AI6121: Computer Vision

Assignment 2
Stereo Vision

Ron Kow Kheng Hui

ID: G1903451J

1 November 2021

School of Computer Science and Engineering

Nanyang Technological University

Contents

1 Introduction

In this assignment, we study a pixel matching method used to estimate the 3D depth of objects and things captured in 2D images. A key parameter in the method is disparity, which is inversely proportional to the 3D depth. Our goal is to develop an algorithm to compute a disparity map of the captured scene. The disparity map shows the different levels of 3D depth. This report is organized as follows:

- In Section 2 (Assignment Task 1), we describe the method of computing the disparity between two corresponding pixels in a pair of rectified images of the same scene captured from two different viewpoints.
- In Section 3 (Assignment Task 2), we present our implementation of an algorithm to compute disparity and to generate the disparity map for a pair of rectified images.
- In Section 4 (Assignment Tasks 3 and 4), we present the disparity maps obtained by applying our algorithm to the two given pairs of images. We compare and discuss the results for different parameter values. We also present the improved disparity maps obtained after applying a median filter.
- In Section 5 (Assignment Task 4), we discuss the factors affecting the accuracy of disparity map computation and some possible improvements to our algorithm.
- Lastly, in Section 6, we summarize and conclude the work presented in this report.

2 Definitions and Methods

In this section, we describe the method of computing the disparity map for a pair of rectified images of the same scene captured from two different viewpoints.

2.1 Relationship Between Depth and Disparity

Stereo matching refers to the process of capturing two or more images of the same scene from different viewpoints (i.e., camera positions), building a 3D model of the scene by matching pixels in the images, and computing 3D depths of points in the scene from their 2D positions in the images [3].

When two images of the same scene are captured from two different viewpoints, a particular point in the scene will be at different positions in the images. If there is a horizontal translation in position, the number of pixels translated horizontally is called *disparity*. It can be shown by simple geometry that the disparity for a point in the scene is inversely proportional to its 3D depth (i.e., perpendicular distance from the line joining the two cameras to the point), as shown by the following equation:

$$Z = \frac{fT}{d},$$

where Z is the 3D depth, f is the focal length of the cameras measured in pixels, T is the distance between the two cameras and d is the disparity. Thus, by computing the disparity associated with a point in the scene, we can estimate the 3D depth of the point.

2.2 Computation of Disparity and Disparity Map

The fundamental problem in stereo matching is that of pixel searching and matching: For two images of the same scene, how do we match correctly a pixel in one image with its corresponding pixel in the other image? To solve this problem, for a given pixel in the left (or right) image, we compute a range of possible candidate pixels in the right (or left) image. Then we select the candidate pixel that is the most likely match based on a cost criterion.

The first step is to *rectify* the two images so that the range of possible locations all lie on a horizontal line, called the *epipolar line*, passing through the left image and right image. In other words, a pixel in one image projects to a pixel on the epipolar line in the other image. The search space is now reduced to the epipolar line. For a pixel at position (x_0, y_0) in the left image, if the matching pixel in the right image is at position (x_1, y_0), then the disparity is the horizontal distance $|x_0 - x_1|$. If we plot the disparities for all points in the scene, we obtain a *disparity map*, which is an image showing the different levels of 3D depth in the scene.

Note that disparity for points at infinity is 0, because the pixel shift in the images will be negligible. In the disparity map, points at infinity will be black. For points very close to the cameras, the pixel shift will be large. In the disparity map, points close to the camera will be light-colored.

2.3 Window-Based Method

We now describe a *local* method (also called *window-based* method) to compute disparity, so-called because disparity computation depends only on pixel intensity values within a square or rectangular window (also called a *support window*).

If an image point matches with a point in another image of the same scene, their pixel intensities should be the same. In other words, the difference in intensity values will be zero. However, for a given point in one image, there are likely to be multiple points in the other image with the same pixel intensity. Furthermore, there could be some differences in colors between the two images for two matching points. Thus, matching individual pixel intensities will not produce accurate disparity maps. Instead, we compare two square or rectangular regions (i.e., the support windows) of the same size in which the pixel of interest lies at the center of the window. Then we compute a cost measure between the two windows. The simplest cost measure is to compute an aggregate of the difference in intensity values between corresponding pixels in the two windows. If two windows match, the distribution of pixel intensities between them should be identical. The best matching pair of windows is the one with the least cost between them.

To illustrate, Figure 1 shows two rectified images with a red epipolar line running through them. The blue windows in the left and right images are at the same position along the epipolar line. However the windows do not match. To find a target window in the right image that matches with the blue window in the left image, we translate the target window in the right image one pixel at a time in the negative direction. For each target window, we compute the cost. In practice, we use a *maximum disparity* parameter to limit the number of pixels by which we translate the target window. This parameter gives the number of candidate windows.

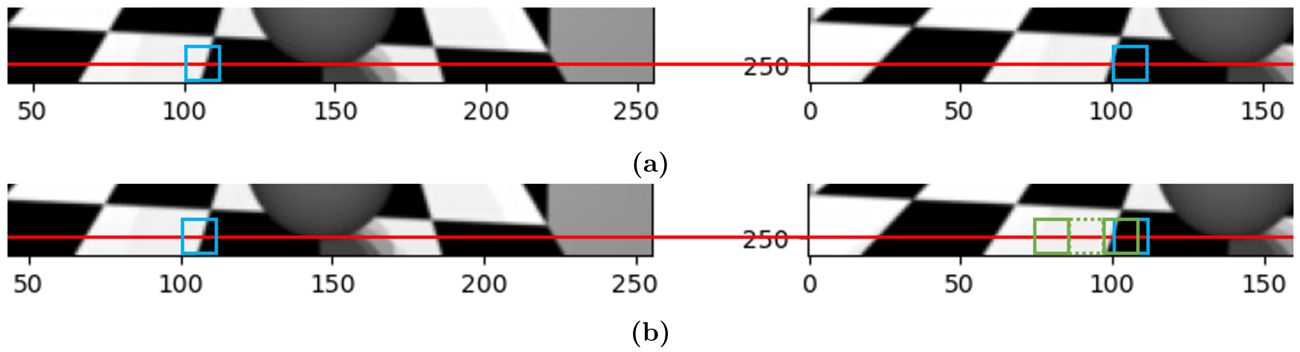


Figure 1: (a) Same x-coordinate positions in left and right images showing different parts of the scene, (b) Translating a window (green square) pixel by pixel in the negative direction from the initial position (blue square) along the epipolar line

3 Implementation

In this section, we present our implementation of the window-based method to compute disparity and the disparity map for a pair of rectified images.

3.1 Four-Step Taxonomy

Scharstein and Szeliski [2] proposed a general taxonomy for stereo matching algorithms consisting of the following four steps:

1. Matching cost computation
2. Cost aggregation
3. Disparity computation and optimization
4. Disparity refinement

The authors referred to these four steps as "building blocks" from which an algorithm can be constructed. The sequence of steps depends on the design of the algorithm. Steps may also be combined or omitted. With reference to the four-step process, the following is the window-based (using a square window) algorithm we developed:

1. **Matching cost computation**: We use *absolute difference* (AD) of intensities to compute the cost at a given point and disparity. This is defined as:

$$AD(x, y, d) = |I_L(u, v) - I_R(u - d, v)|$$

where d is the disparity, u and v are coordinates within the support window, and I_L and I_R are the left and right intensity values.

2. **Cost aggregation**: We use a fixed-sized $w \times w$ square support window. The aggregate cost C at a given point and disparity is computed by summing all the costs over the support window:

$$C(x, y, d) = \sum_{(u,v) \in (x,y)} AD(x, y, d)$$

3. **Disparity computation and optimization**: We select the disparity d_s associated with the minimum aggregate cost from the set of candidate aggregate costs (each associated with one candidate window):

$$d_s(x, y) = \arg\min_{d \in D} C(x, y, d)$$

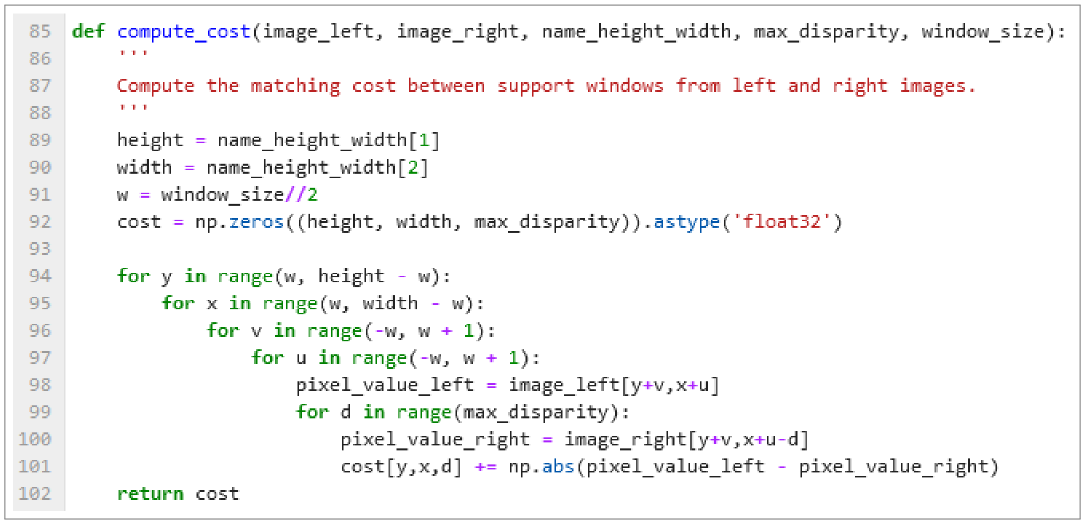
   The number of candidate windows is D, which is the maximum disparity parameter.
4. **Disparity refinement**: The disparity map is plotted using disparities associated with least aggregate costs. Along the four borders (each of width $\frac{w}{2}$) on the disparity map (where our algorithm is unable to compute the disparities), we use the disparities of pixels adjacent to the borders. For further refinement, we also apply a median filter to the disparity map.

3.2 Source Code

We now present our Python source code. The function **load_image** loads a pair of left and right images from the directory and converts them to a Numpy array (using OpenCV).

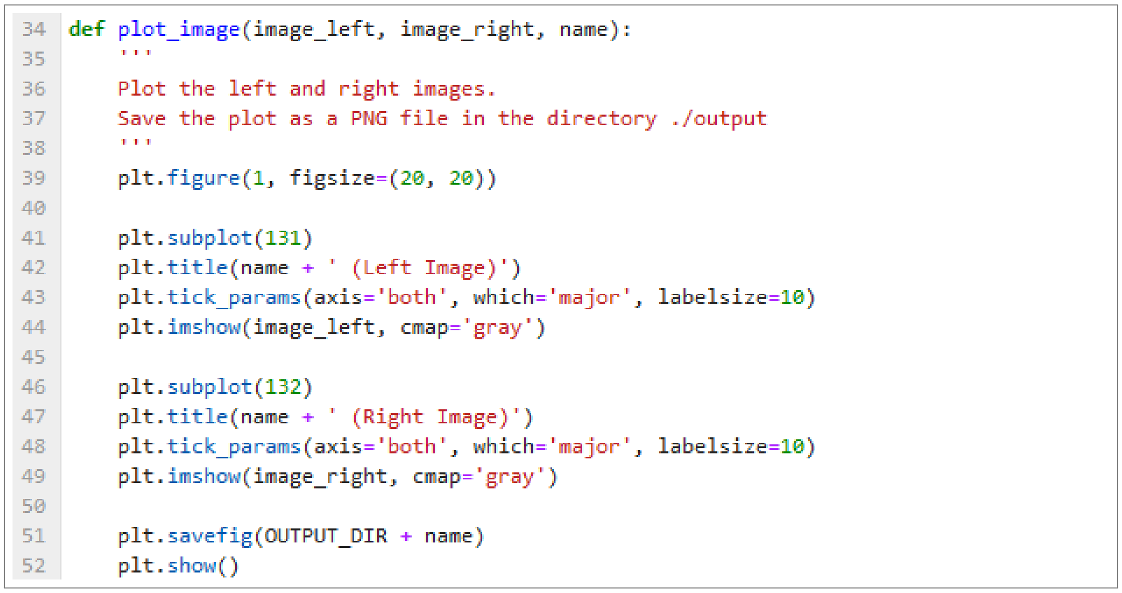
```python
def load_image(name):
    '''
    Use cv2 to read the left and right images from the directory.
    '''
    image_left_path = os.path.join(INPUT_DIR, name + LEFT_EXT)
    image_left = cv2.imread(image_left_path, cv2.IMREAD_GRAYSCALE)
    image_left = image_left.astype('float32')

    image_right_path = os.path.join(INPUT_DIR, name + RIGHT_EXT)
    image_right = cv2.imread(image_right_path, cv2.IMREAD_GRAYSCALE)
    image_right = image_right.astype('float32')

    height = image_left.shape[0]
    width = image_left.shape[1]
    name_height_width = (name, height, width)

    return image_left, image_right, name_height_width
```

The function **compute_cost** computes the aggregate cost for each support window and stores all the costs for the entire image in a Numpy array.



```python
def compute_cost(image_left, image_right, name_height_width, max_disparity, window_size):
    '''
    Compute the matching cost between support windows from left and right images.
    '''
    height = name_height_width[1]
    width = name_height_width[2]
    w = window_size//2
    cost = np.zeros((height, width, max_disparity)).astype('float32')

    for y in range(w, height - w):
        for x in range(w, width - w):
            for v in range(-w, w + 1):
                for u in range(-w, w + 1):
                    pixel_value_left = image_left[y+v,x+u]
                    for d in range(max_disparity):
                        pixel_value_right = image_right[y+v,x+u-d]
                        cost[y,x,d] += np.abs(pixel_value_left - pixel_value_right)
    return cost
```

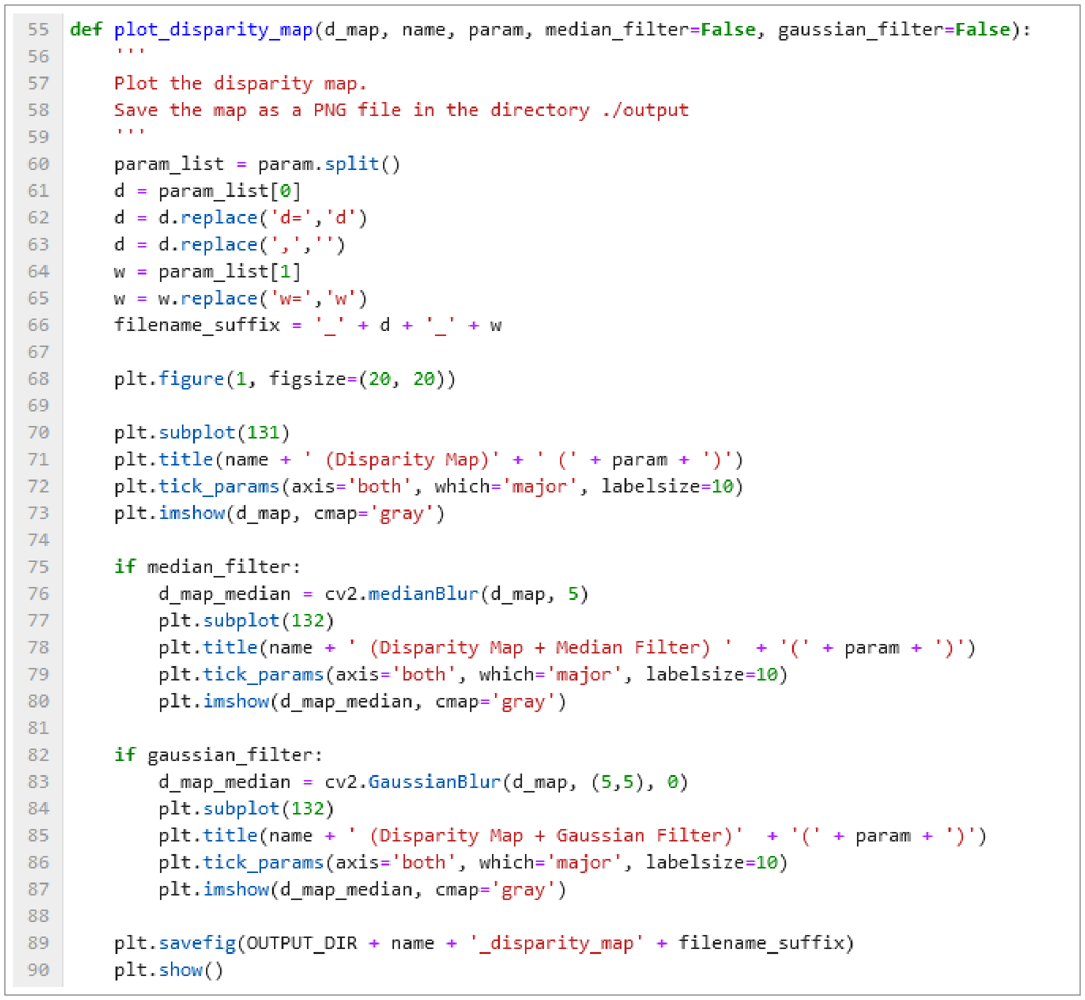
The function **plot_image** plots a pair of images and save the plots to a file.



```python
def plot_image(image_left, image_right, name):
    '''
    Plot the left and right images.
    Save the plot as a PNG file in the directory ./output
    '''
    plt.figure(1, figsize=(20, 20))

    plt.subplot(131)
    plt.title(name + ' (Left Image)')
    plt.tick_params(axis='both', which='major', labelsize=10)
    plt.imshow(image_left, cmap='gray')

    plt.subplot(132)
    plt.title(name + ' (Right Image)')
    plt.tick_params(axis='both', which='major', labelsize=10)
    plt.imshow(image_right, cmap='gray')

    plt.savefig(OUTPUT_DIR + name)
    plt.show()
```

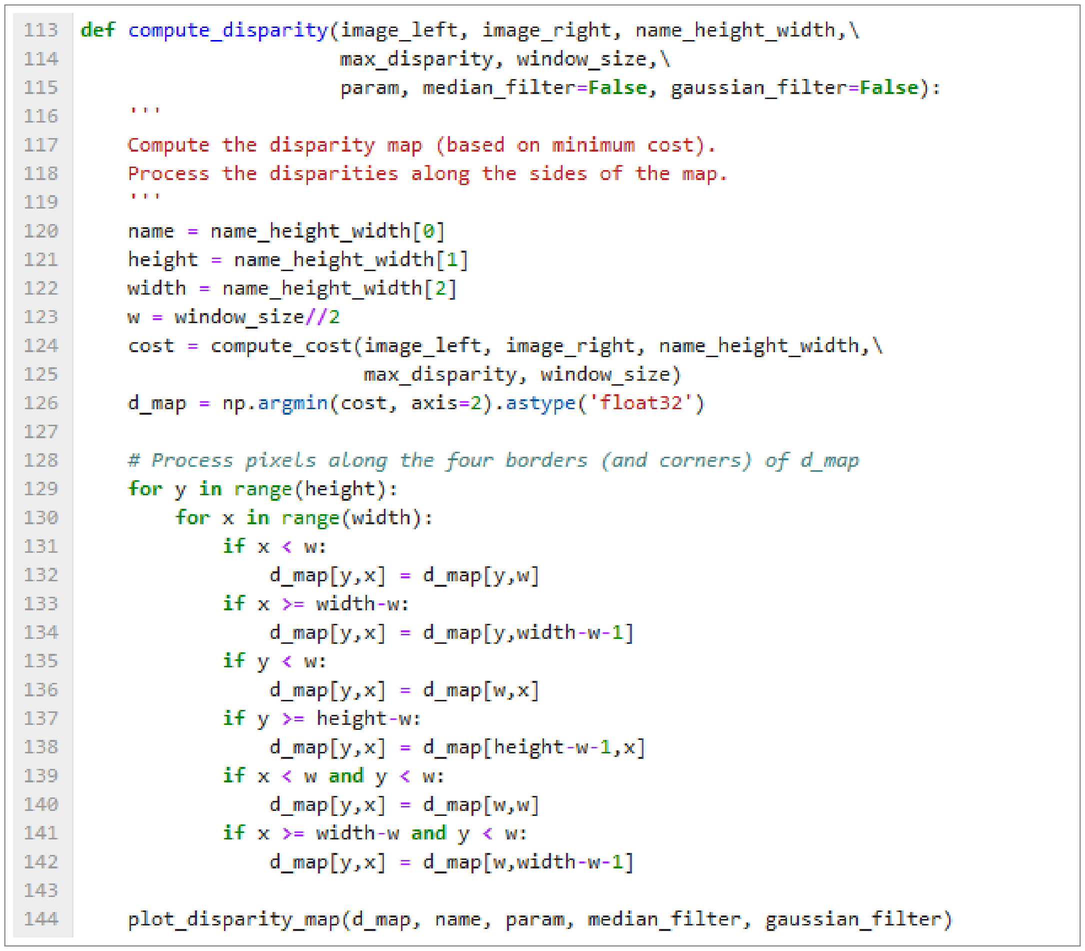
The function **plot_disparity_map** plots a disparity map and if specified, the disparity map with median filter or Gaussian filter applied. It then saves the plots to a file.

We provide options to use OpenCV methods **medianBlur** and **GaussianBlur** with a 5×5 kernel window. The median filter **medianBlur** replaces the intensity of the pixel at the center of the window with the median intensity value of all the pixels in the window. The Gaussian filter **GaussianBlur** estimates pixel values by using neighboring pixels and weights defined by a Gaussian distribution.



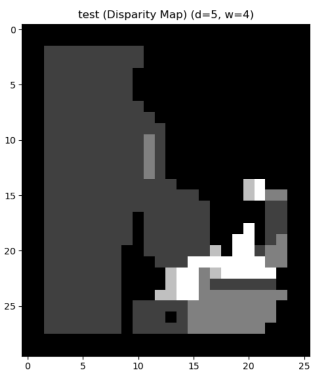
```python
def plot_disparity_map(d_map, name, param, median_filter=False, gaussian_filter=False):
    '''
    Plot the disparity map.
    Save the map as a PNG file in the directory ./output
    '''
    param_list = param.split()
    d = param_list[0]
    d = d.replace('d=','d')
    d = d.replace(',','')
    w = param_list[1]
    w = w.replace('w=','w')
    filename_suffix = '_' + d + '_' + w

    plt.figure(1, figsize=(20, 20))

    plt.subplot(131)
    plt.title(name + ' (Disparity Map)' + ' (' + param + ')')
    plt.tick_params(axis='both', which='major', labelsize=10)
    plt.imshow(d_map, cmap='gray')

    if median_filter:
        d_map_median = cv2.medianBlur(d_map, 5)
        plt.subplot(132)
        plt.title(name + ' (Disparity Map + Median Filter) '  + '(' + param + ')')
        plt.tick_params(axis='both', which='major', labelsize=10)
        plt.imshow(d_map_median, cmap='gray')

    if gaussian_filter:
        d_map_median = cv2.GaussianBlur(d_map, (5,5), 0)
        plt.subplot(132)
        plt.title(name + ' (Disparity Map + Gaussian Filter)'  + '(' + param + ')')
        plt.tick_params(axis='both', which='major', labelsize=10)
        plt.imshow(d_map_median, cmap='gray')

    plt.savefig(OUTPUT_DIR + name + '_disparity_map' + filename_suffix)
    plt.show()
```

The function **compute_disparity** calls the other functions to compute the disparity map. It selects the minimum cost from the set of aggregate costs and processes the disparities along the four borders of the disparity map.

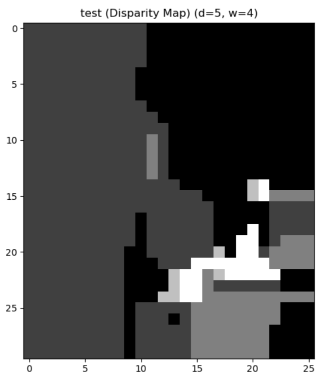


```python
def compute_disparity(image_left, image_right, name_height_width,\
                      max_disparity, window_size,\
                      param, median_filter=False, gaussian_filter=False):
    '''
    Compute the disparity map (based on minimum cost).
    Process the disparities along the sides of the map.
    '''
    name = name_height_width[0]
    height = name_height_width[1]
    width = name_height_width[2]
    w = window_size//2
    cost = compute_cost(image_left, image_right, name_height_width,\
                        max_disparity, window_size)
    d_map = np.argmin(cost, axis=2).astype('float32')

    # Process pixels along the four borders (and corners) of d_map
    for y in range(height):
        for x in range(width):
            if x < w:
                d_map[y,x] = d_map[y,w]
            if x >= width-w:
                d_map[y,x] = d_map[y,width-w-1]
            if y < w:
                d_map[y,x] = d_map[w,x]
            if y >= height-w:
                d_map[y,x] = d_map[height-w-1,x]
            if x < w and y < w:
                d_map[y,x] = d_map[w,w]
            if x >= width-w and y < w:
                d_map[y,x] = d_map[w,width-w-1]

    plot_disparity_map(d_map, name, param, median_filter, gaussian_filter)
```

Figure 2 shows the effect of processing the disparities along the four borders of the disparity map in the function **compute_disparity**. Before processing, the disparity map is surrounded by a black border whose width is half of the support window width. Our algorithm changes the values of the pixels along the four borders to the values of the pixels adjacent to the borders.



(a)



(b)

Figure 2: Disparity maps for a small test image: (a) before processing the borders, (b) after processing the borders

4 Experiments and Observations

In this section, we present the disparity maps for different parameter values in our algorithm (with and without applying a median filter) and compare the results.

4.1 Input Images

We apply our algorithm to the following pairs of rectified images shown in Figure 3. For both pairs of images, it is clear that the each point in the left image is shifted a number of pixels in the negative direction in the right image. Thus, our algorithm only needs to move the support window in the same direction for both pairs of images.

4.2 Disparity Maps

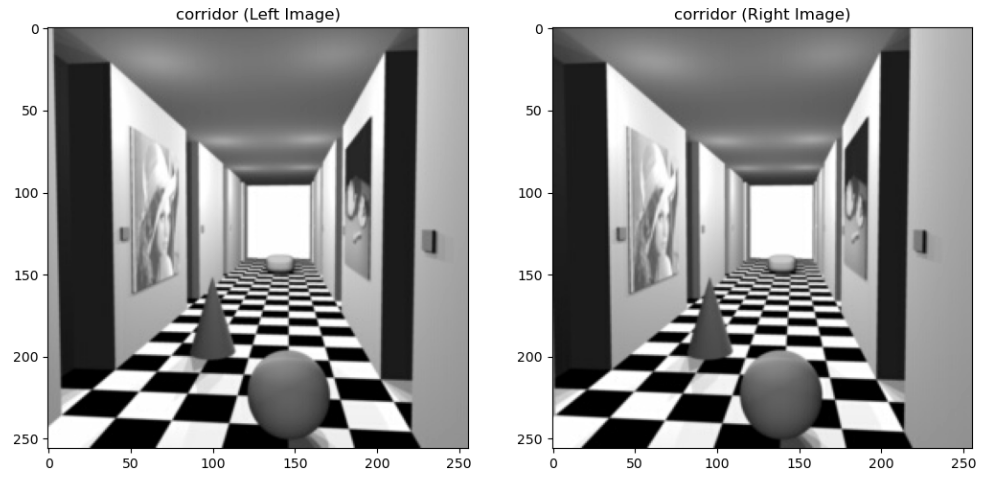
Figures 4, 5, 6 and 7 show the disparity maps obtained with and without applying a median filter for different values of the parameters maximum disparity and window size. We experimented with two parameters: maximum disparity d and square window size $w \times w$. For d, we obtained disparity maps for $d = 10, 15, 20$ and for window sizes 8×8, 12×12 and 16×16. The following are our observations.

Best results: For the *corridor* images, the best results are obtained when we set the maximum disparity to 15 (or 20) and window size to 16×16. For the *triclopsi2* images, the best results are obtained when we set the maximum disparity to 20 and window size to 16×16.

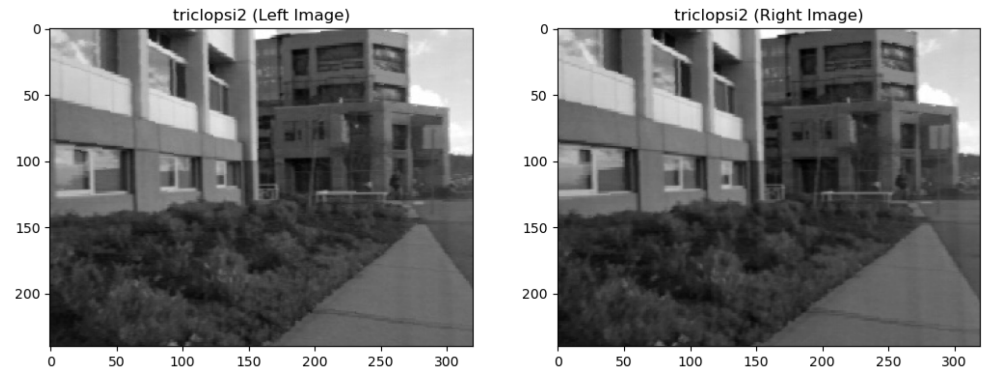
Analysis of *corridor* disparity maps: The regions on the image where the algorithm has difficulty estimating the disparities are those where pixel intensities are relatively uniform (such as the ceiling and the black walls facing the camera on the left and right foreground). Increasing the window size from 8×8 to 16×16 dramatically improves the results in these regions.

Analysis of *triclopsi2* disparity maps: For the walking path in the image, the algorithm is unable to estimate the disparities accurately because of its uniform pixel intensities. Likewise, the sky at the top right corner of the image also has poor results. However, excellent results (Figure 7(c)) are obtained for the patch of bushes to the left of the walking path, because of the uneven pixel intensities.

Effect of median filter: When the OpenCV median filter was applied to each disparity map, noise and small and isolated patches of wrongly estimated disparities are reduced.

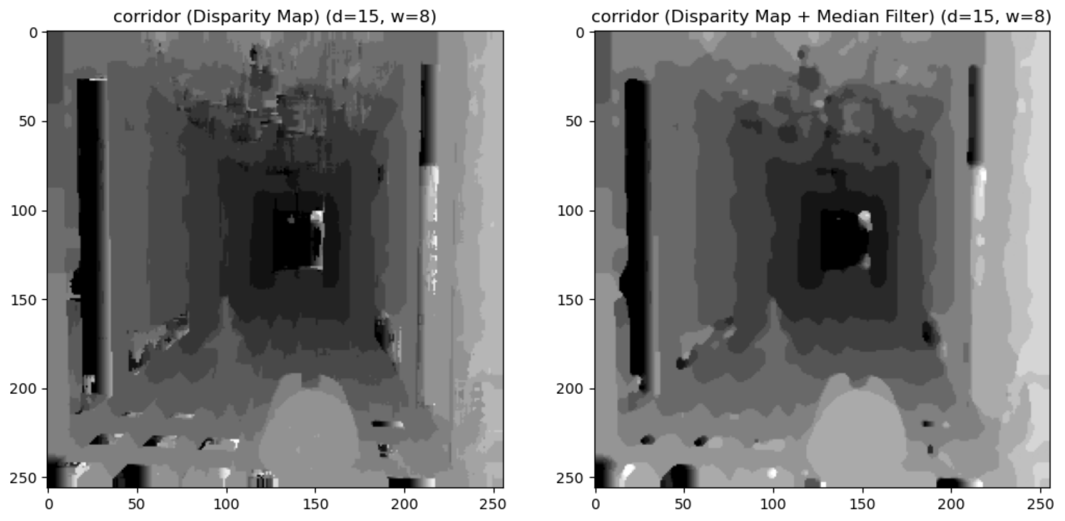


(a)

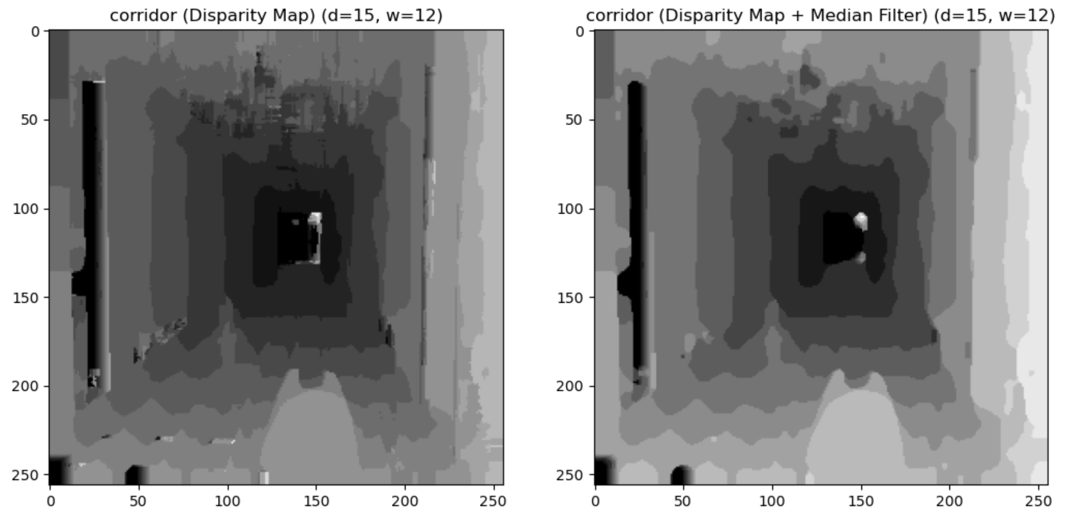


(b)

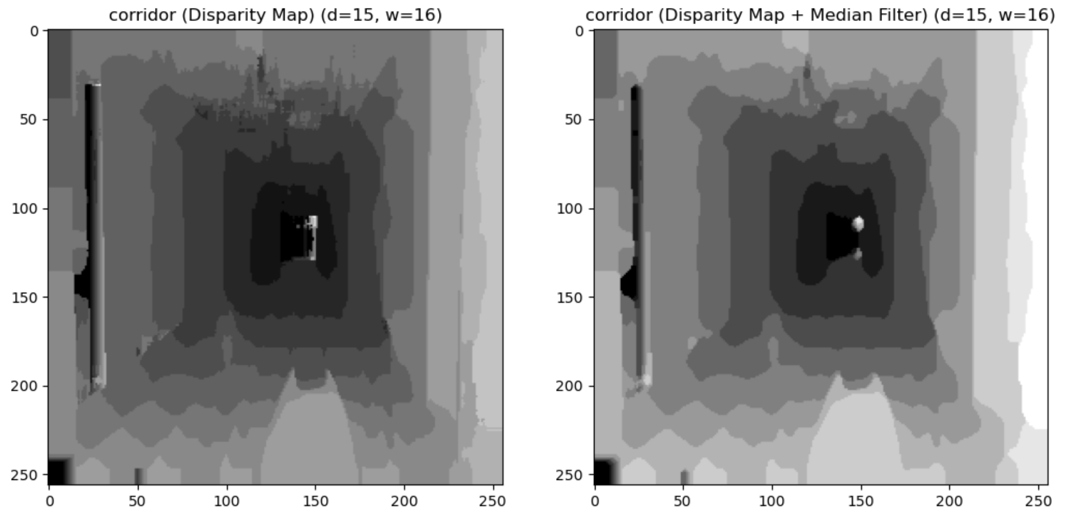
Figure 3: Rectified input images: (a) *corridor*, (b) *triclopsi2*



(a)

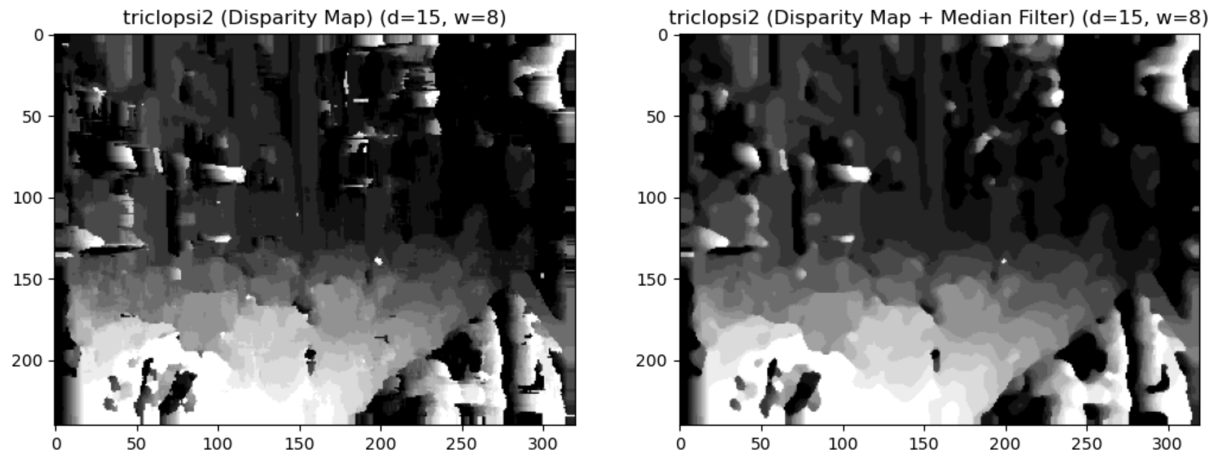


(b)

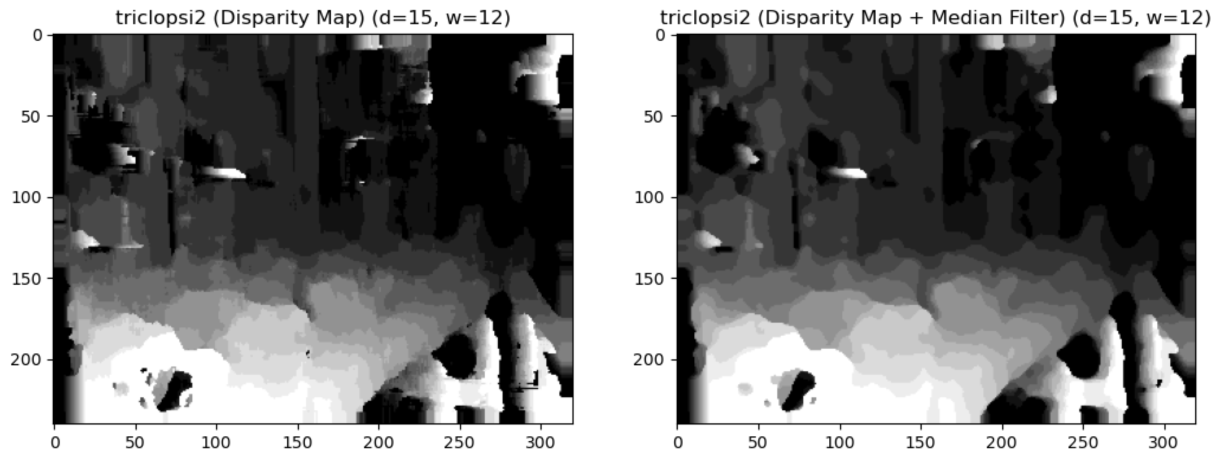


(c)

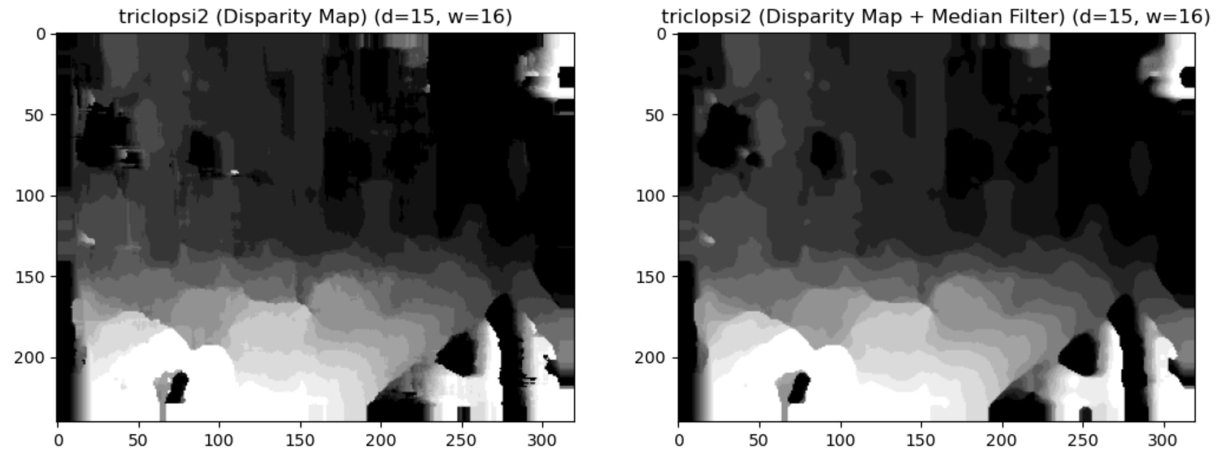
Figure 4: Disparity maps for *corridor* showing improved results for larger window sizes $w \times w$ for the same maximum disparity parameter $d = 15$: (a) 8×8, (b) 12×12, (c) 16×16 (Median filter applied to the disparity maps on the right.)



(a)

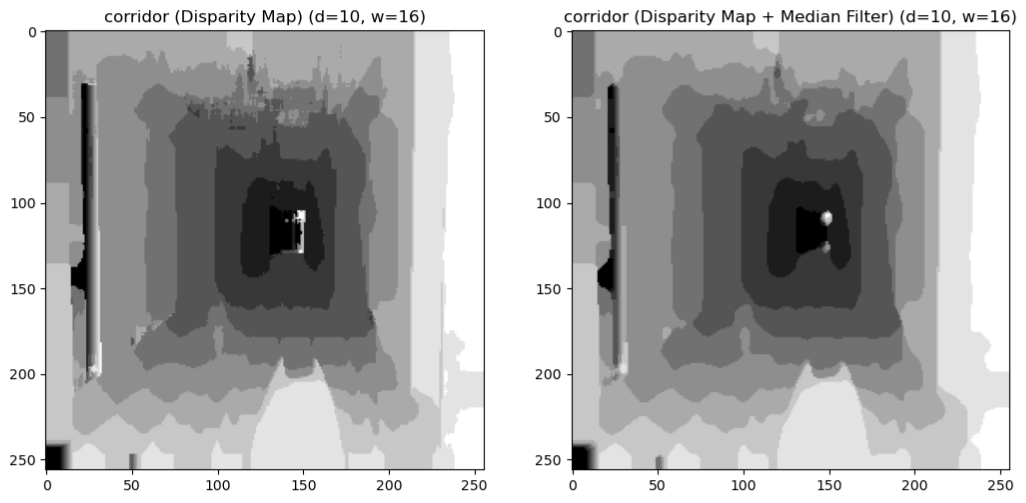


(b)

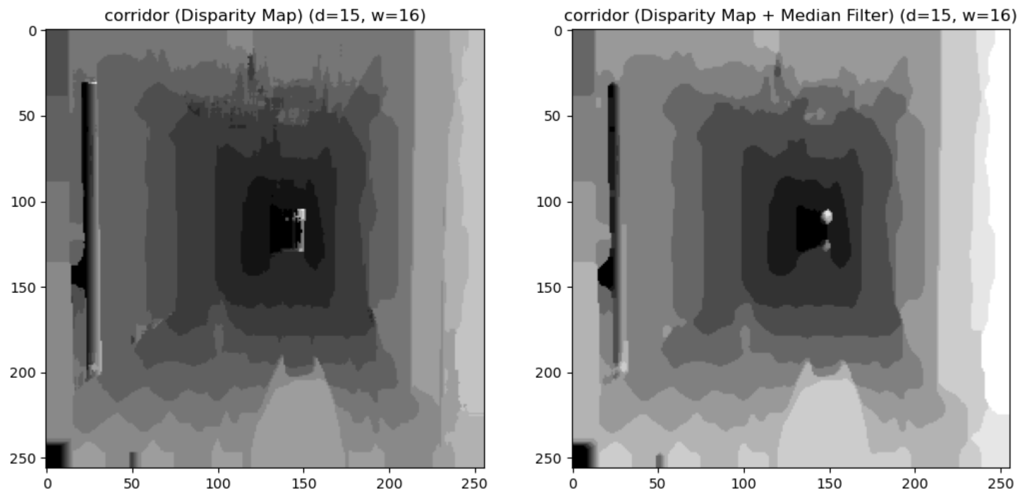


(c)

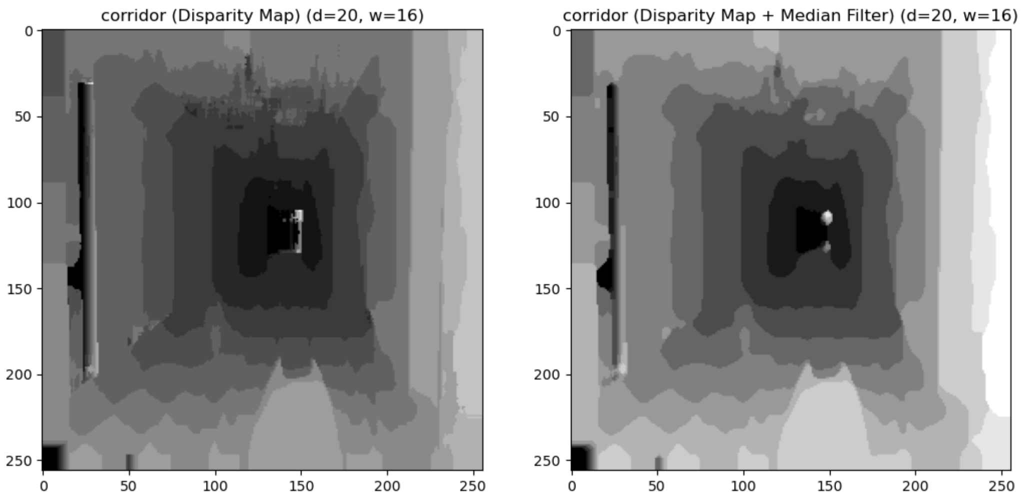
Figure 5: Disparity maps for *triclopsi2* showing improved results for larger window sizes $w \times w$ for the same maximum disparity parameter $d = 15$: (a) 8×8, (b) 12×12, (c) 16×16 (Median filter applied to the disparity maps on the right.)



(a)

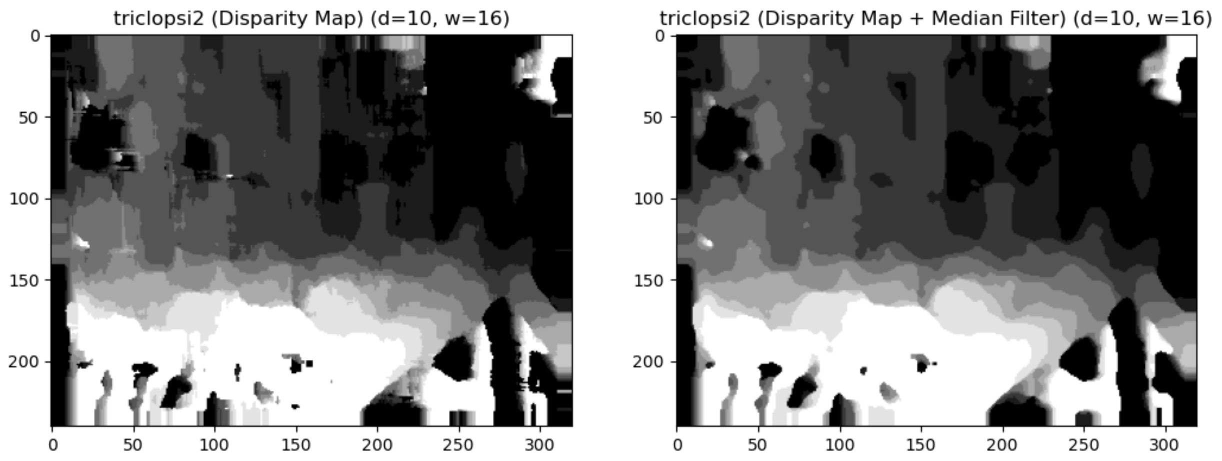


(b)

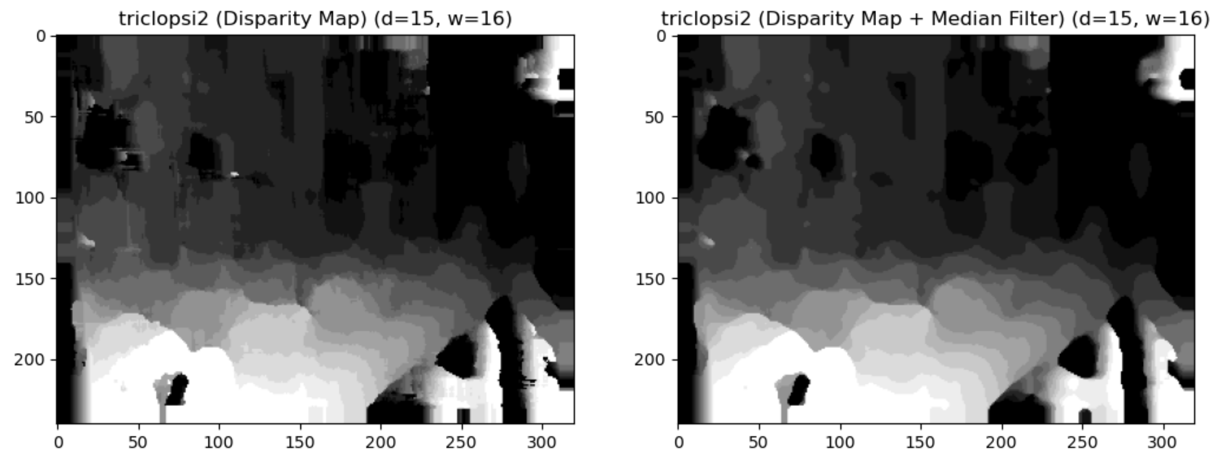


(c)

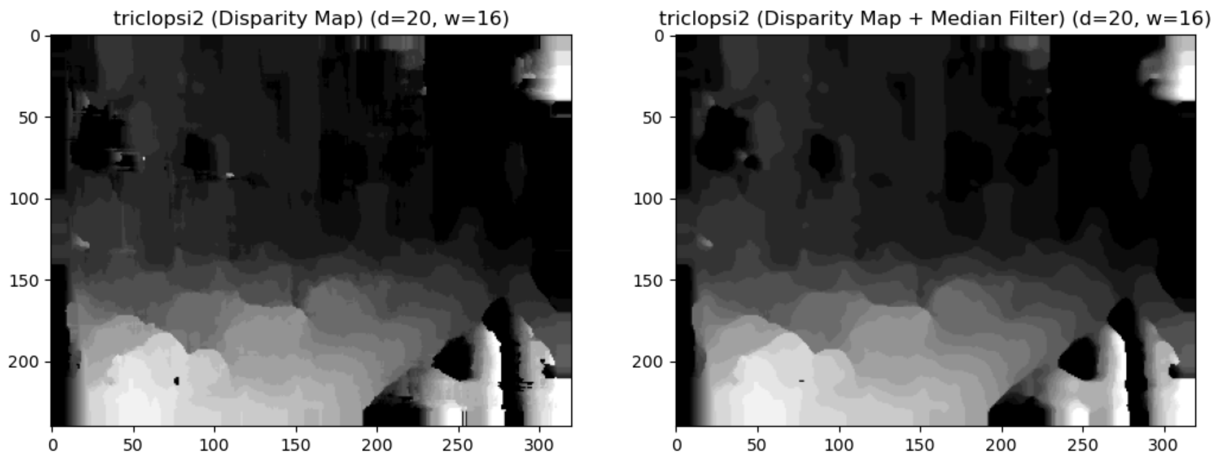
Figure 6: Disparity maps for *corridor* showing improved results for larger maximum disparity parameter $d = 15$ for the same window size 16×16. Identical results are obtained for $d = 15$ and $d = 20$, indicating that the greatest disparity between the images is no greater than 15 pixels: (a) $d = 10$, (b) $d = 15$, (c) $d = 20$ (Median filter applied to the disparity maps on the right.)



(a)



(b)



(c)

Figure 7: Disparity maps for *triclopsi2* showing improved results for larger maximum disparity parameter d for the same window size 16×16: (a) $d = 10$, (b) $d = 15$, (c) $d = 20$ (Median filter applied to the disparity maps on the right.)

4.3 Computation Time

Table 1 shows the computation time for different parameter values (maximum disparity d, window size $w \times w$).

	corridor	*triclopsi2*
$d = 15,\ w = 8$	125.6	150.1
$d = 15,\ w = 12$	247.9	304.6
$d = 15,\ w = 16$	405.1	489.0
$d = 10,\ w = 16$	294.6	400.0
$d = 20,\ w = 16$	551.2	676.5

Table 1: Computation time in seconds with median filter applied

5 Disparity Map Refinement

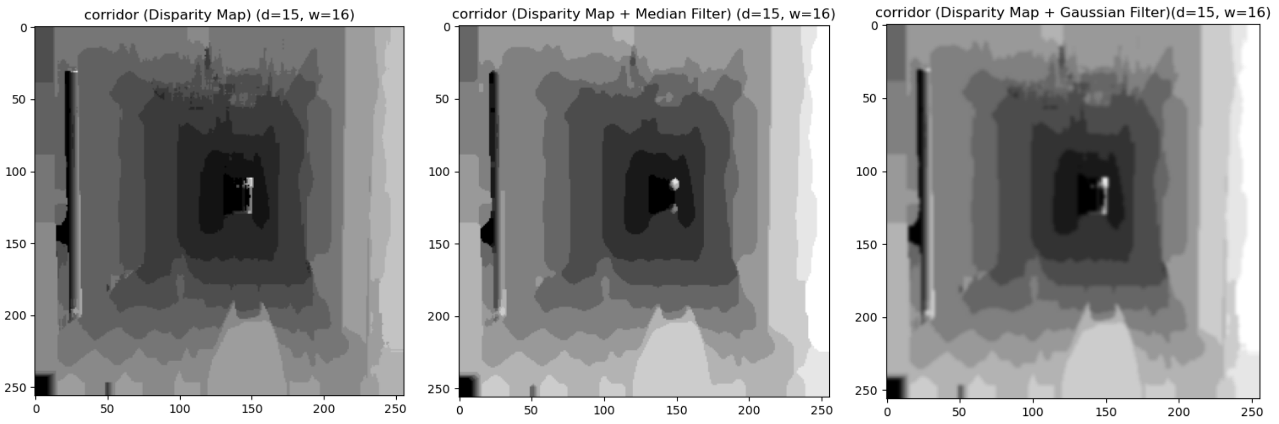
In this section, we discuss the factors that determine the accuracy of disparity computation and suggest possible improvements to our algorithm.

5.1 Factors Affecting Disparity Computation

Our results show that the choice of the parameter values of maximum disparity and window size is important. The ideal maximum disparity value can be estimated by inspecting the pair of images. Estimating the ideal window size is much harder. The window must be large enough to contain sufficient *texture* (i.e., variations in pixel intensities). But if the window is too large, it will straddle depth discontinuities and this may affect the accuracy of the disparity map. Solutions to overcome the problem of window size selection include using adaptive window sizes and pixel-weighted windows [3]. In regions on the image where texture is lacking, such as dark colored walls and clear skies, good estimation of disparities is difficult.

5.2 Refinement Methods Implemented

Stereo matching is a widely research problem. Many different methods for each step of the taxonomy have been proposed for window-based methods [1]. We limit our discussion to Step 4. In Step 4, the disparity map obtained can be improved by further processing. In Section 3, we describe two methods we use. First, we fill the borders of the disparity map by using the disparity values of the pixels adjacent to the borders. Second, we apply a median filter to the disparity map. The median filter helps to reduce noise or small, isolated patches of wrongly estimated disparities. Another popular filter used by researchers is the Gaussian filter. The Gaussian filter estimates disparities by using disparity values of neighboring pixels and weights defined by a Gaussian distribution. Figure 8 shows the original disparity map and the disparity maps obtained after applying OpenCV's median filter and Gaussian filter. The median filter produces better results.



(a) **(b)** **(c)**

Figure 8: Disparity maps for *corridor* showing the results of applying a filter: (a) original disparity map (no filters), (b) with OpenCV's median filter, (c) with OpenCV's Gaussian filter

6 Conclusion

We now conclude our report with a summary of the work presented and the results obtained.

We explained the method of using support windows to compute disparities between corresponding points in rectified left and right images. We implemented an algorithm to compute the disparity of two given pairs of rectified images and plot their disparity maps. We experimented with different values of two parameters, maximum disparity and window size, and compared their results.

We presented the disparity maps with and without applying a median filter. For the given pairs of images, the results improved as the maximum disparity parameter value is increased, up to a certain limit. Increasing the window size also helped in improving the accuracy of disparity maps. From our results, we noted that regions on the image with uniform pixel intensities yield poor results, while regions showing clear variations in pixel intensities produce excellent results.

Lastly, we discussed the factors affecting the accuracy of disparity estimation and some possible improvements to our algorithm.

References

[1] Rostam Affendi Hamzah and Haidi Ibrahim. Literature survey on stereo vision disparity map algorithms. *Journal of Sensors*, 2016, 2016. 14

[2] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International journal of computer vision*, 47(1):7–42, 2002. 3

[3] Richard Szeliski. *Computer Vision: Algorithms and Applications.* Springer Science & Business Media, 2010. 1, 14