Assignment II: Disparity Computing

Li Hantao, G2101725H, MSAI, hli038@e.ntu.edu.sg

Stereo vision **uses stereoscopic ranging techniques to estimate a 3D model of a scene. It employs triangulation, a popular ranging technique, to compute the depth from 2D images. A key parameter in the triangulation is disparity, which is inversely proportional to the depth and can be computed from the correspondence points of two images of the same scene that are captured from two different viewpoints.**

This assignment consists of the following tasks:

1. Describe the procedure of disparity computing given a pair of rectified images of the same scene captured from two different viewpoints.

2. Write a computer algorithm that computes the disparity of two images.

3. Apply the developed algorithm to the two provided image pairs and derive the corresponding disparity maps. Discuss your observation of the obtained disparity maps.

4. Discuss what affects the disparity map computation, and any possible improvements of your developed algorithm. Implement and verify your ideas over the provided test images.

I. Introduction

Stereo is the shape from 'motion' between two views. *Stereo vision* is the term used for the process of inferring 3D depth information from 2D images. [1] 2D images may be from a Stereo Rig, usually consisting of two cameras slightly displaced horizontally, similar to our two eyes. In fact, stereopsis takes inspiration from our ability to infer depth information using our eyes.

A. Stereo Vision

A *parallax* is a displacement or difference in the apparent position of an object viewed along two different lines of sight and is measured by the angle or semi-angle of inclination between those two lines. Change in relative angular displacement of image points across different camera views when seeing 3D points.

B. Depth Estimation

Obviously, the most crucial part in '*disparity computing given a pair of rectified images of the same scene captured from two different viewpoints*' is estimating depth using the disparity map. Therefore, we are going to illustrate the method of estimation depth briefly. [2]

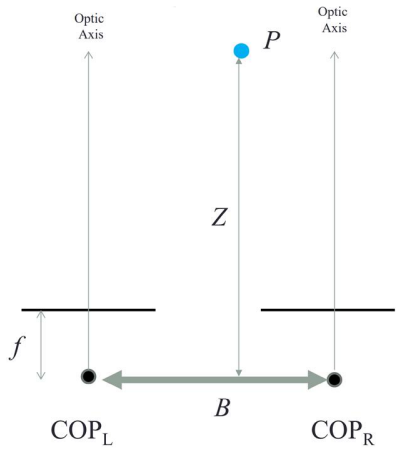


Fig. 1. Geometry for a simple stereo system. [2]

Geometry for a simple stereo system is given. First, assuming parallel optical axes, known camera parameters (i.e., calibrated cameras). Fig. 1 is looking down on the cameras and image planes. ***B*** is the Baseline, ***f*** is the focal length, and point ***P*** is distance ***Z*** in camera coordinate systems.

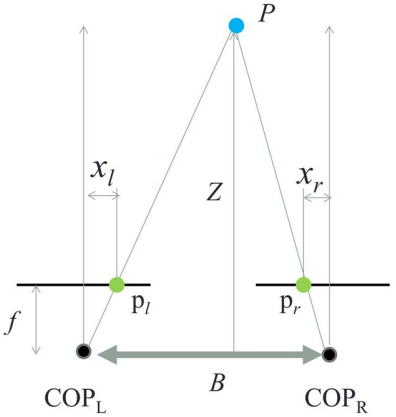


Fig. 2. Geometry for a simple stereo system when capturing photos. [2]

When capturing photo with two cameras simultaneously, point ***P*** projects into left and right images, as shown in Fig. 2. The distance which the stereo system caused is positive in the left image and negative in the right. Using two similar triangles (p_l, ***P***, p_r) and (COP_L, ***P***, COP_R), we can get the following formula:

$$\frac{B-x_l+x_r}{Z-f}=\frac{B}{Z} \tag{1}$$

Then, the distance $\boldsymbol{Z}$ of point $\boldsymbol{P}$ can be calculated by:

$$\boldsymbol{Z} = \boldsymbol{f}\frac{\boldsymbol{B}}{x_l - x_r} \quad (2)$$

where the x_l-x_r is the disparity.

C. Points Matching

We have explained how to implement the disparity computing given a pair of matching points in rectified images of the same scene captured from two different viewpoints. Nevertheless, we still have no idea about the scheme of matching points. In this section, we will illustrate the method briefly. Point matching has two fundamental methods, including appearance-based matching and feature-based matching. In this task, we will focus on appearance-based point matching, whose simple steps are discussed below:

Firstly, we assume that corresponding points in two images have image patches that look almost identical, which demands the following promises: minimal geometric distortion between images, Lambertian reflectance, and no occlusion. Fortunately, those promises are reasonable for stereo cameras with a small baseline distance..

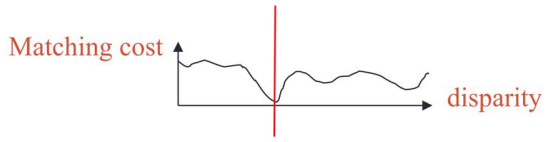


Fig. 3. Diagrammatic sketch for appearance-based matching. [2]

Secondly, for each pixel/window in the left image, we compare it with every pixel/window on the same epipolar line in the right image to pick a position with minimum match cost. Eventually, we can suppose the window having the minimum matching cost, as shown in Fig. 3, is the reasonable disparity.

II. Disparity Computing

With the discussion in the above section, we can obtain the basic numerical understanding that the depth of a point is inversely proportional to the disparity, i.e., from the depth estimation equation, we have $\boldsymbol{Z} \propto (x_l\text{-}x_r)^{-1}$. As the disparity x_l-x_r increases, $\boldsymbol{Z}$ decreases; for lower disparity x_l-x_r, we would have higher $\boldsymbol{Z}$. The analysis is intuitive if you hold your index finger near your eyes and alternate seeing from your left and right eyes. You will notice that your finger jumps a lot in your view compared to other distant objects.

Since we only need to get the disparity map in this task, it is not essential to know whether $\boldsymbol{B}$ and $\boldsymbol{f}$ are known, as long as they are constant. In this task, we will directly use the obtained disparity to draw the image instead of converting it to depth, which will not make any variance to the image display. According to the description in the previous section, the most crucial step in point matching is to compare the match cost of images in two windows to determine whether they are corresponding points. Therefore, the design of the cost function is essential.

A. Cost Function

In the lecture, Sum-of-Squared Difference (SSD), as the cost function has been introduced. Thus, we utilized SSD to compute the cost in this task initially, which have the following formula:

$$\underset{(x,y)}{\operatorname{argmax}} \sum_{u=-N}^{N} \sum_{v=-N}^{N} [I(x+u, y+v) - g(u,v)]^2 \quad (3)$$

where $g(u,v)$ is the point to be matched in the one image, and $I(x+u, y+v)$ represents the block in another image.

B. Searching Procedure [3]

Input: Left image and right image (from perfectly aligned cameras) of width $\boldsymbol{w}$ and height $\boldsymbol{h}$, **block size** in pixels, and **search block size**.

Block size refers to the neighborhood size we select to compare pixels from the left and right images, specified as the number of pixels in height and width. An example block is shown as a white box in both left and right images in Fig. 4.

Search block size refers to a rectangle (black one on the right image) in which we will search for the best matching block. To get the corresponding image region for the selected block from the left image, we move the smaller white rectangle to the left in the black rectangle, as shown in the third image in Fig. 4.

Output: Disparity map of width $\boldsymbol{w}$ and height $\boldsymbol{h}$.

Fig. 4. Sketch for the searching procedure. [4]

The searching method will be implemented in the following steps:

i. For a pixel in the left image, select the pixels in its neighborhood specified as block size from the left image.

ii. Compute similarity score by comparing each block from the left image and each block selected from the search block in the right image. Slide block on the right image by one pixel within the search block.

iii. Find the highest pixel similarity score from the previous step. For the block with the highest similarity, return the pixel location at the center of the block as the best matching pixel. If x_l is the column index of the left pixel, and the highest similarity score was obtained for a block on the right image whose center pixel has column index x_r, we will note the disparity value of $|x_l\text{-}x_r|$ for the location of the left image pixel.

iv. Repeat the matching process for each pixel in the left image and note all the disparity values for the left image pixel index.

The core part of code will be given in the *Appendix*.

C. *Results*

We use different block size and search block size to repeat the experiment on the two pictures given, shown in Fig. 5.

Fig. 5. Input images.

The results are shown in Fig. 6 and Fig .7. The outcomes are listed and compared in the Appendix for an entire display. We only extract a part of the result in this part for comparative analysis.

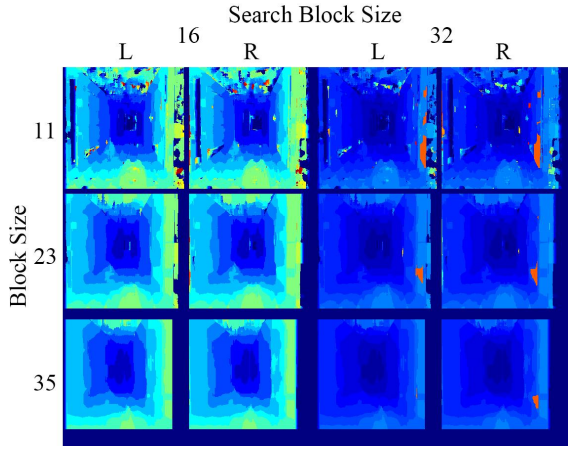


Fig. 6. Left and right image results with SSD for corridor. (Partly)

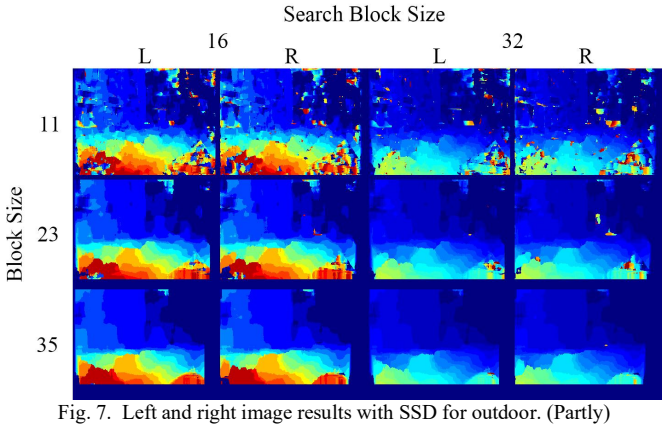


Fig. 7. Left and right image results with SSD for outdoor. (Partly)

D. *Discussion*

Among the results listed, we can do some analysis:

i. The programs we write work effectively with Disparity Computing with reasonable outputs.

ii. With the same block size, changes in search block size do not cause drastic changes in the output image. In general, disparity maps with larger search block sizes have lower 'contrast,' due to the increasing search block size, which means the widening range of disparity obtained through point matching. When matching errors, it may be larger than the rest of the disparity in the image. (For example, when search block size = 32, some red error maxima occurred in the result.) After normalization, errors will narrow the disparity scale on the rest parts, which reduces the 'contrast' of the map.

iii. Under the same search block size, changes in Block size result in more significant distinctions in output results. When the block size is too small, the output shows enough detail but much noise. (For example, only the image with Block size = 11 can clearly see cones and spheres in the corridor image, with considerable noise.) On the other hand, the output noise is suppressed when the block size is too large, but the image will be too smooth to show details. These varieties can be simply explained by the meaning of block size since a block is a convolution-like kernel whose size directly influences the response of the output, which is intuitive for us.

Moreover, we can obtain the most reasonable parameters of block size and search block size, which are 23 and 16, through the output image comparations.

III. Improvements

A. *Pre-processing*

Since all operations in our algorithm directly utilize the two input images, the quality of the original pictures will straight affect the output result. If we can filter the noise of the input in pre-processing, the program's effect will be improved.

We already know (from the course slides) that SSD-based search is not robust: one reason is that there may be complicated and continuous transition light, as shown in Fig. 8. When we perform point matching, complex light on a plane is likely to impact the matching results significantly.

Fig. 8. Example of complicated and continuous transition light.

Therefore, if we can extract the edge of the plane in the original image and enhance its presentation, the above influence will be diminished. We considered Sobel gradient filtering, learned in the spatial image filtering course, as an ideal method. Sobel differential operators attempt to approximate the gradient at a pixel via Sobel masks. Thresholding the gradient produces edge pixels, as shown in Fig. 9. We notice that the ‘gradient’ image after the Sobel mask eliminates the shadow on the wall in the input image.

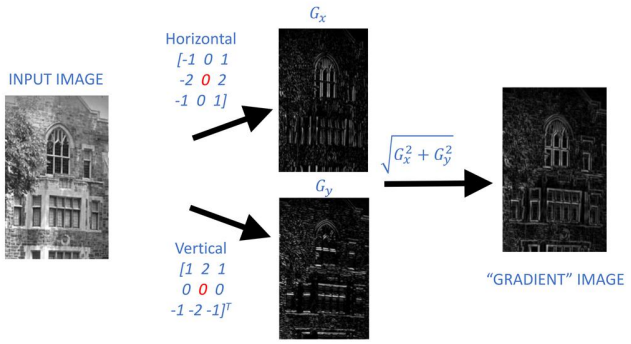


Fig. 9. Sketch for the procedure of Sobel gradient filtering.

Consequently, we performed Sobel filtering as the pre-processing on the image before point matching to obtain more effective features in the original image. The outputs of the pre-processing are shown in Fig. 10. The code will be shown in the Appendix.

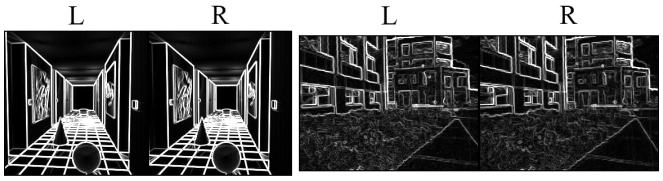


Fig. 10. Output of Sobel filter with input images.

B. *Post-processing*

In the above program, we will take left and right images as the reference to carry out two symmetrical calculation processes, respectively, resulting in two output pictures. However, in many third-party libraries, such as OpenCV, the result of disparity computing has only one image. To solve this discrepancy, we refer to the idea of *cv2.StereoBM()*, that is, two images will perform cross-validation first, and only the point pair obtained by both sides can be counted as effective. When restoring the algorithm, we adopt a relatively simple method by recalculating the two disparity maps: if the matching points do not coincide and the disparity and exceeds a certain threshold, the disparity of this position is regarded as pairing failure, and a result of 0 is given.

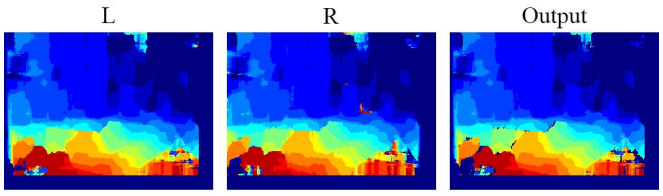


Fig. 11. Output of post-processing with SSD-16-23.

In this algorithm, we take the left image as the benchmark. After such post-processing, we can combine the two output images and eliminate the points with inaccurate pairing. An example of merging is given in Fig. 11. The code will be shown in the Appendix.

It can be seen from the results that post-processing can eliminate some wrong results in the original output, such as the red area in the distant parts on the R-image. In addition, some empty edge regions appear in the nearby parts of the fusion result, where were pairing failed areas. Thus, it is unreasonable to color them.

There are also these empty areas in the output results of OpenCV. We will discuss the comparison with OpenCV later.

C. *New Procedure*

So far, we have obtained a new procedure, including pre-processing and post-processing. We schematically rearrange the process and show it in Fig. 12.

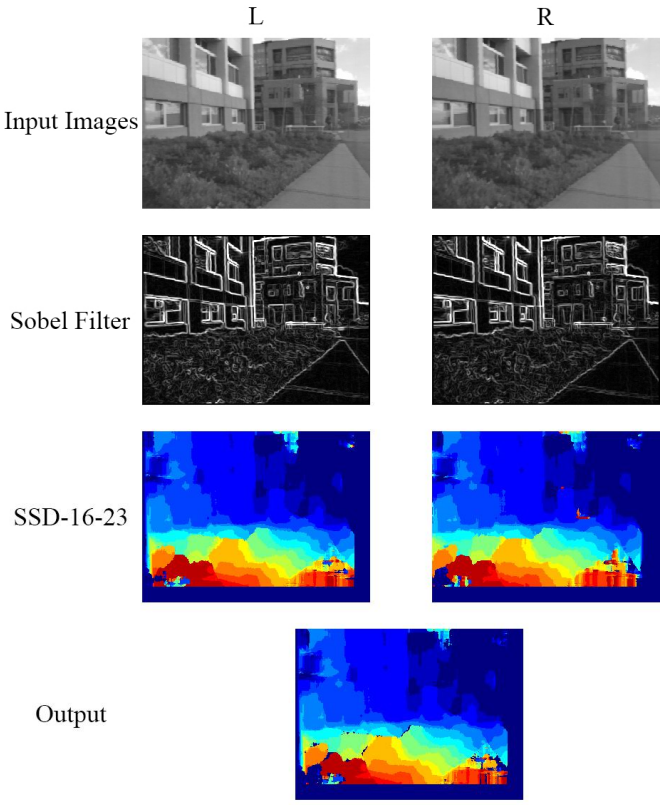


Fig. 12. Sketch for the new procedure.

D. *Cost Function - SAD/NCC*

The core of disparity computing in this task is matching points, more precisely, designing cost functions. In previous experiments, we directly used the SSD algorithm introduced in slides. However, we need to identify whether there is a better and suitable cost function than SSD.

In the previous reference to *cv2.StereoBM()*, we discovered that it does not use the SSD algorithm but the SAD algorithm. Sum-of-Absolute-Differences (SAD) utilize the l_1-norm as the distance function instead of the l_2-norm of SSD, and its equation is as follows:

$$\underset{(x,y)}{\operatorname{argmax}} \sum_{u=-N}^{N} \sum_{v=-N}^{N} |I(x+u, y+v) - g(u,v)| \quad (4)$$

In addition, we find an algorithm widely used in appearance-based point matching, named Normalization-Cross-Correlation (NCC), and its equation is as follows:

$$NCC(p,d) = \frac{\sum_{(x,y)}[I_1(x,y)-\bar{I_1}(p_x,p_y)]\cdot[I_2(x+d,y)-\bar{I_2}(p_x+d,p_y)]}{\sqrt{\sum_{(x,y)}[I_1(x,y)-\bar{I_1}(p_x,p_y)]^2\cdot\sum_{(x,y)}[I_2(x+d,y)-\bar{I_2}(p_x+d,p_y)]^2}} \quad (4)$$

where the head above I means the average value of the pixel in the block. When NCC = -1, it means that the two matching windows are entirely irrelevant. On the contrary, if NCC=1, the correlation between the two matching windows is very high.

In order to compare the performance of SSD and the two new algorithms, SAD and NCC, we write the new algorithm into the code and repeat the experiment with different parameters, while search block size is 16. Part of the results is shown in Fig. 13.

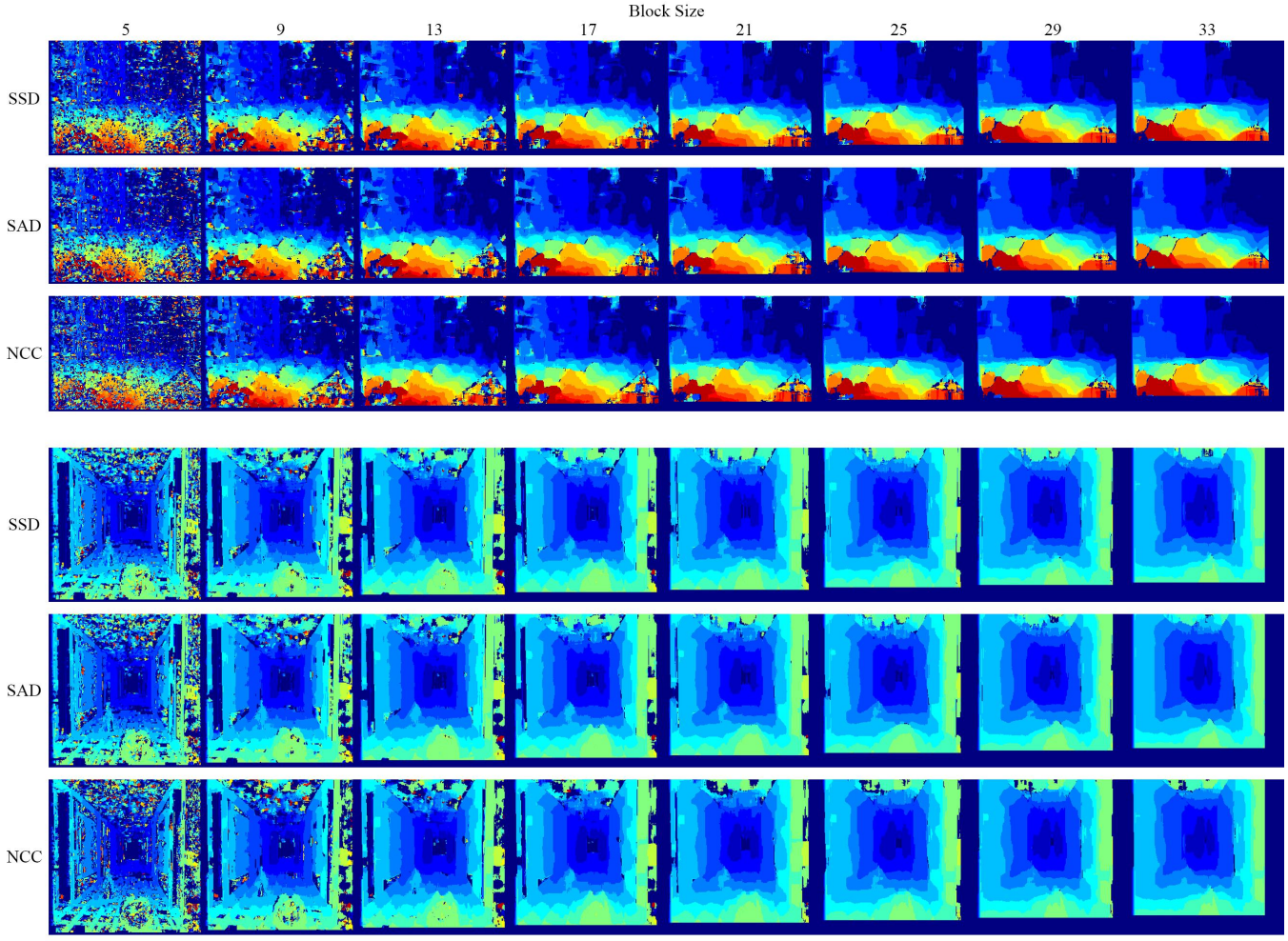


Fig. 11. Output of post-processing with SSD-16-23.

From the output results, we can observe that there is little distinction between the three algorithms. Specifically, the gap between SSD and SAD is minimal; only careful observation can notice the subtle difference. The output of NCC is more accurate than the above two, which is reflected in the ability to restore the object's shape, for instance, the rectangle of the corridor and the straight lines of the building. When the output images of SSD and SAD have been distorted, NCC still retains some level of the original pattern contours.

However, there are three parts, the top of the corridor, the bushes in the lower-left corner of the street, and the path on the right side of the street, having problems. These three areas cannot be recognized by the algorithm. We speculate that this is because SAD-like algorithms cannot calculate regions with similar lightness, like these three regions.

Moreover, we must notice the complexity of the computing in the three algorithms. For SSD and SAD, the difference between them lies in multiplication and addition when calculating different norms. We should realize that the complexity of one-step multiplication is often several times that of addition on the CPU-register level, which cannot be ignored even after optimization. Therefore, SAD must be faster than SSD. We guess therefore OpenCV chose SAD instead of SSD.

For NCC, matters have become more complicated. The calculation process of NCC requires repeated references to a matrix with block size, which needs the cache to repeatedly read

a data string that cannot hit and will significantly increase the running time of the program. The experimental results also verify our inference. Fig. 14 shows the running time of the program for different matching methods and block sizes. We should note that since the program we write has not been optimized, the running time will be much longer than an optimized function library, such as OpenCV. However, as we analyzed above, the relationship between the three methods will not change.

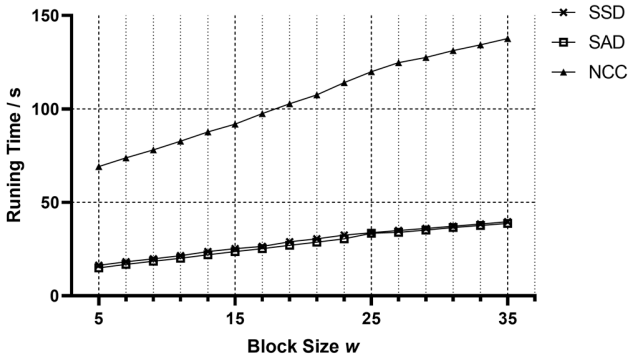


Fig. 14. Running time for different method.

It can be observed that SSD is slightly slower than SAD, while NCC takes much more time than SAD, requiring us to consider whether it is worth such consumption in exchange for slight output effect improvement. We need to do the trade-off between the results and the running speed.

E. Cost Function - SGM/SGBM

In addition, we learned about another algorithm implemented in OpenCV, Semi-Global-Block-Matching (SGBM), which is a semi-global matching algorithm used to calculate the disparity in binocular vision. The relationship between SGM and SGBM are in Fig. 15. Its core method is Semi-Global-Matching (SGM) algorithm, which formula is shown below:

$$E(D) = \sum_{p}(C(p, D_q) + \sum_{q \in N_p} P_1 T[|D_p - D_q| = 1] + \sum_{q \in N_p}(P_2 T[|D_p - D_q|])$$

where P_1 is related to the smoothing of the disparity map, and P_2 is related to the edge of the disparity map.

SGM is essentially a cost aggregation algorithm. In order to minimize the global energy function as the global stereo matching algorithm, more or all pixels of the whole image need to participate in the constraints of the current pixel.

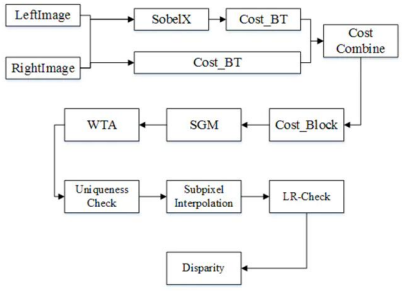


Fig. 15. Relationship between SGM and SGBM

The author came up with the idea of multi-path constraint aggregation, which simply means that the cost aggregation process of the current pixel is affected by all pixels in multiple directions (or paths), shown in Fig. 16. The more directions (generally speaking, 8-16 directions are better), the more pixels affect the neighborhood. It not only ensures the constraint of global pixels but also computes without establishing the global minimum energy function, avoiding complex operators; thus, it is called a semi-global algorithm.

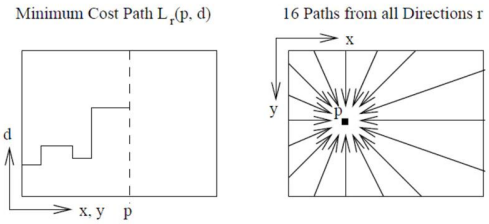


Fig. 16. Sketch for SGM

Since the workload of manually re-implementing the SGBM is too large, we will directly call the *cv2.StereoSGBM()* instead of re-write it. Moreover, we also call the *cv2.StereoBM()* which mentioned above. Their outputs are shown in Fig. 17.

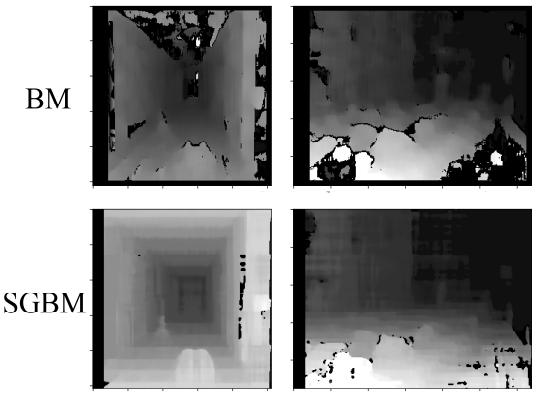


Fig. 17. Output of OpenCV functions.

Firstly, we can observe that the output result of BM is consistent with the SAD we implemented before, which shows that the algorithm we completed is accurate. Second, the problem parts in the BM algorithm have also existed in our previous results, that is, the top of the corridor, the bushes in the lower-left corner of the street, and the path on the right side of the street. Third, we can see that the SGBM output fixes the above problem, showing that the semi-global algorithm can effectively improve the local algorithm. Last but not least, during the call, we observe that the calculation time of SGBM is significantly longer than that of BM. Therefore, the trade-off between computational complexity and output effect is still worth considering.

IV. Code Availability

The main functions related to assignment are given in *Appendix*. For the file containing all the source-code, see another file in the compressed package.

REFERENCES

[1] Forsyth, D., & Ponce, J. (2003). Computer vision: A modern approach. Upper Saddle River, N.J: Prentice Hall.

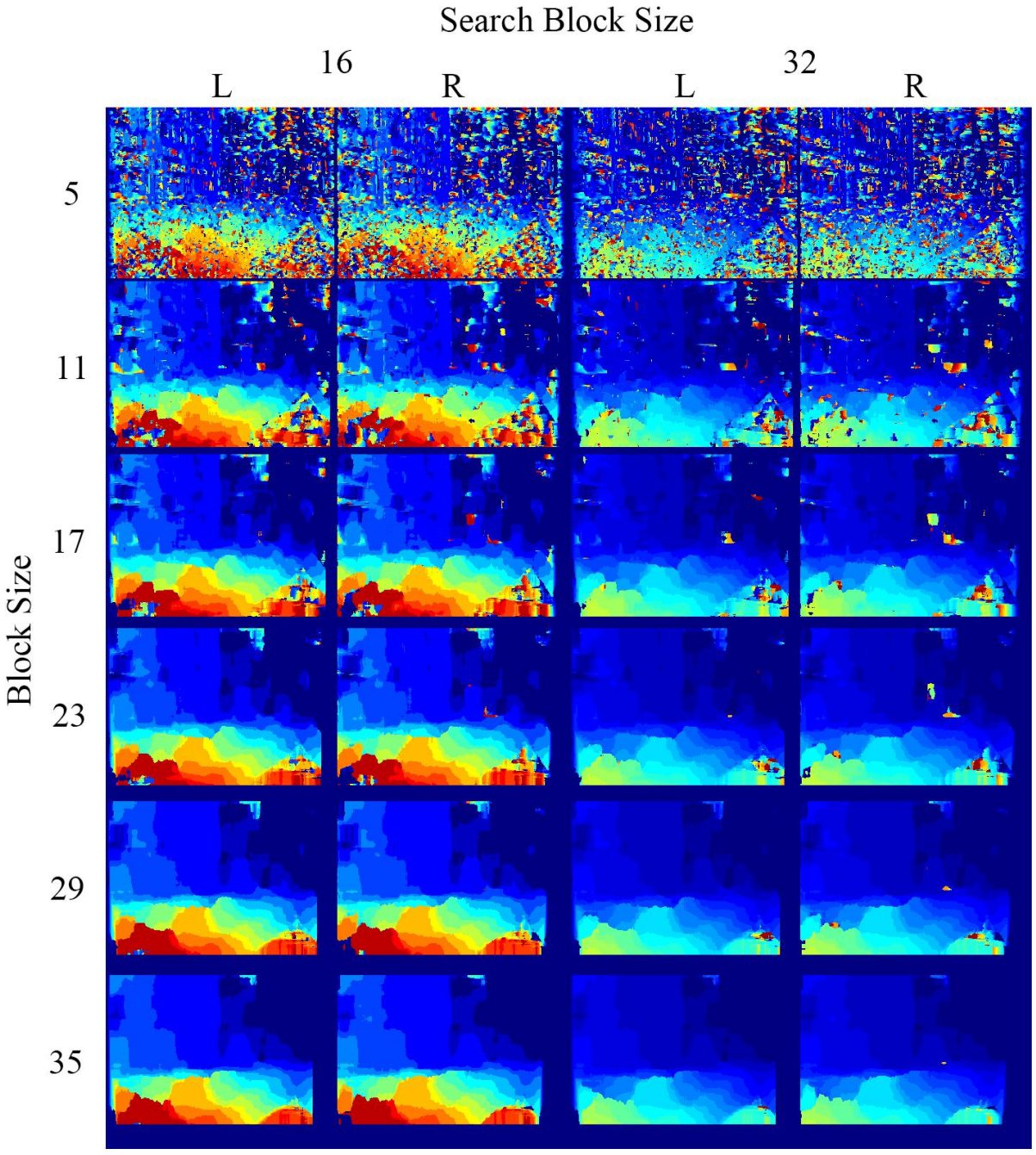
[2] source: CS 4495 Computer Vision – A. Bobick

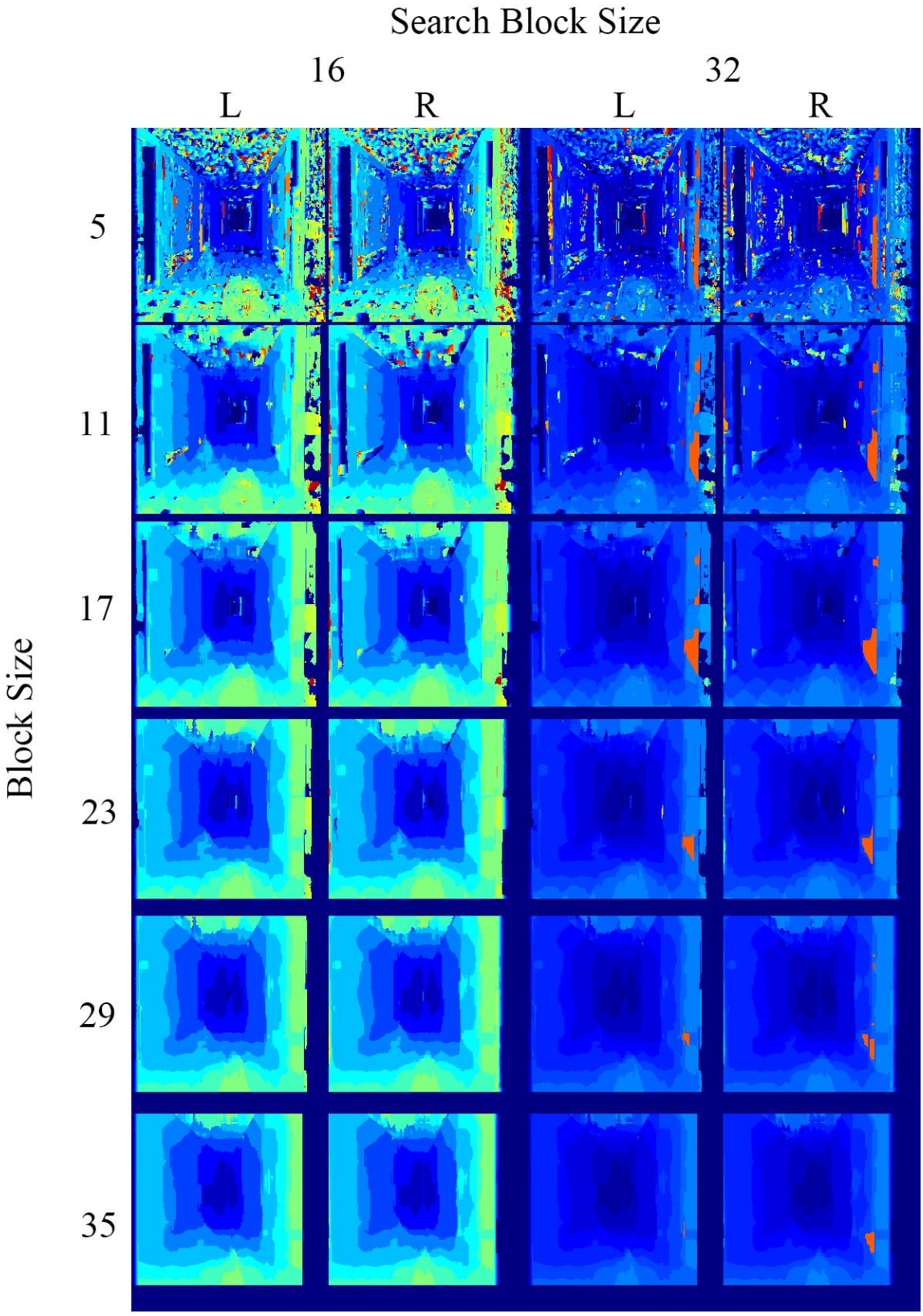
[3] https://pramodatre.github.io/2020/05/17/stereo-vision-exploration/#fn:1

[4] Middlebury Stereo Datasets

V. APPENDIX

A. Left and right image results with SSD




Search Block Size
16
32
L
R
L
R
Block Size
5
11
17
23
29
35

B. Main function source-code

```
def CalcSAD(target_block, crspd_block):
    return (sum(sum(abs(target_block - crspd_block))))


def CalcSSD(target_block, crspd_block):
    return (sum(sum(abs(target_block - crspd_block)**2)))


def CalcNCC(target_block, crspd_block):
    diff_tar = target_block - np.mean(target_block)
    diff_crs = crspd_block - np.mean(crspd_block)
    NCC_up = sum(sum(diff_tar * diff_crs))
    NCC_down = math.sqrt(sum(sum(diff_tar**2)) *
sum(sum(diff_crs**2)))
    return (NCC_up / NCC_down)

def CalcDisparity(gray_left, gray_right, num_disparity,
block_size,
                  matching_method, direction):

    #Check the input direction
    target_image, crspd_iamge = AssignDirct(gray_left,
gray_right, direction)

    #Check the input images' shape
    if (target_image.shape == crspd_iamge.shape):
        height, width = target_image.shape
    else:
        print('Error! Two Image have different shape!')
        exit(0)

    disparity_matrix = numpy.zeros((height, width),
dtype=numpy.float32)
    half_block = block_size // 2

    #Load the range for kernel
    for i in range(half_block, height - half_block):
        print("%d%% " % (i * 100 // height), end=' ',
flush=True)

        for j in range(half_block, width - half_block):
            target_block = target_image[i - half_block:i +
half_block,
                                        j - half_block:j +
half_block]

            bar, dist = AssignBar(matching_method)
            edge_value = CalcDEdge(j, half_block, width,
direction)

            for d in range(0, min(edge_value,
num_disparity)):
                up_edge, below_dege, left_edge, right_edge =
CalcBlockEdge(
                    i, j, half_block, d, direction)
                crspd_block = crspd_iamge[up_edge:below_dege,

left_edge:right_edge]

                Val = MatchImage(target_block, crspd_block,
matching_method)
                bar, dist = UpdateVal(bar, dist, Val, d,
matching_method)

            disparity_matrix[i - half_block, j - half_block]
= dist
    print('100%')
    return disparity_matrix
```

```
def SobelFilter(image):
     height, width = image.shape
     out_image = numpy.zeros((height, width))

     table_x = numpy.array(([-1, -2, -1], [0, 0, 0], [1, 2,
1]))
     table_y = numpy.array(([-1, 0, 1], [-2, 0, 2], [-1, 0,
1]))

     for y in range(2, width - 2):
         for x in range(2, height - 2):
             cx, cy = 0, 0
             for offset_y in range(0, 3):
                 for offset_x in range(0, 3):
                     pix = image[x + offset_x - 1, y +
offset_y - 1]
                     if offset_x != 1:
                         cx += pix * table_x[offset_x,
offset_y]
                     if offset_y != 1:
                         cy += pix * table_y[offset_x,
offset_y]
             out_pix = math.sqrt(cx**2 + cy**2)
             out_image[x, y] = out_pix if out_pix > 0 else 0
     numpy.putmask(out_image, out_image > 255, 255)
     return out_image

def Postprocessing(disp_left, disp_right):
    height, width = disp_left.shape
    out_image = disp_left

    for h in range(1, height - 1):
        for w in range(1, width - 1):
            left = int(disp_left[h, w])
            if w - left > 0:
                right = int(disp_right[h, w - left])
                dispDiff = left - right
                if dispDiff < 0:
                    dispDiff = -dispDiff
                elif dispDiff > 1:
                    out_image[h, w] = 0
    return out_image
```