LYU Mingzhi

G2002734E

AI6121 Asg2

Method To Calculate Disparity

To match the points of the pair of rectified images of the same scene captured from two viewpoints, the most straight-forward idea is matching the points from the two images which is most similar to each other at the same row. Here, I used Sum of Squared Differences (SSD) method to compute the disparity for each point.

Input:

$Image\ of\ left\ viewpoint: ImagL = [p_{i,j}]_{h\times w}$

$Image\ of\ right\ viewpoint: ImagR = [p_{i,j}]_{h\times w}$

$Range\ of\ Disparity: MaxDisparity$

Out:

$Disparity\ Map: DisparityMap = [d_{i,j}]_{h\times w}$

SSD Algorithm:

1. $for\ each\ row\ in\ ImagL$:

2. $for\ each\ pixel\ p\ in\ row$:

3. $best_{disparity} = argmin_{d\in(0,MaxDisparity)}\left(\sum_{i\in W}(I_i - I'_{i-d})^2\right)$,

$where\ I_i\ is\ a\ pixel\ in\ ImagL\ at\ location\ i, and\ I'_{i-d}\ is\ a\ pixel\ in\ ImagR\ at\ locatoin\ i - d.\ W\ is\ the\ window\ whose\ center\ is\ at\ the\ location\ of\ p.$

4. $DisparityMap[i] = best_{disparity}$

Experiment And Analysis

SSD algorithm introduced above is applied to generate the disparity map as follows (Fig 1, Fig 2):

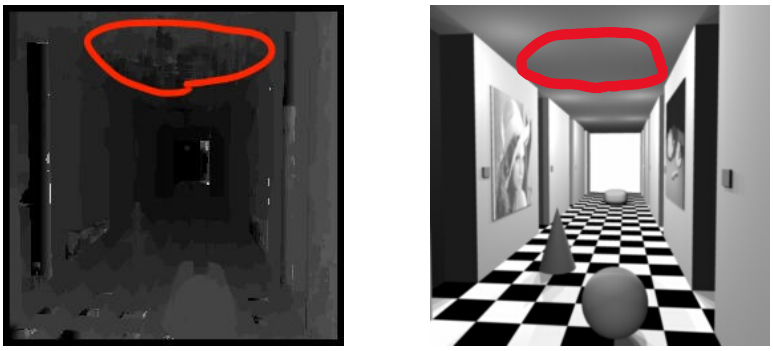
Figure 1: The left image is the picture taken from the left viewpoint and the middle image is the picture taken from the right viewpoint for the same scene. The right image is the disparity map.

Figure 2: The left image is the picture taken from the left viewpoint and the middle image is the picture taken from the right viewpoint for the same scene. The right image is the disparity map.

As the disparity maps show, generally, the changes of depths of the whole scenes are rendered successfully. Darker pixels mean smaller disparity on that pixels. However, the performance is not perfect enough. Obviously, some details are not accurate and there are a number of noise blocks appearing in the disparity maps, which are much darker or brighter than its neighborhood pixels. The main reasons are as follows:

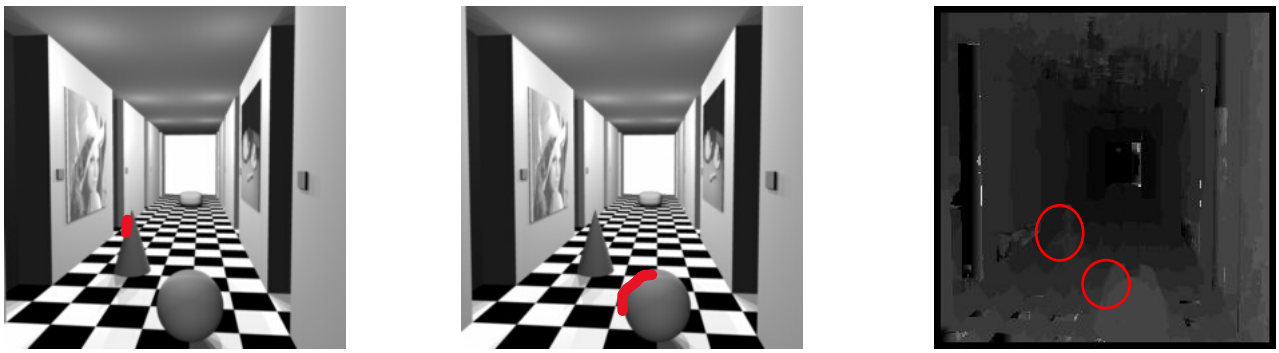
1. For homogenous region of the scene, SSD Algorithm will fail to detect the correct disparity. This is because that in a homogenous region, all the pixels are of the same or similar intensity. Therefore, for a certain range of disparity, cost function

$\sum_{i \in W}(I_i - I'_{i-d})^2$ will output the same cost value for all the d in the range of disparity and for the pixel i in the region. (Example 1)



Example 1: The left image is the disparity map and the right one is image taken from the left viewpoint. In the red circles, the region in the right image is nearly homogeneous and corresponding part in the disparity map has large noise.

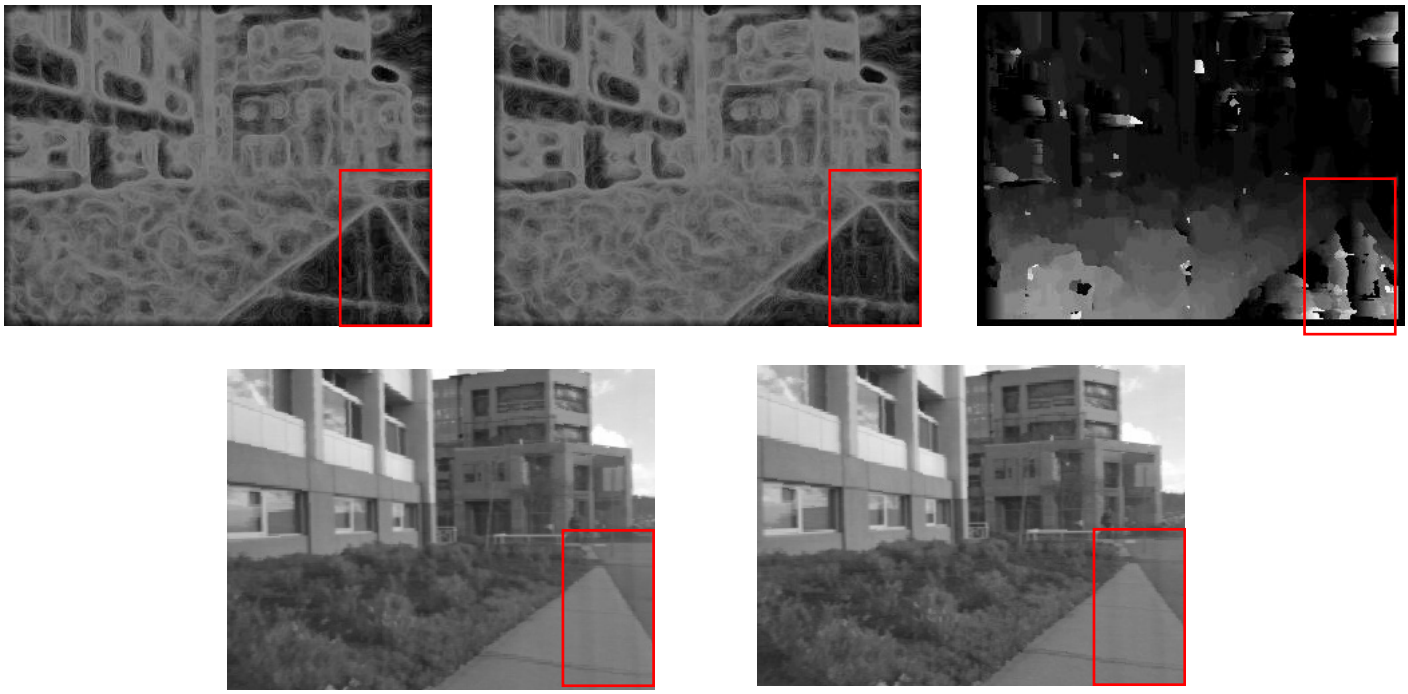
2. Occlusion exists in the pair of images taken from the two viewpoints. As the Example 2 shows, there are some points which appear in one of the images but disappear in the other one because some objects in the other one occlude them.



Example 2: For the first two images, the red marks of an image indicate the pixels that are occluded in the other image. For the disparity map on the right hand size indicate the noise caused by this phenomenon.

3. Noise in the images will also causes noise in the generated disparity map. As the Example 3 shows, there are some noise occurring in the original pictures, which will influence the disparity map seriously. Here, rank transformation is applied on the images and it is obviously that the noise in the images is dependent from the

viewpoints where the pictures are taken.



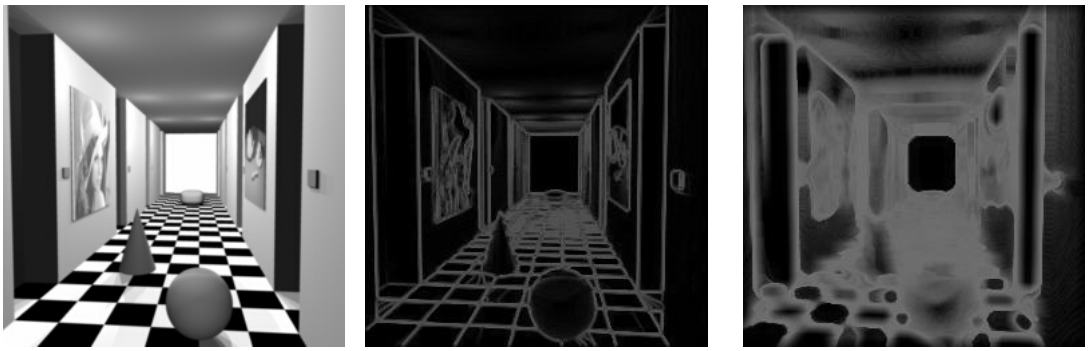
Example 3; For the first two images which is generated by extended rank transformation[1], there are some noise in the red frames. Therefore, in the disparity map on the right hand side, there are also noise in the red frame.

Proposed Solution

1. To tackle the failure of SSD on the homogeneous region of the image, I use an extension of the rank transformation to extract more texture features for the images, so that the transformed homogeneous region will be less homogeneous, as Example 4 shows. The value of a pixel in the image is defined as following:

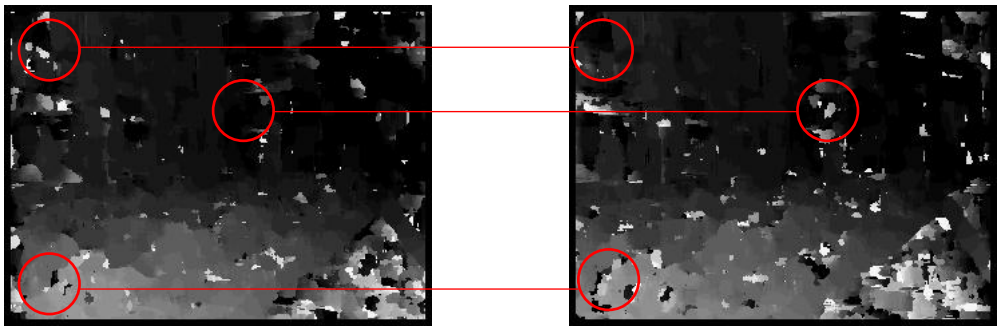
$$Rank = \sum_{q \in W} \min\left(1, \max\left(0, \frac{I_q - median}{K}\right)\right),$$

where W is a sliding window on the image, and q is the pixel in the window, and I_q is the intensity of the pixel q, and $median$ is the median value of the intensity of the pixels in the window. Finally, K is the hyperparameter. Smaller it is, more texture features of the image can be extracted. Moreover it will reduce the noise introduced by illumination with large K.



Example 4: The left image is its original version. The last two images are processed by the extension rank transformation, respectively with $K = 18, 8$.

2. To tackle the occlusion effect, I used Left-Right Consistency Check. Firstly, the disparity maps will be respectively generated for the images taken from the left viewpoint and right viewpoint. Then, each pixel in the disparity map of left viewpoint will have a corresponding pixel in the disparity map of the right viewpoint. And we find each pair of pixels using the disparity values of the pixels of left viewpoint's disparity map. Finally, we will check the difference between the disparity values of the two pixels in each pair. If the difference of a pair is larger than a threshold, we will regard the pixel of disparity map of left viewpoint as an occluded point, and replace its disparity value with the median of the disparity value of its neighbor pixels in their disparity map. (Example 5)



Example 5: The left image is the disparity map of left viewpoint and the right one is the disparity map of right viewpoint. In the red circles, there are some pixels of one of the disparity maps, which have large difference in disparity values with their corresponding pixels in the other disparity map.

3. To tackle the noise in the original images, I used median filter to process the final disparity map. The main idea of the median filter is to run through the signal entry by entry, replacing each entry with the median of neighboring entries. The pattern of neighbors is called the "window", which slides, entry by entry, over the entire signal.[1] Finally results are as follows. For Figure a and b, the proposed method seems not outperform the pure SSD algorithm and have more noise in the disparity map, while for Figure c and d, the proposed method does reduce the noise of the final disparity map and seems better. (Figure 3)

[1] https://en.wikipedia.org/wiki/Median_filter

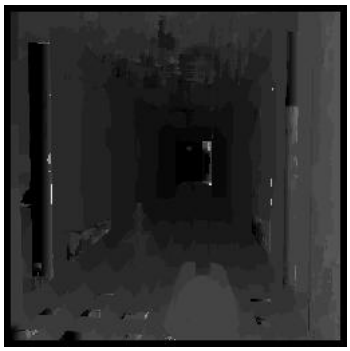


Figure a

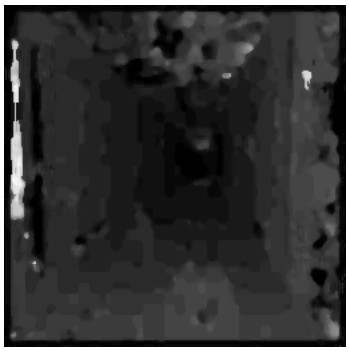


Figure b

Figure c

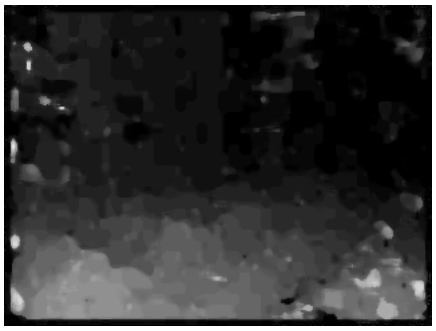


Figure d

Figure 3: Figure a and c are the disparity maps generated only by SSD algorithm. Figure b and d are the disparity maps generated with SDD and the procedures we described above.

Reference:

[1] Zhao, G., Du, Y. K., & Tang, Y. D. (2012). A new extension of the rank transform for stereo matching. In Advanced Engineering Forum (Vol. 2, pp. 182-187). Trans Tech Publications Ltd.

Appendix of Source Code (Python):

```
import numpy as np
from PIL import Image
def rank_transform(img,w_size,K=18):

    w_boundary=int(w_size/2)
    transformed_img=np.zeros(img.shape,np.uint8)
    w_size_adjust=255/(w_size**2)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            window=img[max(0,i-w_boundary):min(img.shape[0]-
1,i+w_boundary),max(0,j-w_boundary):min(img.shape[1]-1,j+w_boundary)]
            median=np.median(window)
            rank=0
            for u in range(window.shape[0]):
                for v in range(window.shape[1]):
                    rank+=min(1,max(0,(window[u,v]-median)/K))
            transformed_img[i,j]=int(rank*w_size_adjust)
    return transformed_img
def LRC(filenamel,filenamer,kernel_size,max_disparity):
    imgl=Image.open(filenamel).convert('L')
    imgr=Image.open(filenamer).convert('L')
    imgl,imgr=np.asarray(imgl),np.asarray(imgr)
    assert imgl.shape==imgr.shape
    kernel_boundary=int(kernel_size/2)
    # imgl=np.pad(imgl,kernel_boundary,'constant')
    # print(imgl[4:-4,4:-4 ])
    # imgr=np.pad(imgr,kernel_boundary,'constant')
```

```
    offset_adjust = 255 / max_disparity
    # imgl=rank_transform(imgl,5)
    # Image.fromarray(np.asarray(imgl)).save('imgl1.jpg')
    # imgr=rank_transform(imgr,5)
    # Image.fromarray(np.asarray(imgr)).save('imgr1.jpg')
    # exit()
    dml=ssd(imgl,imgr,offset_adjust,kernel_boundary,max_disparity,1,0)
    dmr=ssd(imgl,imgr,offset_adjust,kernel_boundary,max_disparity,0,1)

Image.fromarray(np.asarray(dmr*offset_adjust,np.uint8)).save('disparity_map_r.jpg')

Image.fromarray(np.asarray(dml*offset_adjust,np.uint8)).save('disparity_map_l.jpg')
    disparity_map=np.zeros(imgl.shape,np.uint8)
    exit()
    for i in range(dml.shape[0]):
        for j in range(dml.shape[1]):
            if abs(dml[i,j]-dml[i,j-dml[i,j]])>6:
                disparity_map[i][j]=np.median(dml[max(0,i-
w_boundary):min(dml.shape[0],i+w_boundary),max(0,j-
w_boundary):min(dml.shape[1],j+w_boundary)])
    # disparity_map*=offset_adjust
    Image.fromarray(np.asarray(dml*offset_adjust,np.uint8)).save('disparity_map.jpg')
def ssd(imgl,imgr,offset_adjust,kernel_boundary,max_disparity,lmode,rmode):
    disparity_map=np.zeros(imgl.shape,np.uint8)

    for x in range(kernel_boundary,imgr.shape[0]-kernel_boundary):
        for y in range(kernel_boundary,imgr.shape[1]-kernel_boundary):
            best_disparity=max_disparity
            loss=float('inf')
```

```
                for disparity in range(max_disparity):
                    ssd=0
                    # disparity+=1
                    # left_boundary=max(y-disparity-kernel_boundary,0)
                    # print(y+kernel_boundary-(y-kernel_boundary-min(y-disparity-kernel_boundary,0)),y-disparity+kernel_boundary-left_boundary)
                    # if y-disparity+kernel_boundary<0:
                    #         continue
                    # ssd=imgl[x-kernel_boundary:x+kernel_boundary,y-kernel_boundary-min(y-disparity-kernel_boundary,0):y+kernel_boundary]-imgr[x-kernel_boundary:x+kernel_boundary,left_boundary:y-disparity+kernel_boundary]
                    # ssd=(ssd**2/(ssd.shape[0]*ssd.shape[1])).sum()
                    for u in range(-kernel_boundary,kernel_boundary):
                        for v in range( -kernel_boundary,kernel_boundary):
                            if lmode==1:
                                ssd_tmp=int(imgl[x+u,y+v])-int(imgr[x+u,y+v-disparity*lmode])
                            else:
                                ssd_tmp=int(imgl[x+u,y+v+disparity if y+v+disparity<imgl.shape[1] else y+v+disparity- imgl.shape[1]])-int(imgr[x+u,y+v])
                            # ssd+=ssd_tmp*ssd_tmp
                            ssd+=abs(ssd_tmp)
                    if ssd<loss:
                        loss=ssd
                        best_disparity=disparity
                disparity_map[x,y]=best_disparity
                # disparity_map[x,y]=best_disparity*offset_adjust
            # exit()
        # color_coef=int(255/(np.max(disparity_map)-np.min(disparity_map)))
```

```
        # disparity_map=((disparity_map-np.min(disparity_map))*color_coef)
        # disparity_map=(disparity_map*(255/max_disparity))
        return disparity_map
        if np.max(disparity_map)>255:
            print('exceed 255')
            exit()
        print(disparity_map[0],type(disparity_map))
        Image.fromarray(np.asarray(disparity_map)).save('disparity_map.jpg')

LRC('xid-14489712_1.jpg','xid-14489713_1.jpg',6,20)
# LRC('img11.jpg','imgr1.jpg',11,30)
```