Wenyan Bi

README

The aim of this project is to simulate SIFT to implement local feature matching algorithm. To do this, we basically need three major steps:

1) Interest point detection (Harris Corner detection);

2) Feature Descriptor generation;

3) Feature Matching.

I. Algorithm

1. Interest point detection

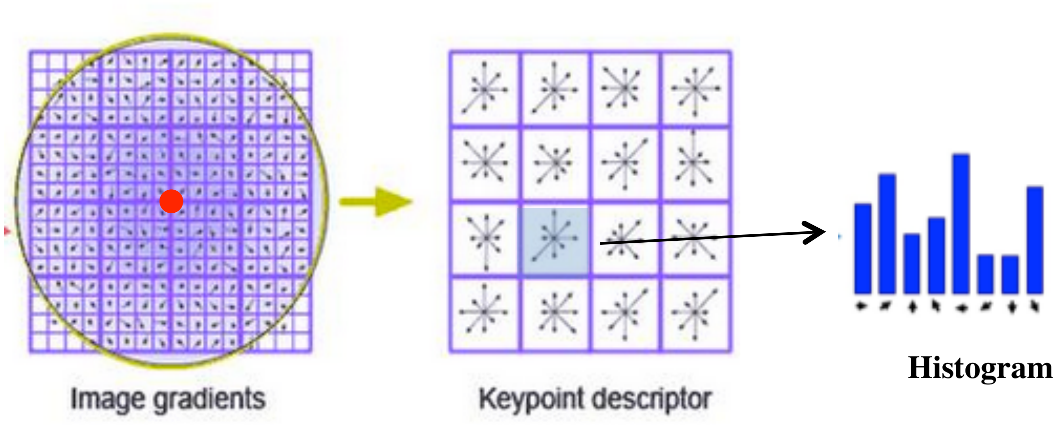
I used Harris corner detection to locate the interesting points (corners). The fundamental idea is that sliding a window in both x and y directions will cause huge intensity change.

To do this, I first computed the x and y derivative of the image Ix and Iy. Then I computed Ix^2, Iy^2, and IxIy, and convolved them with a Gaussian. After then I calculated the value of determinant/trace. Lastly, I found the local maxima above the a threshold (here I used 0.05 or 0.1) and reported them as the detected interest points.

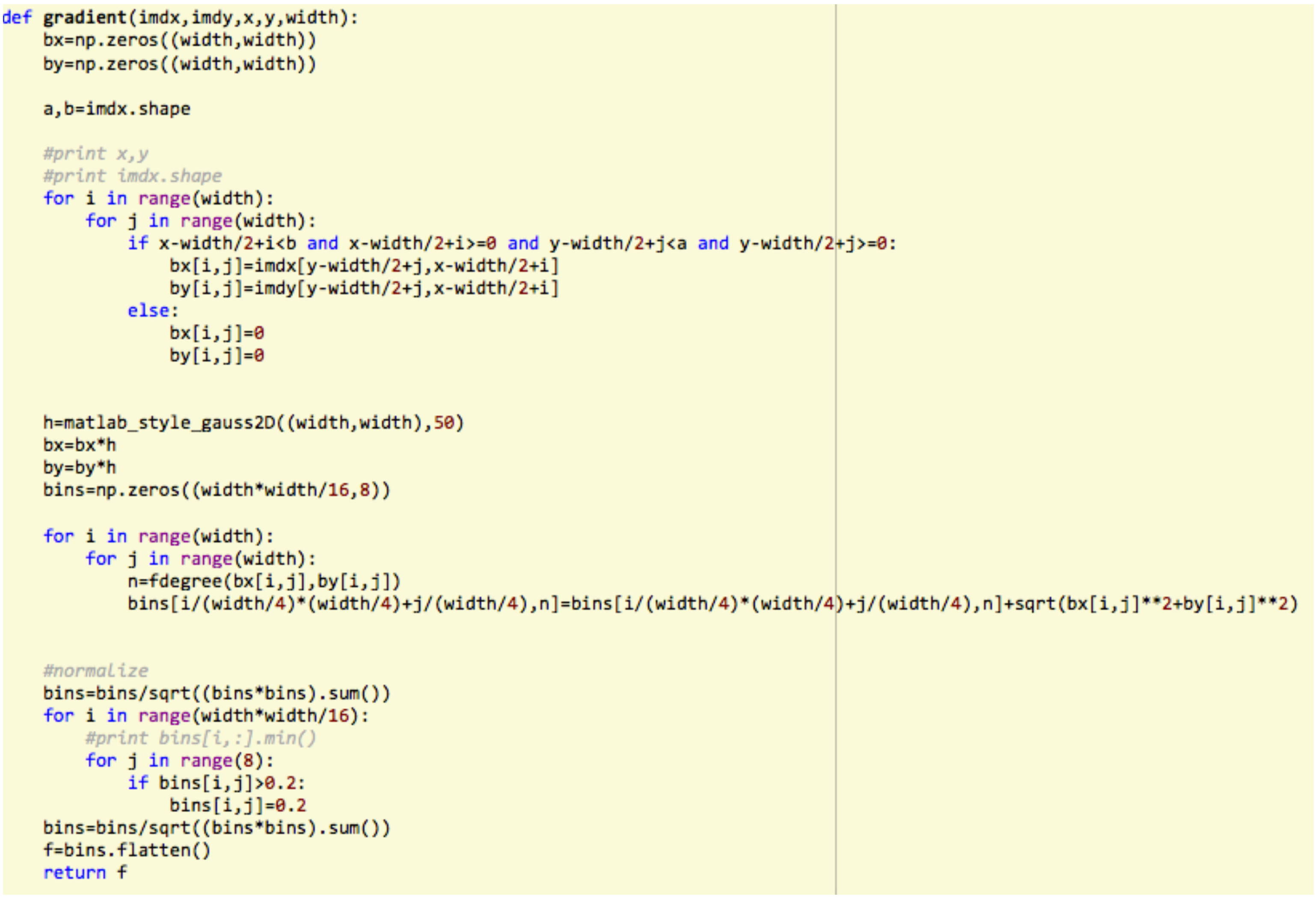
I totally copied Dr. Bei's code for this step.

2. Feature Descriptor generation (SIFT)

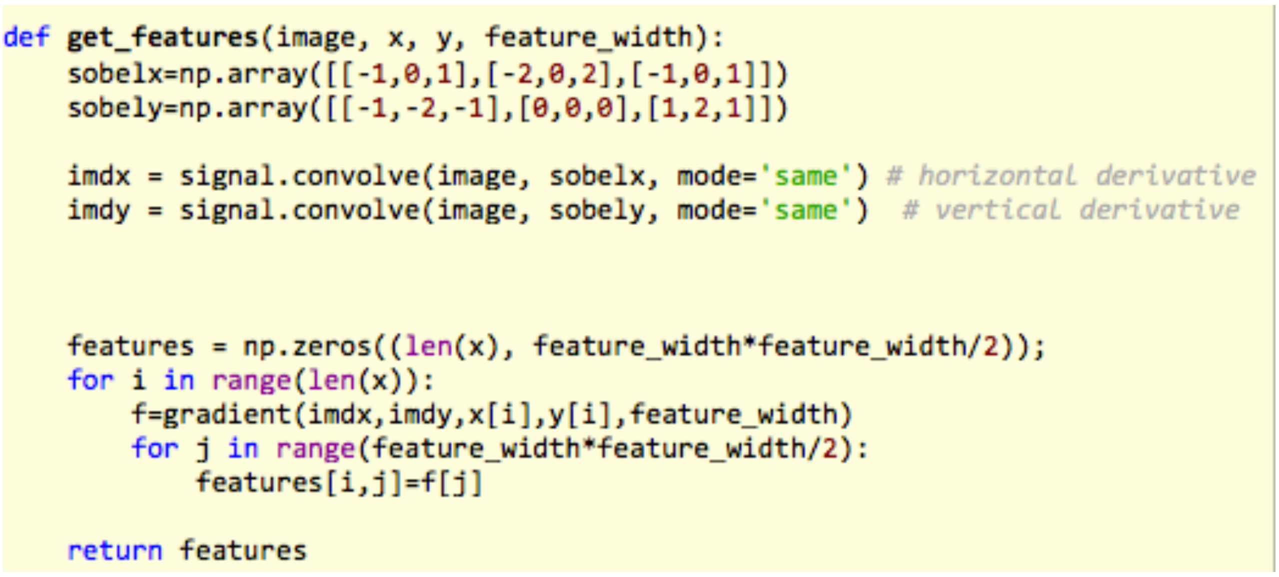
I simulated SIFT to generate the feature descriptors. Firstly, for each interest point, a window (16*16) around the point was generated, which was then split into sixteen 4*4 grid. Within each grid, gradient magnitudes and orientations were calculated, which were then put into a 8 bin histogram (0~2 π). For example, if the gradient orientation is 35 degree, then it would be put into the first bin (0~ π /4). Then a Gaussian kernel was convolved with the gradient of the orientations so that the center part (around interest point) would get larger weights. By doing so, I generated 8*4*4=128 numbers for each interest points. Then I normalized these 128 numbers. For those number which were larger than 0.2, I equaled them to 0.2. After doing this, I normalized these numbers again and get the descriptor (128 vectors) for each interest point.



I wrote 2 functions to implement this. “Gradient” is used to generate the descriptor (128 vectors) of each interest point; then “get_features” is used to calculate the descriptors of all interest points of the 2 matching images.



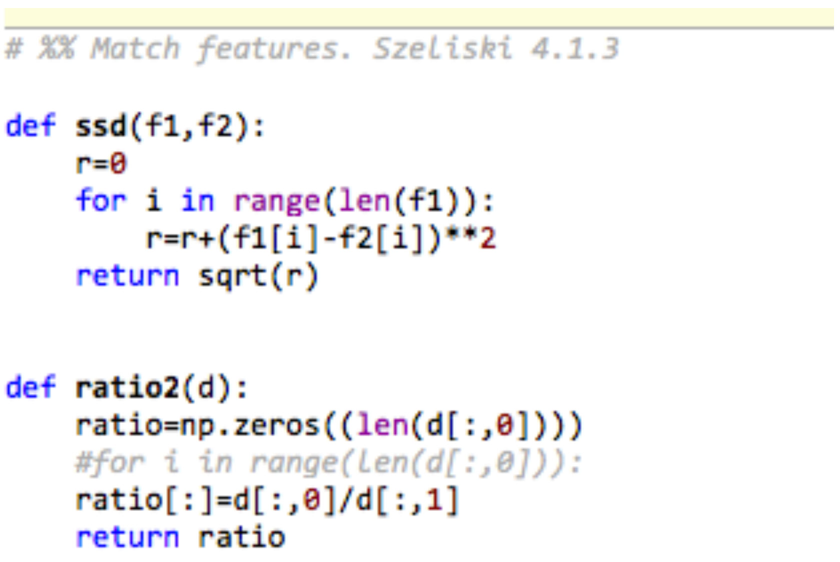
```python
def gradient(imdx,imdy,x,y,width):
    bx=np.zeros((width,width))
    by=np.zeros((width,width))

    a,b=imdx.shape

    #print x,y
    #print imdx.shape
    for i in range(width):
        for j in range(width):
            if x-width/2+i<b and x-width/2+i>=0 and y-width/2+j<a and y-width/2+j>=0:
                bx[i,j]=imdx[y-width/2+j,x-width/2+i]
                by[i,j]=imdy[y-width/2+j,x-width/2+i]
            else:
                bx[i,j]=0
                by[i,j]=0


    h=matlab_style_gauss2D((width,width),50)
    bx=bx*h
    by=by*h
    bins=np.zeros((width*width/16,8))

    for i in range(width):
        for j in range(width):
            n=fdegree(bx[i,j],by[i,j])
            bins[i/(width/4)*(width/4)+j/(width/4),n]=bins[i/(width/4)*(width/4)+j/(width/4),n]+sqrt(bx[i,j]**2+by[i,j]**2)


    #normalize
    bins=bins/sqrt((bins*bins).sum())
    for i in range(width*width/16):
        #print bins[i,:].min()
        for j in range(8):
            if bins[i,j]>0.2:
                bins[i,j]=0.2
    bins=bins/sqrt((bins*bins).sum())
    f=bins.flatten()
    return f
```



```python
def get_features(image, x, y, feature_width):
    sobelx=np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
    sobely=np.array([[-1,-2,-1],[0,0,0],[1,2,1]])

    imdx = signal.convolve(image, sobelx, mode='same') # horizontal derivative
    imdy = signal.convolve(image, sobely, mode='same')  # vertical derivative



    features = np.zeros((len(x), feature_width*feature_width/2));
    for i in range(len(x)):
        f=gradient(imdx,imdy,x[i],y[i],feature_width)
        for j in range(feature_width*feature_width/2):
            features[i,j]=f[j]

    return features
```

3. Feature Matching

I used nearest neighbor distance ratio to do the feature matching. For each interest point, I found its two nearest neighbors (d1: nearest neighbor; d2: second nearest neighbor). If d1/d2<threshold, I printed the interest point out as the matching point. I tried lots of times and found that the best threshold was 0.75 (0.8 would get too many false alarms, and 0.7 would get too few hits).



```
# %% Match features. Szeliski 4.1.3

def ssd(f1,f2):
    r=0
    for i in range(len(f1)):
        r=r+(f1[i]-f2[i])**2
    return sqrt(r)


def ratio2(d):
    ratio=np.zeros((len(d[:,0])))
    #for i in range(len(d[:,0])):
    ratio[:]=d[:,0]/d[:,1]
    return ratio
```

II. Discussion

My algorithm (proj2.py, show_correspondence.py) didn't work so well. For the Notre Dame pairs, I used 0.1 as Harris Corner detection threshold, 64*64 window size, and 0.75 as ratio test threshold. My code used 20-30 minutes to generate 31 matches/3 bad matches=90.3% correct (see Figure 1). Although the ACC was acceptable, I was wondering why my code generated so few points. I tried different thresholds, Gaussian size/sigma, and window size, but none of them would make a huge difference. Thus, I thought over other possible reasons.

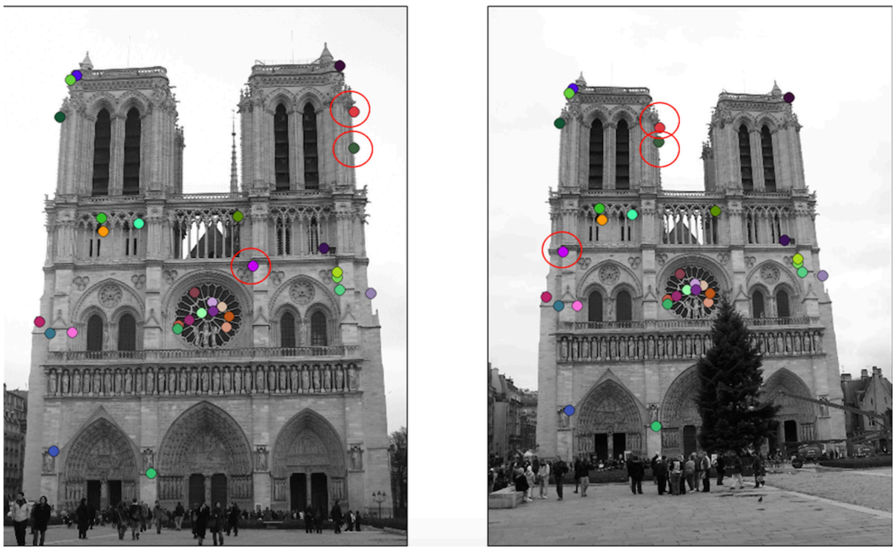


Figure 1 Matching results of my code

My first guess was that the Harris Corner detection code might not be correct. Because for some matching points shown on the ground truth image, my Harris Corner code didn't even detected them as the interests points (see Figure 2)! Then I used cv2.cornerHarris (proj2_cv2.harris.py, show_correspondence_cv2.harris.py) to check whether my Harris Corner code is correct. The parameter I used was 16*16 window size and 0.75 as ratio test threshold. As Figure 3 shows, the results were better. I got 41 matches/3 bad matches=92.7% correct. Taken into the limited time, I didn't try other algorithms. Anyway, the results were still not good enough, which lead me to think other possible reasons.

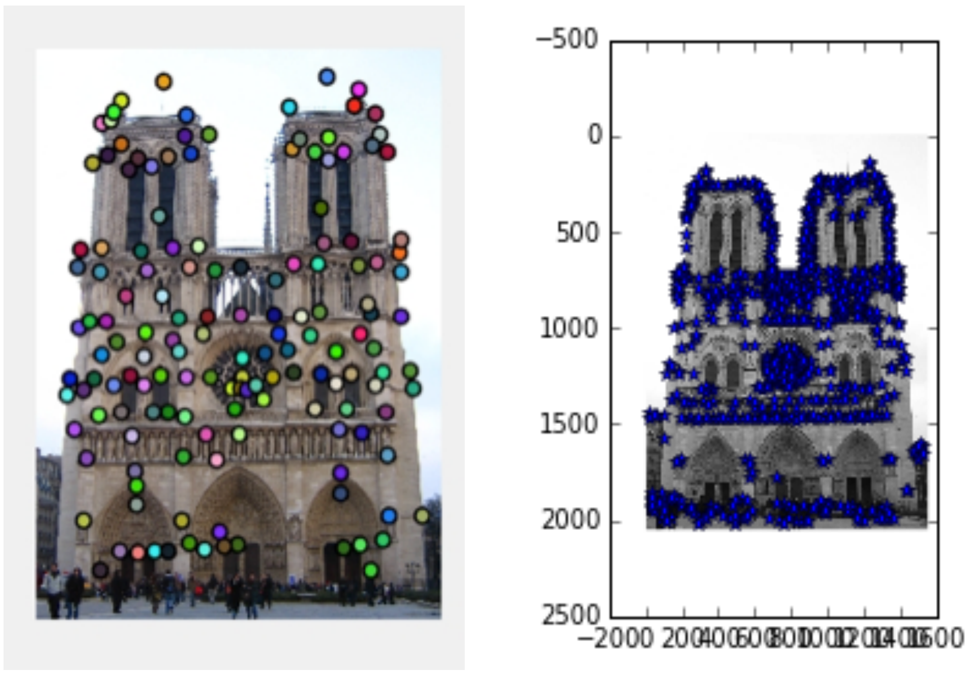


Figure 2 Ground truth (left) and interest points I generated (right)

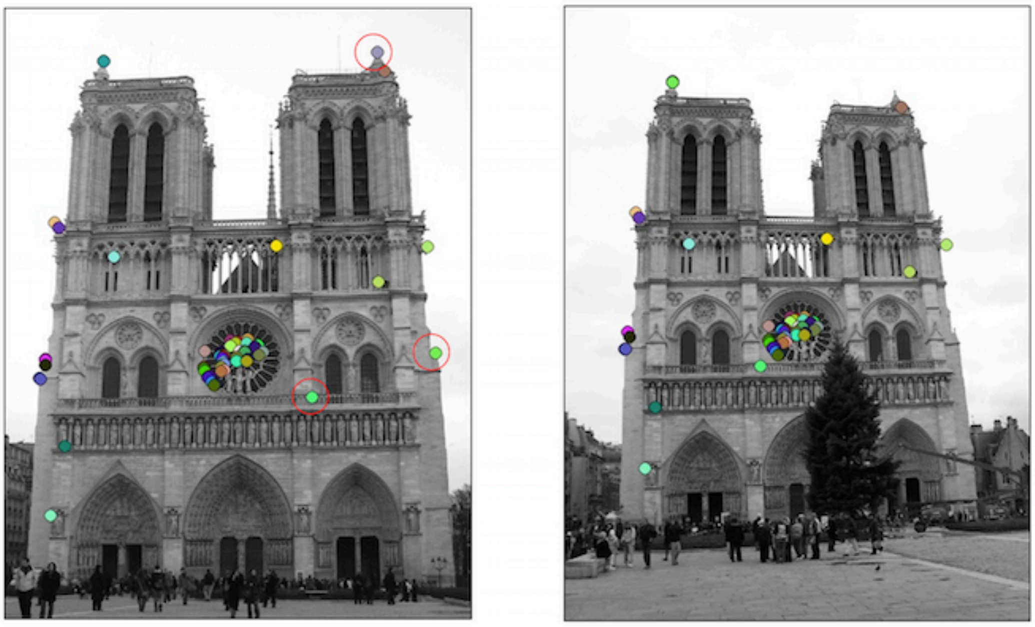


Figure 3 Matching results using cv2.cornerHarris

My second guess was that my code wasn't so good at dealing angle changes. I actually didn't do anything to handle the situation where same object/feature but with different angles. Then I used another "famous" pair (Yosemite), which only contains object translation but without any transform. The algorithm I used was 0.05 as Harris Corner detection, 64*64 window size, 0.75 as ratio test threshold. As Figure 4 shows, my code generated 106 matches in total, out of which there was only 1 bad match. The ACC for this pair was 99.1%. I think this might be the main reason why my code didn't work well. However, due to limited time, I didn't improve my code.

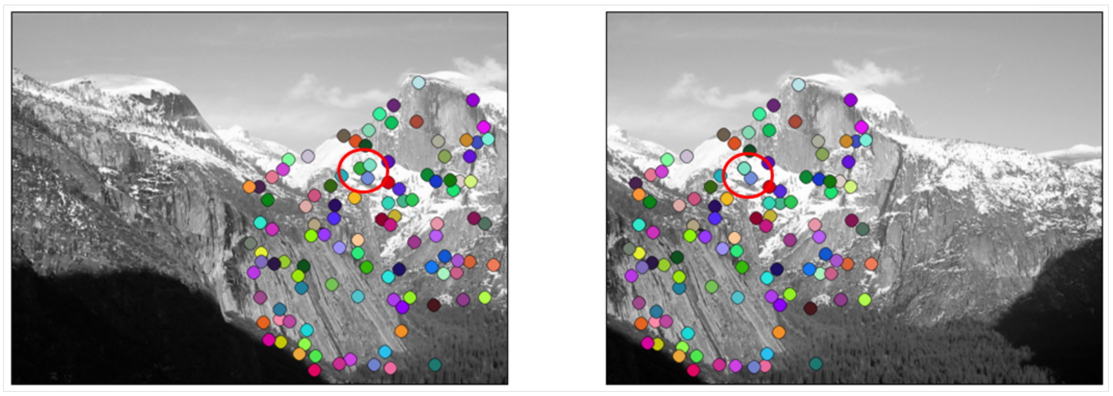


Figure 4 Yosemite matching results (106 matches/1 bad match=99.1% correct)