Report

AI6121 Assignment 02 Disparity and Depth

Group Members:

Wang Yuhui (Matriculation Number: G2202262J)

Kishore Rajasekar (Matriculation Number: G2101949G)

Sean Goh Ann Ray (Matriculation Number: G2202190G)

Team Contribution:

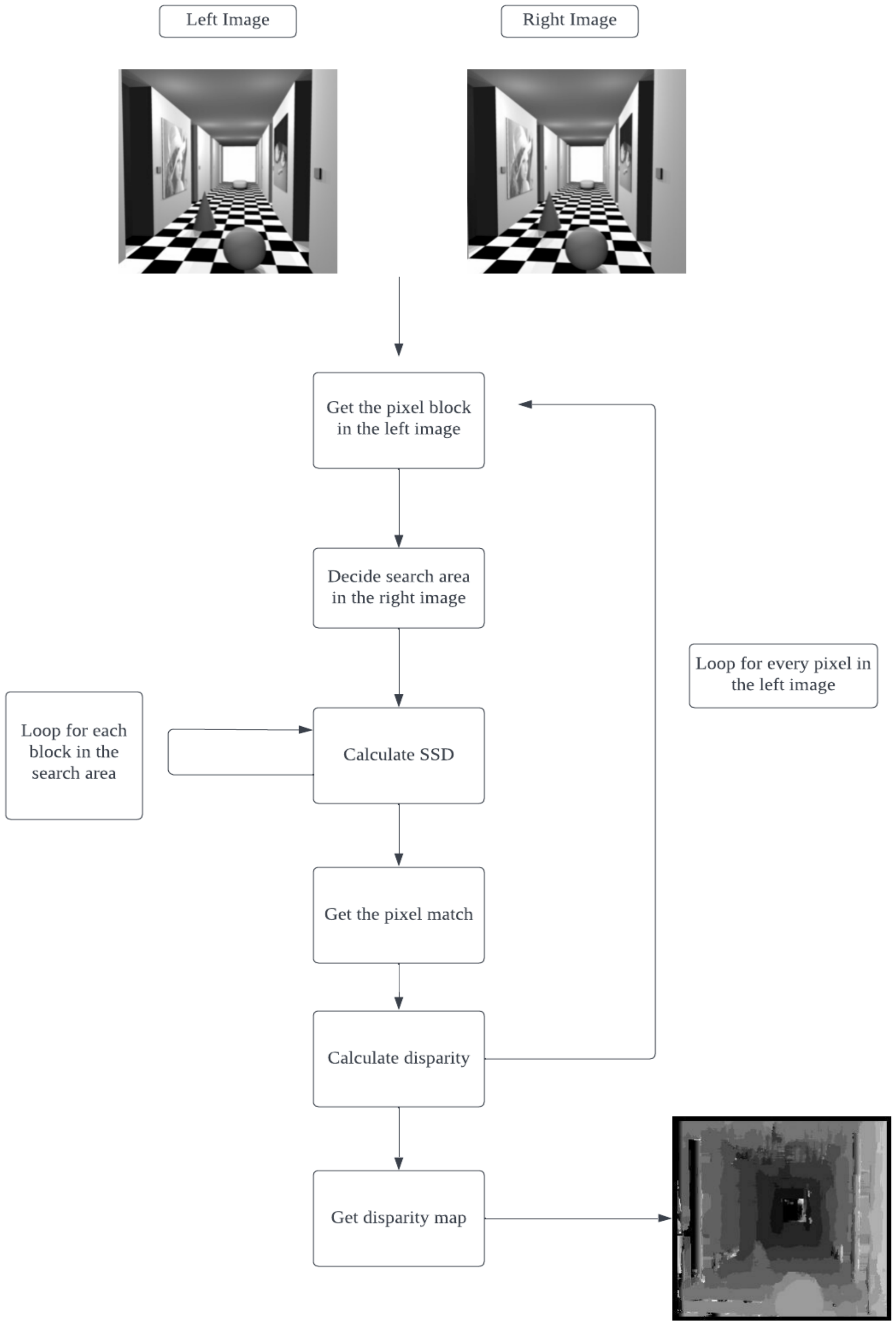
1. Wang Yuhui
   - Report structure, flowchart and procedure description
2. Kishore Rajasekar
   - Improved disparity algorithm and result discussion
3. Sean Goh Ann Ray
   - Baseline disparity algorithm and result discussion

1. Introduction

In this report, we will first describe the procedure for computing the disparity map. Then we provide the detailed code of our algorithm, show and discuss the result obtained. Finally, the factors which affect the computation and the corresponding improvements will be discussed.

2. Disparity Computing

2.1 Flow Chart


Left Image
Right Image
Get the pixel block in the left image
Decide search area in the right image
Loop for each block in the search area
Calculate SSD
Loop for every pixel in the left image
Get the pixel match
Calculate disparity
Get disparity map

2.2 Procedure Description

To compute the disparity map for a pair of rectified images which have the same scene from different viewpoints, the most important step is to find every pixel match in the two images. This means that for every pixel in the left image, a corresponding pixel in the right image needs to be found.

Taking one pixel in the left image as an example (x_l,y_l), we first look at its neighbouring pixels as a square block. Then we need to search for an area in the right image which contains blocks of the same size as the block selected before. Since the images are already rectified, the corresponding points will be on the same horizontal line ($y_r = y_l$). So, the search area will only be in the same horizontal line of the selected block.

After deciding the search area and getting every block within the search area, the next step is to compare these blocks with the selected block in the left image to find the best matching block. For each pair of blocks, we calculate the sum of squared difference (SSD) of the pixels in the block. The SSD formula is as follows:

$$SSD = \sum_{u=-N}^{N} \sum_{v=-N}^{N} [I(x + u, y + v) - g(u,v)]^2$$

where I(x,y) is the center pixel of the block of interest in the right image, and g(u,v) is the corresponding pixel of the block of interest in the left image. The SSD of each pair of blocks is then compared, where we select the pair with the minimum SSD. The pixel in the center of the right block (x_r,y_r) is the match for the pixel in the left image that we choose at the beginning. The disparity value is then calculated as the absolute difference of the match. For example, if the center pixel of the block in the left image was (x_l,y_l) and the center pixel of the best matched block found in the right image was (x_r,y_r), the disparity value would be $|x_r$-$x_l|$. Since disparity is inversely proportional to the depth or the distance between the camera and the object, as given by the formula:

$$d = \frac{fT_x}{Z}$$

where d is the disparity value, Z is the depth, f is the focal length of the camera, and T_x is the distance moved by the camera in the horizontal axis. Since f and T_x are constants, we can see that the greater the disparity, the smaller the value of Z, meaning that the point of interest is closer to the camera.

Since we can get the disparity of one pixel, the process is repeated for every possible pixel in the left image to obtain the disparity map. Pixels at and near the border do not have the full block of pixels, hence they are not computed, thus producing the black border in the disparity map. The disparity map is then scaled from 0 to 255 for easier visualization.

Disparity maps can be used to identify depths from a pair of stereo images, assuming that focal length of the camera and T_x is known. Therefore, with stereo cameras in video recording modes, it is possible to know the depth of any point of interest, and hence the depth of the environment. Applications include robots with stereo vision for navigating around its environment.

2.3 Code

```
## import libraries

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image


## define functions

# get the index of the minimum Sum of Squared Difference in the row
def row_wise_comparison(row_idx, col_idx, patch_size, img_width, search_width):
  """
  row_idx, col_idx: index of the pixel of interest of the left image in the center of patch
  patch_size: size of patch, must be N pixels above, below, left, and right of the center pixel,
therefore must be 2N + 1 (odd number), equivalent of ‘block’ in the report
  img_width: horizontal size of image
  search_width: the horizontal search size (in pixels) for the right image to the left and right
  of the pixel of interest, equivalent of ‘search area’ in the report
  """
  N = int((patch_size - 1) / 2)

  # define row-wise search area on right image, constrained to border of thickness N
  col_min_idx = max(N, col_idx-search_width) # left limit of col_idx
  col_max_idx = min(img_width-N, col_idx+search_width) # right limit of col_idx

  # initialize dist from col_idx
  old_dist = max(col_idx - col_min_idx, col_max_idx - col_idx)

  # initialize min_SSD as maximum difference possible
  min_SSD = (255**2) * (patch_size**2)

  # initialize min_idx as first search index
  min_idx = col_min_idx

  # define patch of left image for comparison
  left_patch = left_img[(row_idx-N):(row_idx+N+1), (col_idx-N):(col_idx+N+1)]
#     print(f'left patch array at {row_idx, col_idx}: \n {left_patch}')

  # repeat for search_width
  for i in range(col_min_idx, col_max_idx):

     # define right patch area
```

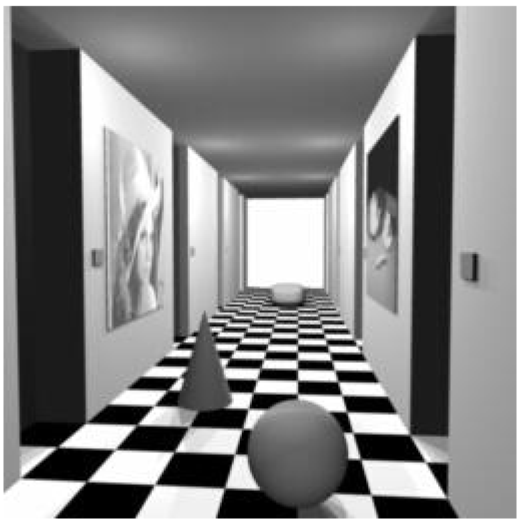
```python
        right_patch = right_img[(row_idx-N):(row_idx+N+1), (i-N):(i+N+1)]
#           print(f'right patch array at {row_idx, i}: \n {right_patch}')

        # get Sum of Squared Difference between left and right patches
        SSD_patch = ((right_patch - left_patch)**2).sum()

        # get current absolute dist from col_idx
        new_dist = abs(i-col_idx)

        # if the SSD of current patch is smaller than previous smallest SSD
        if SSD_patch < min_SSD:

            # save min_SSD and min_idx of the patch
            min_SSD = SSD_patch
            min_idx = i

        # if the SSD of current patch is equal to previous smallest SSD but nearer to col_idx
        elif SSD_patch == min_SSD and new_dist < old_dist:

            # save min_SSD and min_idx of the patch
            min_SSD = SSD_patch
            min_idx = i

        # update old_dist
        old_dist = new_dist

    return min_SSD, min_idx

# get the disparity map of the entire image
def get_disparity_map(left_img, right_img, patch_size, search_width):

    # get image height and width
    img_height = left_img.shape[0]
    img_width = left_img.shape[1]

    N = int((patch_size - 1) / 2)
#       print(f'N is {N}, img height is {img_height}, img width is {img_width}')

    # initialize disparity map array
    disparity_SSD_map = np.zeros((img_height, img_width)) # may not be required
    disparity_map = np.zeros((img_height, img_width))

    # get disparity map for every possible pixel in left_img, constrained to border of size N
    for i in range(N, img_height - N):
        for j in range(N, img_width - N):
            min_SSD, min_idx = row_wise_comparison(i, j, patch_size, img_width,
search_width)
            disparity_SSD_map[i,j] = min_SSD
            disparity_map[i,j] = j - min_idx

    disparity_SSD_map = np.floor(255 * (disparity_SSD_map /
np.amax(disparity_SSD_map)))
    return disparity_SSD_map.astype('uint8'), disparity_map


## Main algorithm

for i in range(2):
    # import left and right images as greyscale and convert to numpy array
    if (i == 0):
        # corridor image set
        left_img_file = "corridorl.jpg"
        right_img_file = "corridorr.jpg"
        save_filename = "corridor_disparity_map.jpg"
        print('Image set: Corridor')
    else:
        # triclopsi2 image set
        left_img_file = "triclopsi2l.jpg"
        right_img_file = "triclopsi2r.jpg"
        save_filename = "triclopsi2_disparity_map.jpg"
        print('Image set: triclopsi2')

    left_img = Image.open(left_img_file)
    left_img = left_img.convert(mode='L')
    display(left_img)
    left_img = np.asarray(left_img)

    right_img = Image.open(right_img_file)
    right_img = right_img.convert(mode='L')
    display(right_img)
    right_img = np.asarray(right_img)

    # set parameters
    patch_size = 13 # Heuristically chosen
    multiplier = 1 # Heuristically chosen
    search_width = patch_size * multiplier
    N = (patch_size-1)//2
    print(f'Patch size: {patch_size}, \t maximum search width: {2*N + search_width*2 + 1}')
```

```
  # get disparity map
  disparity_SSD_map, disparity_map = get_disparity_map(left_img, right_img, patch_size,
search_width)

#     print('SSD map:')
#     disparity_SSD_map_img = Image.fromarray(disparity_SSD_map, mode="L")
#     display(disparity_SSD_map_img)

  # disparity map to be upscaled for clearer depiction of disparity
  # brighter areas means larger disparity differences, meaning closer to camera
  print('Disparity map:')
  disparity_map = abs(disparity_map)
  disparity_map = disparity_map.astype('uint8')
  disparity_map = np.floor(255 * (disparity_map / np.amax(disparity_map)))
  disparity_map = disparity_map.astype('uint8')
  disparity_map_img = Image.fromarray(disparity_map, mode="L")
  display(disparity_map_img)

  # save image
  disparity_map_img.save(save_filename)
```

3. Result Discussion

The code developed in section 2.3 produces a disparity map such that the brighter the pixel, the larger the disparity, which suggests that it is closer to the camera in the real world, as discussed in the earlier section.

3.1 Corridor Image Set



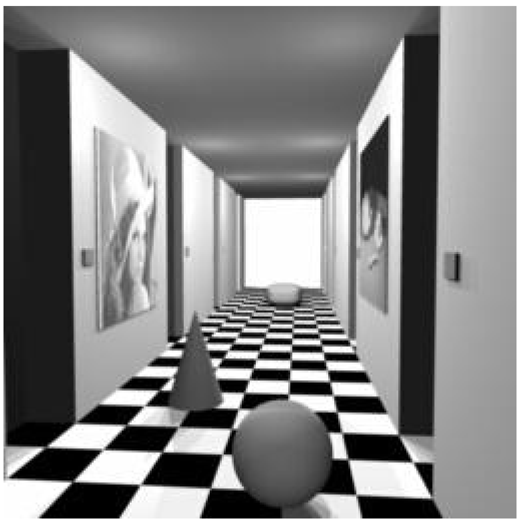


Fig. 1. Corridor_left

Fig. 2. Corridor_right

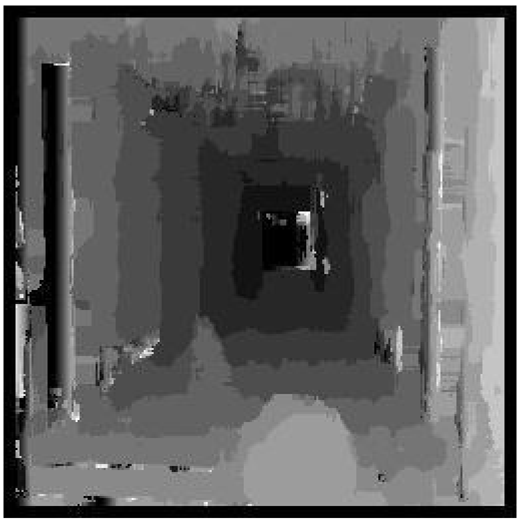
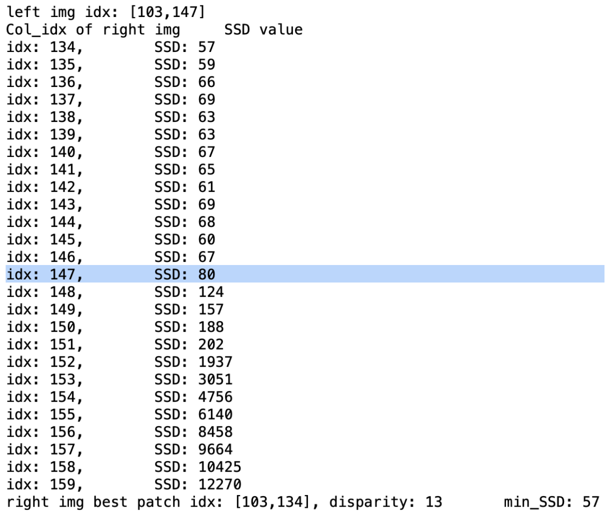


Fig. 3. Corridor_disparity_map


```
left img idx: [103,147]
Col_idx of right img     SSD value
idx: 134,        SSD: 57
idx: 135,        SSD: 59
idx: 136,        SSD: 66
idx: 137,        SSD: 69
idx: 138,        SSD: 63
idx: 139,        SSD: 63
idx: 140,        SSD: 67
idx: 141,        SSD: 65
idx: 142,        SSD: 61
idx: 143,        SSD: 69
idx: 144,        SSD: 68
idx: 145,        SSD: 60
idx: 146,        SSD: 67
idx: 147,        SSD: 80
idx: 148,        SSD: 124
idx: 149,        SSD: 157
idx: 150,        SSD: 188
idx: 151,        SSD: 202
idx: 152,        SSD: 1937
idx: 153,        SSD: 3051
idx: 154,        SSD: 4756
idx: 155,        SSD: 6140
idx: 156,        SSD: 8458
idx: 157,        SSD: 9664
idx: 158,        SSD: 10425
idx: 159,        SSD: 12270
right img best patch idx: [103,134], disparity: 13       min_SSD: 57
```
Fig. 4. SSD Values of Pixel [103,147]

On the corridor image set, the disparity map produced (Fig. 3) by the algorithm proves to be effective in most areas of the image. Along the floor and corridor walls of the images, the disparity map shows a gradual increase in the brightness of the pixels, suggesting that the areas near the border of the image are nearer to the camera, which is true from a visual perspective. Objects such as the triangular cone and sphere in the corridor can also be identified in the disparity map, where the sphere is brighter than the floor it is on, which is also true from the camera's perspective. The disparity map shows that the end of the corridor is mostly dark, meaning that the disparity between the left and right images is very small or almost zero, and this suggests that the end of the corridor is very far away.

However, there is a small area at the end of the corridor which shows bright pixels, meaning high disparity values. Upon closer inspection, the SSD difference of the compared patches is minimal. For example, as shown in Fig. 4, the pixel [103,147] is a pixel in the bright area mentioned. It shows a disparity of 13 pixels, which is also the search width size as defined in the algorithm. However, SSD values of some other blocks within the search area are also very small and comparable to that of the best block, largely due to the homogeneity of that area in the images. This shows that there are many similar-looking blocks within the search area, but the algorithm takes the index with the smallest SSD value to compute the disparity, without considering other factors such as the locations of the small difference in pixel values for each of these patches.

3.2 Triclopsi2 Image Set

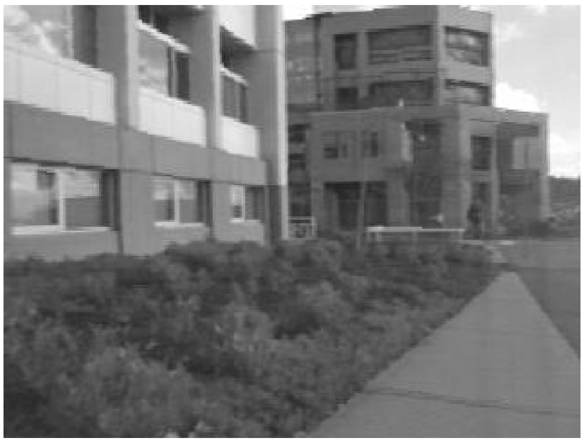


Fig. 5. Triclopsi2_left

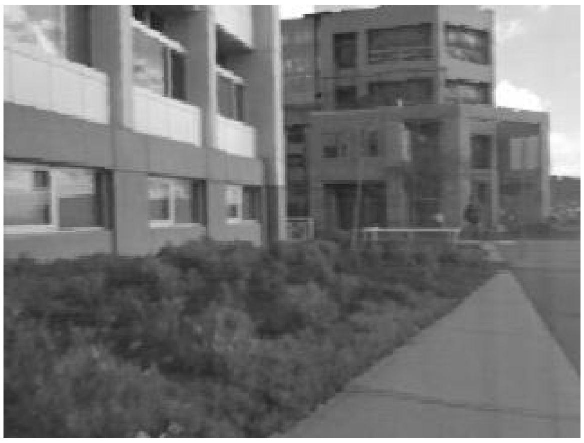


Fig. 6. Triclopsi2_right

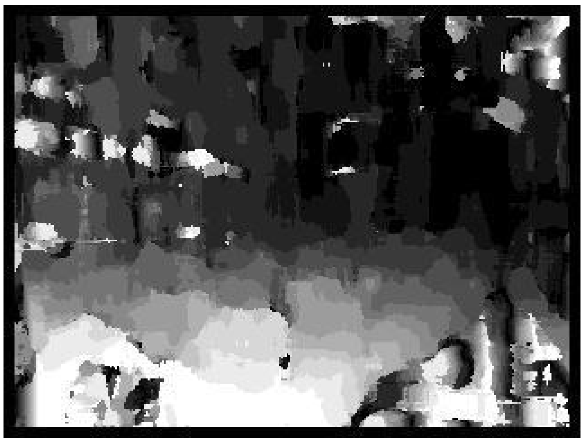


Fig. 7. Triclopsi2_disparity_map

On the triclopsi2 image set, the disparity map produced (Fig. 7) by the algorithm proves to be somewhat effective in the bushes, but not so much for the pavement and buildings. Nearer bushes do appear brighter with a gradient towards darker intensities as it goes further, with a small exception at the front, likely due to the 'noise' from the bush leaves in the original images.

However, the pavement shows a mix of bright and dark areas in a pattern that is inconsistent to common sense. Upon closer inspection of the original images, there are several observed vertical lines of discoloration along the entire height of the images on the pavement side of the images, and can be more obviously seen when alternating quickly from one photo to the other in a photo preview application window.

Additionally, some parts of the building walls are shown to have a large disparity, which does not align with common sense as the bushes in the foreground are much nearer. This is due to the same reason as the problem of the small area in the end of the corridor image set, where there are several blocks with similar SSD values but the algorithm simply takes in the one with the lowest SSD difference.

4. Potential Problems and Improvements

As observed in section 3 above, the quality of the stereo images would affect the quality of the disparity map. While this may be fixed using additional preprocessing algorithms, it is not part of the disparity computation and thus excluded.

The algorithm provided in section 2.3 to produce the results in section 3 uses a heuristically chosen value of the block size and search width, and may not be the best parameter values in both of the image sets as different stereo images would require different parameter settings.

Additionally, the algorithm utilizes SSD to compare the similarity of blocks of pixels. One other way to compare the blocks' similarity is to use the sum of absolute difference (SAD) [1]. We also introduced another dimension in our disparity map in order to capture the depth. This can be observed in the code where the computed ssd or sad matrix is a three dimensional array. This change in the algorithm helps to capture the depth information in terms of how close or farther away from the camera a given pixel might be. We then conduct experiments with varying parameters of depth and block size using the SAD algorithm.

4.1 Code

```
import numpy as np

import matplotlib.pyplot as plt
```

ssd function

```
def compute_depth(l_img, r_img, depth, w_size):
 """
 l_img: left image
 r_img: right image
 depth: depth of the disparity map, equivalent of ‘search area’ in the report
 w_size: size of the window, equivalent of ‘block’ in the report
 """
 Y, X = l_img.shape
 sim = np.zeros((Y, X, depth))
 half_w = w_size//2

 # loop through the image
 for x in range(half_w, Y - half_w):
   for y in range(half_w, X - half_w):
     # loop through the window:
     for z in range(-half_w, half_w + 1):
       for u in range(-half_w, half_w + 1):
        # loop through all depths
        for d in range(depth):
          sim[x,y,d] += (int(l_img[x+z, y+u]) - int(r_img[x+z, y+u-d])) ** 2  # used for ssd
    #sim[x,y,d] += np.abs(int(l_img[x+z, y+u]) - int(r_img[x+z, y+u-d]))#used for sad

 return sim
```

sad function

```
def compute_depth_sad(l_img, r_img, depth, w_size):
 """
 l_img: left image
 r_img: right image
 depth: depth of the disparity map
 w_size: size of the window
 """
 Y, X = l_img.shape
 sim = np.zeros((Y, X, depth))
 half_w = w_size//2

 # loop through the image
 for x in range(half_w, Y - half_w):
   for y in range(half_w, X - half_w):
     # loop through the window:
     for z in range(-half_w, half_w + 1):
       for u in range(-half_w, half_w + 1):
         # loop through all depths
         for d in range(depth):
           sim[x,y,d] += np.abs(int(l_img[x+z, y+u]) - int(r_img[x+z, y+u-d]))
 return sim

def compute_disparity(sim):
 return np.argmin(sim, axis=2)

corridor = [cv.imread('/content/gdrive/MyDrive/AI6121/assignment 2/corridorl.jpg',
cv.IMREAD_GRAYSCALE), cv.imread('/content/gdrive/MyDrive/AI6121/assignment
2/corridorr.jpg', cv.IMREAD_GRAYSCALE)]
```

```
triclopsi2 = [cv.imread('/content/gdrive/MyDrive/AI6121/assignment 2/triclopsi2l.jpg',
cv.IMREAD_GRAYSCALE), cv.imread('/content/gdrive/MyDrive/AI6121/assignment
2/triclopsi2r.jpg', cv.IMREAD_GRAYSCALE)]


sim = compute_depth(corridor[0], corridor[1], 16, 7)
d_map = compute_disparity(sim)

fig, (ax1) = plt.subplots(1,1)
ax1.imshow(d_map, cmap='gray')
plt.show()
```

4.2 Conclusion

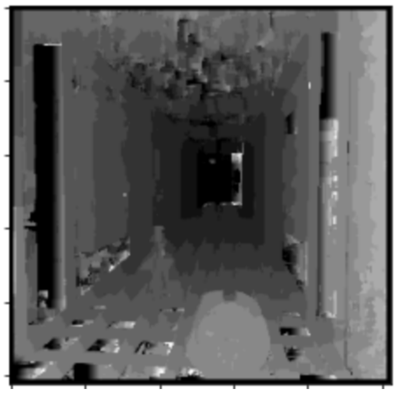


Fig. 8. disparity using ssd

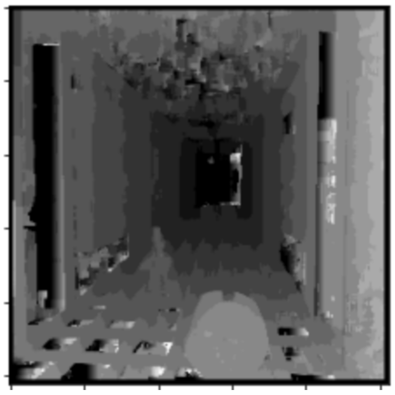


Fig. 9. disparity using sad

Above shown is the comparison of the disparity obtained from using the SSD algorithm and SAD algorithm and it can be said that there is very little noticeable difference when we use the same parameters such as maximum depth and block size. This could be due to argmin at the end of each operation even though SSD maximizes its value by squaring for large values and minimizes values below 1. However, in terms of the disparity image, there is more detail revealed in the one obtained using SAD when examined closely.

Hence, we experimented by tuning the parameters to gauge the performance of the new algorithm.

We have depth information capturing how close or farther away from the camera a given pixel might be. Therefore a lower value set for maximum depth would mean that the disparity image contains less information in terms of its depth. Below are the disparity images shown for different values of depth. When the depth value is increased, it can be observed that the disparity of the image contains more layers of depth and hence captures more information. The computation power also increases as we increase the value of the max depth.

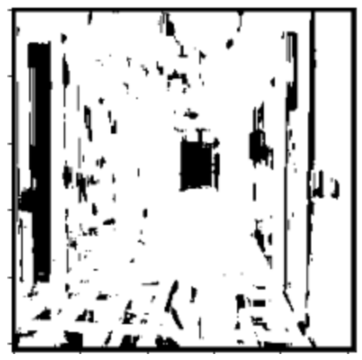


Fig. 10. disparity using depth=2

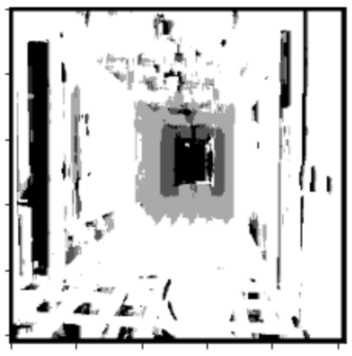


Fig. 11. disparity using depth=4

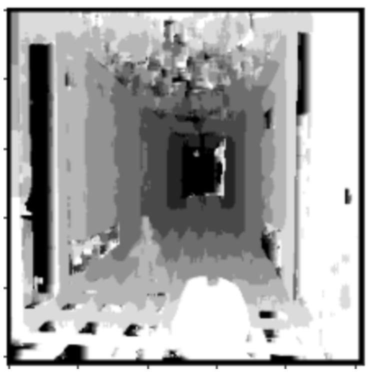


Fig. 12. disparity using depth=8

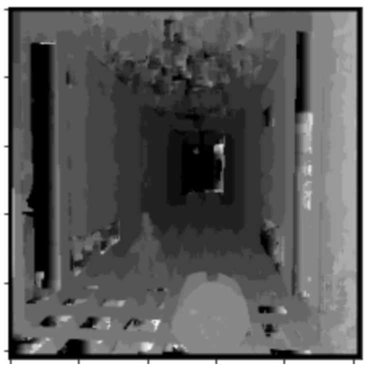
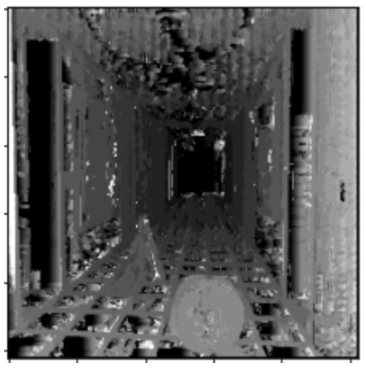


Fig. 13. disparity using depth=16

Similarly, below are the disparity images shown for varying values of block sizes.

Every pixel in the left image is used as a reference to compare and match with the pixel and its nearby pixel within the window range. It is observed that as we increase the block size, the depth information is captured better bearing the excess computational power.



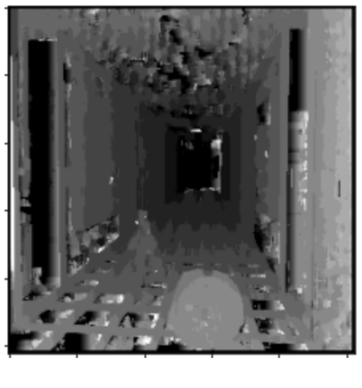
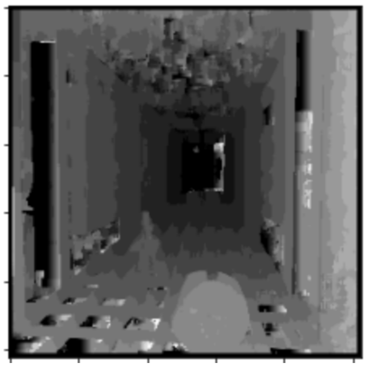


Fig. 14. disparity w_size=3

Fig. 15. disparity w_size=5



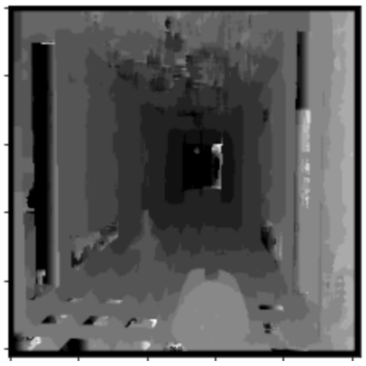


Fig. 16. disparity w_size=7

Fig. 17. disparity w_size=9

References

[1] GitHub - Txuso/SAD-Disparity-Map-Algorithm-: SAD in Python, C++ and C++ with parallelization added, from https://github.com/Txuso/SAD-Disparity-Map-Algorithm-