalar scaling factor. The determinant of a 2 by 2 matrix can be simply computed using the

formular $\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ac - bd$ and the trace is just the sum of the diagonal elements in a matrix. With these components we can compute C .

```
detM = Ix2_sum.*Iy2_sum - Ixy_sum.^2;
traceM = Ix2_sum+Iy2_sum;
C = detM - k*traceM.^2;
```

1.4 Detection criteria

There are two detection criteria, with the first one being the corner must be a local maximum and second its Harris response function must be higher than a given threshold. The first condition is realized using the `imregionalmax` function in MATLAB, By default, it will look for the maximum in its 3x3 neighborhood and return a matrix with the same dimensions as the picture of ones and zero. It is one if that pixel is a local maximum, so we scale those pixel with their Harris response function to test the second condition.

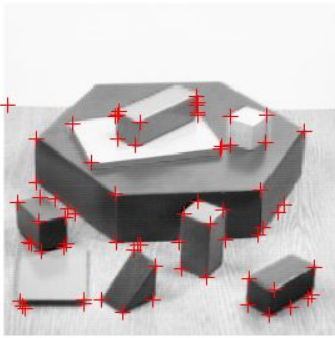

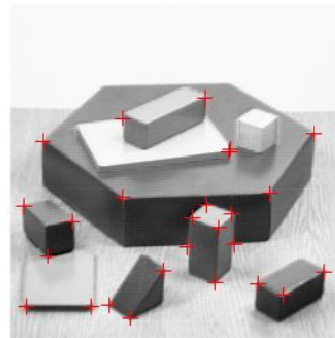
```
Cmax = imregionalmax(C).*C;  
C = (C==Cmax) & (C>thresh);
```

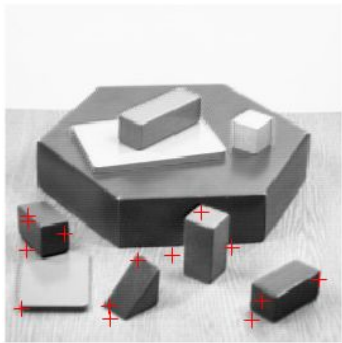
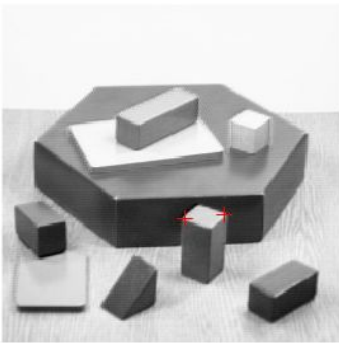
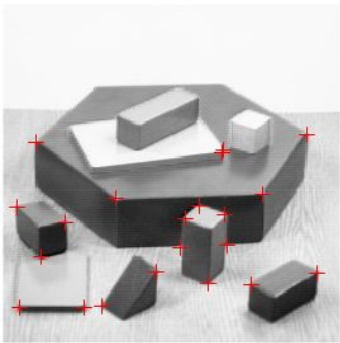
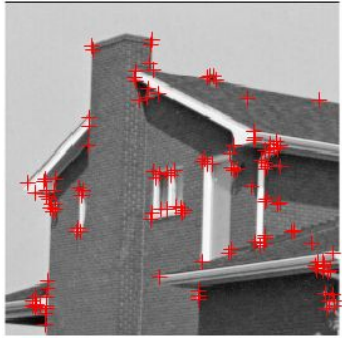
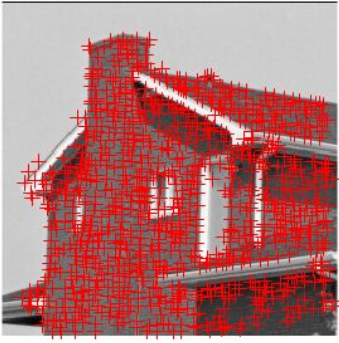
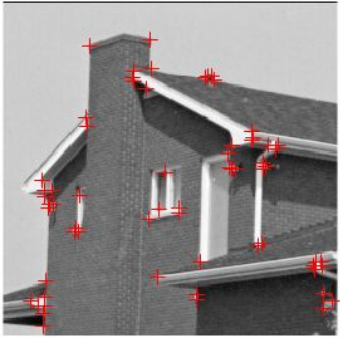
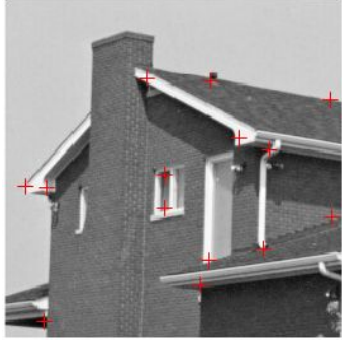
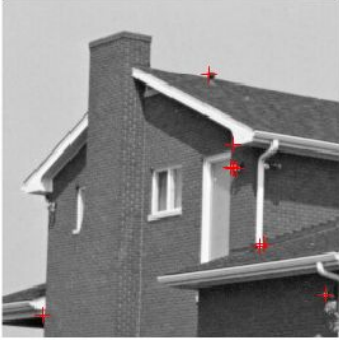
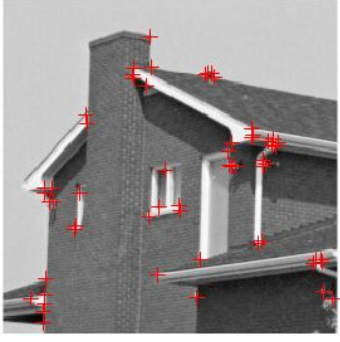
During the implementation, I ran into the issue that a lot of corners are detected around the borders of the image, especially around the border of 2 pixels. I assume that this is due to the convolution function used for the gradient computation and the harris response function. At the borders, the gradient is high, the filter does not fully fit into the image plane yet and therefore is zero-padded. This results in wrong or high values such that corners are detected easily there.

This is why I manually removed all corners that lay within the border of 2 pixels, assuming that one would not place the interesting object at the border while taking a picture.

Now we need to determine our parameters with the following considerations:

- Sigma
 - smoothes gradient, thus large sigma means less corners and position might shift
- Threshold - absolute threshold for corner values, significant drop in #corners if high
- k - negative impact on Harris response function, therefore high k leads to less corners

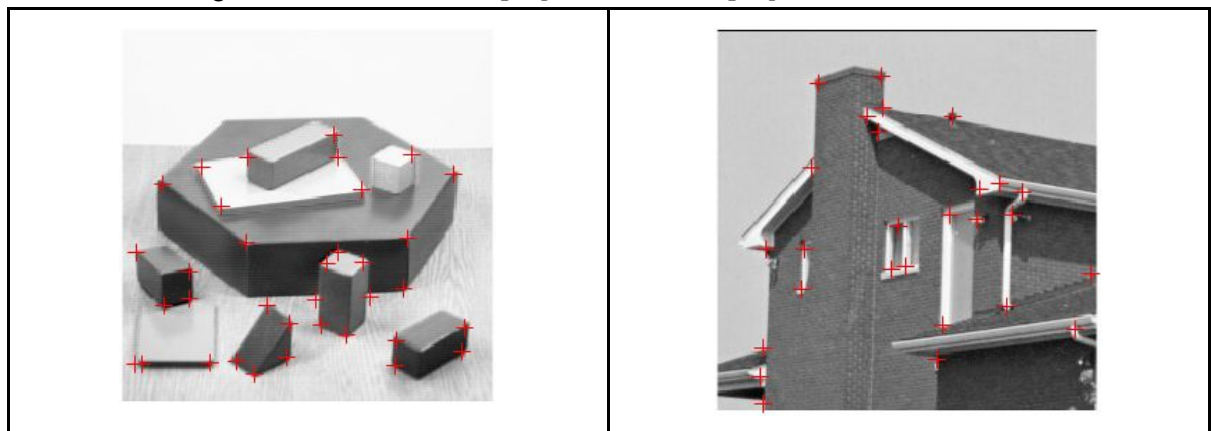
	sigma	threshold	k
low			

high			
low			
high			

After a little bit of tuning, I got the following values for my parameters:

```
k = 0.05;
thresh = 0.0002;
sigma = 5;
```

And the resulting corners of `blocks.png` and `house.png` look like this:



2 Description & Matching

2.1 Local descriptors

Before calling `extractPatches` on the key points, we first need to remove those whose patch would not fit into the picture anymore. So we need to remove all corners in the border that are within the width of half of the patch size.

```
%filter out corners around the borders of the image
patchsize = 9;
[n,m] = size(img);

%remove all keypoints that are too close to the border.
border = ceil(patchsize/2);
keypoints = keypoints(:, keypoints(2,:) > border & keypoints(2,:) <
                        n-border);
keypoints = keypoints(:, keypoints(1,:) > border & keypoints(1,:) <
                        m-border);

%extract patches
descriptors = extractPatches(img, keypoints, patchsize);
```

2.2 SSD one-way nearest neighbors matching

For the one-way nearest neighbors matching, we need to find a key point for picture 1 in picture 2, that has the minimum sum-of-squared-differences of its descriptor values. The distance of two descriptors can be computed as follows:

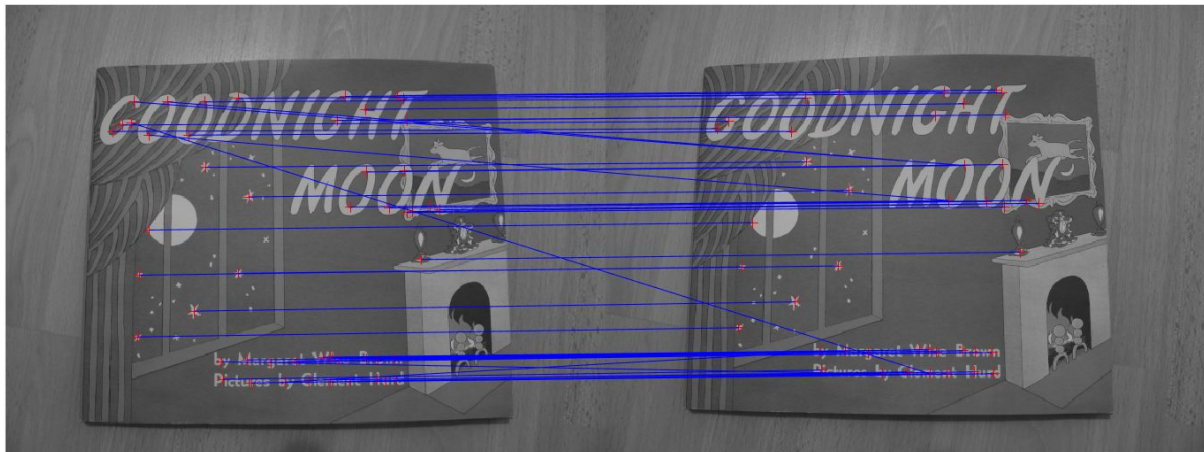
```
distances(i,j) = pdist2(descr1(:,i)', descr2(:,j)', 'squaredeuclidean');
```

This must be done for every pair of keypoints in picture 1 and picture 2.

Then, the one-way match is found by finding the index of the descriptor in picture 2 that has the smallest distance using the `find` function in MATLAB.

```
distances = ssd(descr1, descr2);
r = size(descr1,2);
s = size(descr2, 2);
%find the one with the minimum distance for each in descr1
matches = zeros(2, r);
for i = 1:r
    matches(1,i) = i;
    matches(2, i) = find(distances(i,:) == min(distances(i,:)));
end
```


The result can be seen here:

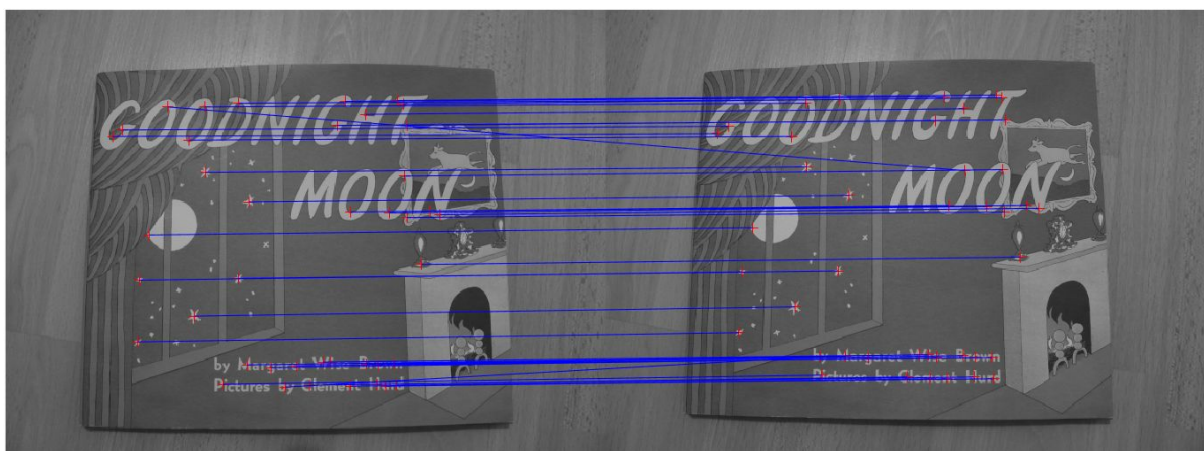


2.3.1 Mutual nearest neighbors

For mutual nearest neighbors, we do the same thing as for one-way neighbor, plus the condition that it is only a match if the pair is also matched when swapping the images. This is why we can only add the match if the following if condition is true:

```
for i = 1:r
    match1 = find(distances(i,:) == min(distances(i,:)));
    if match1 <= s
        match2 = find(distances(:,match1) ==
                       min(distances(:,match1)));
        if match2 == i
            matches = [matches, [i match1]'];
        end
    end
end
```

The matching looks a bit better in a way, that key points in picture 1 that were not detected in picture 2 are not necessarily matched.



2.3.2 Ratio test

The ratio test has an even stronger condition, the match must be twice as good (threshold = 0.5) as the second best match. This means that the SSD must be half of the one of the second best match.

The second best match with the second smallest distance can be found by sorting the distance vector of the keypoint in picture 1. Then, the match is only added if the ratio of the distances between best and second best match is smaller than a threshold value.

```
for i = 1:r
    distancesi = sort(distances(i,:));
    if(distancesi(1)/distancesi(2) < thres)
        %distancesi(1)/distancesi(2)
        match = find(distances(i,:) == distancesi(1));
        matches = [matches, [i match]'];
    end
end
```

Thus, it results in a good matching with less matches, since most of the blue lines are horizontal:



If we want an even better matching and can sacrifice a few more corners, we can set the threshold even smaller (in this case to 0.1):

