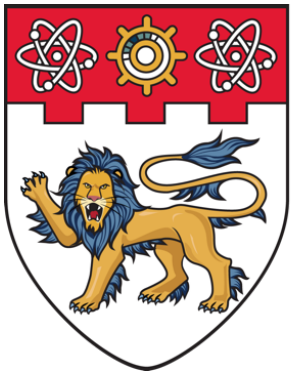


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

AI6121 Computer Vision

Assignment 2 Report

authored by

Ong Jia Hui
JONG119@e.ntu.edu.sg
G1903467L

Abstract

Stereo matching, which is to estimate depth or disparity map from two rectified images from left and right cameras, has always been a challenging problem in computer vision. There are two main types of stereo matching, namely appearance-based and feature-based. In terms of appearance-based matching, there are two main approaches, namely local and global approaches [1]. The most generic method is a local window-based approach that matches regions in window across a scanline between the left and right images. The most prominent type of feature-based approach uses scale-invariant feature transform (SIFT) [3] used for robust feature point detection and description.

In this assignment, the general procedure of disparity map computation will first be discussed in Chapter 2. It will present the four main stages namely, cost computation, cost aggregation, disparity selection and disparity refinement. The following chapter will elaborate on the Python implementation of the basic sliding window approach. As such approach is highly sensitive to the parameters used during the computation such as the window size, the parameter tuning process will be documented in Chapter 4. Observations of the final disparity maps produced by the window-based approach will be also be discussed and analyzed in the same chapter. Motivated by the flaws of the window-based approach, the last chapter seeks for better techniques to improve the overall quality of the disparity maps. The results of the improvised approach by using guided filters will be shown and compared against the window-based approach.

Contents

1. Introduction

1.1 Background

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.

Stereo matching, which is to estimate depth or disparity map from two rectified images from left and right cameras, has always been a challenging problem in computer vision. There are two main types of stereo matching, namely appearance-based and feature-based. In terms of appearance-based matching, there are two main approaches, namely local and global approaches [1]. The most generic method is a local window-based approach that matches regions in window across a scanline between the left and right images. The most prominent type of feature-based approach uses scale-invariant feature transform (SIFT) [3] used for robust feature point detection and description.

1.2 Tasks Description

(a) Corridor

(b) triclopsi2

Figure 1.1: Sample stereo images

Given the two pairs of stereo images shown in Figure 1.1(a) and 1.1(b) are rectified images, stereo matching becomes the disparity estimation problem. This aim of solving this correspondence problem is to match points with similar appearances in the left and right images to reconstruct the depth perception. Visually, the two given images revealed that they are captured by stereo cameras with small baseline distances. Furthermore, there are little to none occlusions in the given pairs of images. From these observations, correspondence by appearance matching can be carried out.

The first chapter will elaborate on the four main stages of stereo mapping. The second chapter will follow up with the basic methodologies discussed and enumerate the step-by-step process of performing a sliding window-based approach. The third chapter focuses on the discussion on the results obtained from the implemented approach. Improvement ideas derived from the observations made in the previous chapter will be implemented in chapter four. Finally, the results were analyzed, and their evaluated effectiveness were documented.

2. Task 1: General Procedure

2.1 Window-based Disparity Map Computation

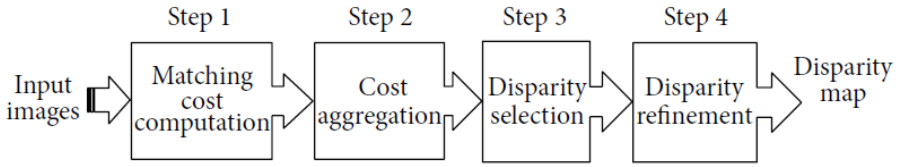


Figure 2.1: General pipeline for stereo mapping [1]

Disparity map algorithms have often performed using multi-stage techniques. Scharstein and Szeliski summarized these techniques into four main steps [4] as shown in Figure 2.1. In order to compute the disparities, at least two images captured by stereo cameras placed horizontally apart are required. Image rectification transforms these captured images by aligning their epipolar lines before they are used as inputs. In this assignment, this step will not be necessary as the sample images given are pairs of rectified images of the same scene captured from two different viewpoints.

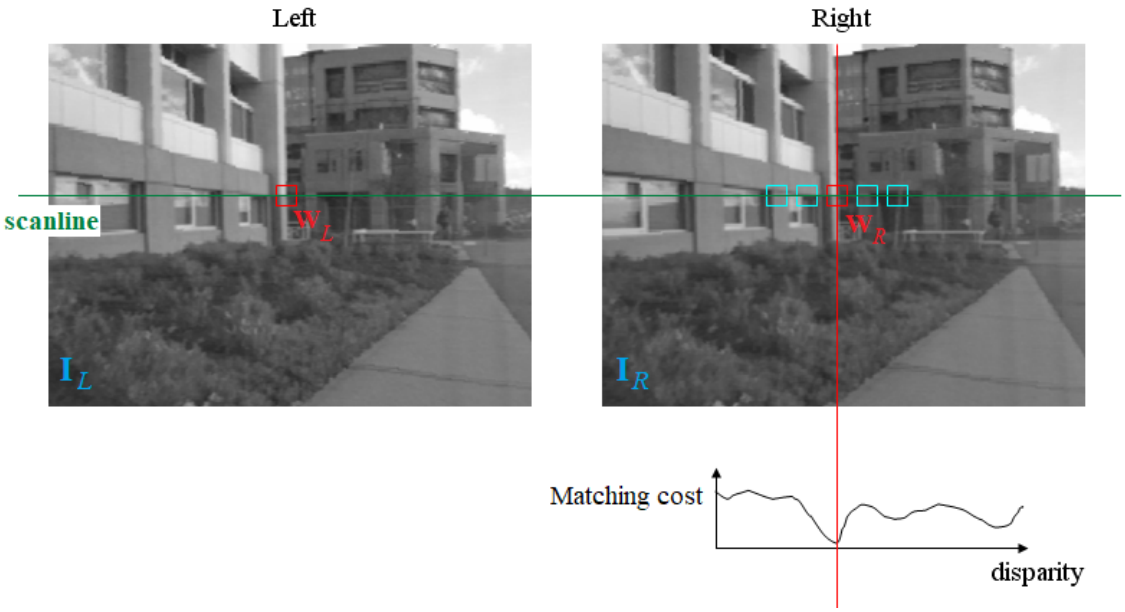


Figure 2.2: Correspondence search using window-based approach

The process of disparity computing starts with the matching cost computation. This first step determines whether the values of two pixels correspond to the same scene by computing the disparity between them. The disparity is the difference between the x coordinates of two corresponding points in the left and right image lying on the same

scanline. In window-based approaches, matching cost computation are often combined with the cost aggregation step by summing the matching cost over square windows with the same disparity. The common matching cost algorithms used include sum of absolute differences (SAD) and sum of squared differences (SSD).

Given w_L and w_R are corresponding windows of $m \times m$ pixels, the window function can be defined as:

$$W_m(x,y) = \{u,v|x - \frac{m}{2} \leq u \leq x + \frac{m}{2}, y - \frac{m}{2} \leq u \leq y + \frac{m}{2}\} \tag{2.1}$$

With I_L representing the left stereo image and I_R representing the right stereo image, the SAD cost function can be defined as:

$$SAD(x,y,d) = \sum_{(u,v)\in W_m(x,y)} |I_L(u,v) - I_R(u-d,v)| \tag{2.2}$$

Likewise, the SSD function can be computed using:

$$SSD(x,y,d) = \sum_{(u,v)\in W_m(x,y)} [I_L(u,v) - I_R(u-d,v)]^2 \tag{2.3}$$

Equation 2.2 depicts the sum of absolute differences algorithm, which takes the absolute value of the difference between the pixel intensity in window of left image and corresponding pixel intensity in window of right image. The alternative SSD algorithm shown in Equation 2.3 aggregates over the squared differences of the two corresponding pixels in the windows of the left and right images.

$$d_{(x,y)} = \arg\min_{d\in D} C(x,y,d) \tag{2.4}$$

In the third stage of disparity selection, the most common method is to used the winner-takes-all (WTA) strategy formulated in Equation 2.4. This strategy selects the minimal aggregated values of each pixel as final disparity values. The set of these final values can be used to plot the disparity map.

As the raw disparity map generated often contains large amount of noises, the disparity map refinement stage helps to improve the quality of these maps. A common technique used is to apply a median filter, which selects the median value within the window of pixels to replace the central pixel.

3. Task 2: Implementation

3.1 Pre-requisites/ Setup

The source code of this assignment are implemented in Python with the use of Numpy library as well as Matplotlib for graphical representations. The OpenCV2 library will also be used to load and save images as well as to perform image filtering.

3.1.1 Loading the dataset

There are two pairs of rectified images provided for this assignment, namely "corridor" and "triclopsi2". A dataclass named "StereoImage" was created to store the left and right images of the same scene. The implementation codes can be found in Listing 6.1 in Appendix A. All sample stereo images are first loaded to a list prior to passing into the disparity mapping pipeline.

Listing 3.1: StereoImage Dataclass Object

```
# Define constants
DATA_DIR = "../data"
LEFT_EXT = "l.jpg"
RIGHT_EXT = "r.jpg"
SAMPLE_NAMES = ["corridor", "triclopsi2"]
OUTPUT_DIR = "../out"

def load_stereo_images():
    global DATA_DIR
    global LEFT_EXT, RIGHT_EXT
    global SAMPLE_NAMES
    stereo_images = []
    for img_name in SAMPLE_NAMES:
        img_l_path = os.path.join(DATA_DIR, img_name + LEFT_EXT)
        try:
            img_l = cv2.imread(img_l_path, cv2.IMREAD_GRAYSCALE)
        except:
            print(f"fail to find left image at: {img_l_path}")
            continue

        img_r_path = os.path.join(DATA_DIR, img_name + RIGHT_EXT)
        try:
            img_r = cv2.imread(img_r_path, cv2.IMREAD_GRAYSCALE)
        except:
            print(f"fail to find right image at: {img_r_path}")
            continue
        assert img_l.shape == img_r.shape # check both images have same H, W
        stereo_images.append(StereoImage(img_name, img_l, img_r))
    return stereo_images

stereo_images = load_stereo_images()
print(f"Loaded {len(stereo_images)} images.")
```

3.2 Sliding Window Approach

This section breaks down the implementation steps for a generic sliding window approach to generate the disparity map of the stereo images. Table 3.1 shows the methods used for the four stages of pipeline, previously described in Section 2.1.

Step	Stage	Method
1	Match Cost Computation	SD
2	Cost Aggregation	SSD + Median Filter
3	Disparity Selection	WTA
4	Disparity Refinement	Median Filter

Table 3.1: Sliding Window Disparity Computing Pipeline

3.2.1 Stage 1 - Match Cost Computation

In traditional window-based approach, two windows of size $m \times m$ are traversed across the scanline along the same y axis. D is the preset maximum disparities to check against the window on the right image such that $d \in D$. As shown in Equation 2.1, the x-coordinates window bound would be $\pm\frac{m}{2}$ for the left image, and $(\pm\frac{m}{2} - d)$ for the right image.

Listing 3.2: compute_matching_cost_window function

```
def compute_matching_cost_window(S: StereoImage, D: int, m: int, algo='ssd'):
    """
    This function computes the matching cost between the stereo images.

    Args:
        S (StereoImage): data class containing left & right image arrays
        D (int): max number of disparities
        m (int): window size
        algo (str): choose either 'ssd' or 'sad'

    Returns:
        cost (np.ndarray): cost volume of size (H,W,D)
    """
    assert algo == 'ssd' or algo == 'sad'
    cost = np.zeros((S.H, S.W, D))
    m_2 = m//2

    # move to pixel location
    for y in range(m_2, S.H - m_2):
        for x in range(m_2, S.W - m_2):
            # window (m x m)
            for v in range(-m_2, m_2 + 1):
                for u in range(-m_2, m_2 + 1):
                    w_l = S.I_l[y+v,x+u]
                    # disparities range
                    for d in range(D):
                        w_r = S.I_r[y+v,x+u-d]
                        if algo == 'ssd':
                            cost[y,x,d] += (w_l - w_r)**2
                        elif algo == 'sad':
                            cost[y,x,d] += np.abs(w_l - w_r)
    return cost
```

Based on the selected algorithm, this function defined in Listing 3.2 either performs SSD or SAD. The implementation codes closely follow the formulas defined in Equations 2.3 and 2.2 for window-based sum of squared errors and sum of absolute errors respectively. The output of the function returns a three-dimensional array of {y, x, d} matching cost values. For each x pixel in the left window (w_l), the corresponding the right image window (w_r) will be shifted to the left side by 0 to d pixels before their differences are computed.

3.2.2 Stage 2 - Cost Aggregation

In this stage, the cost volume will be aggregated by convolving the disparity dimension with a normalized low-pass filter. The central element will be replaced with the average value of the kernel area. This aggregation helps to remove the noise between the windows. The function created can be found in Listing 3.3.

Listing 3.3: cost_aggregation_avg function

```
def cost_aggregation_avg(cost:np.ndarray, f:int=3):
    """
    This function takes the average of all the disparities
    under kernel area and replaces the central element with the average

    Args:
        cost: matching cost volume
        f (int): kernel size for averaging

    Returns:
        agg_cost (np.ndarray): agg cost volume of size (H,W,D)
    """
    agg_cost = np.zeros(cost.shape)
    D = cost.shape[2]
    for d in range(D):
        agg_cost[:,:,d] = cv2.blur(cost[:,:,d],(f,f))

    return agg_cost
```

3.2.3 Stage 3 - Disparity Selection

In this stage, the winner-takes-all (WTA) strategy is applied on the disparities dimension to select the minimum value of each (y,x) coordinate in Listing 3.3.

Listing 3.4: disparity_selection_wta function

```
def disparity_selection_wta(cost: np.ndarray):
    """
    Implement WTA strategy:
    Select the minimum disparity value for each pixel

    Args:
        cost (np.ndarray): 3D array (h, w, d)

    Returns:
        dmap (np.ndarray): 2D array (h, w)
    """
    return np.argmin(cost, axis=2)
```

3.2.4 Stage 4 - Disparity Refinement

In this stage, the disparity map smoothing will be carried out. The disparity map is smoothed by convolving using an average filter as shown in Listing 3.5.

Listing 3.5: disparity_refinement_smooth function

```
def disparity_refinement_smooth(dmap: np.ndarray, f=3):
    """
    Performs disparity map smoothing:
    Args:
        dmap (np.ndarray): disparity map
        f (int): kernel size for averaging
    Returns:
        dmap (np.ndarray): smoothed disparity map
    """
    return cv2.blur(dmap,(f,f))
```

3.2.5 Overall function

By calling the window_based_disparity function, the stereo images will pass through the four stages in the pipeline. Stage two and four can be turned off individually to see the effects of the raw disparity maps by setting the arguments "agg_filter" and "ref_filter" to zero respectively. The source codes for 'S.show()" function can be found in Listing 6.1 of Appendix A. The output is a StereoImage data class object containing the left, right and disparity maps.

Listing 3.6: window_based_disparity function

```
def window_based_disparity(S: StereoImage, D: int, m: int, algo='ssd',
                   agg_filter=5, ref_filter=3, plot=True, save_dir=None):
    """
    Implements the window-based disparity computing of two stereo images.
    Args:
        S (StereoImage): data class containing left and right image arrays
        D (int): max number of disparities
        m (int): window size
        algo (str): choose either 'ssd' or 'sad'
        agg_filter (int): aggregation filter size
        ref_filter (int): refinement filter size
        plot (bool): show disparity map
        save_path (str): save directory, set None to not save
    Returns:
        S (StereoImage): object with updated disparity map
    """
    cost = compute_matching_cost_window(S, D=D, m=m, algo=algo)
    if agg_filter > 0:  cost = cost_aggregation_avg(cost, f=agg_filter)
    d_map = disparity_selection_wta(cost)
    if ref_filter > 0:  d_map = disparity_refinement_smooth(d_map, f=ref_filter)
    S.set_disparity_map(d_map)
    if plot:
        save_path = save_dir
        if save_dir is not None:
            save_path = os.path.join(save_dir,
             ↪  f"DMAP_{S.name}_D={D}_m={m}_{algo}_af={agg_filter}_rf={ref_filter}.jpg")
            print(save_path)
        S.show(save_path=save_path)
    return S
```

4. Task 3: Results Discussion

In this chapter, the subsequent sections will discuss how the changes in the parameters will affect the generated disparity maps and the final disparity output of each stereo image pairs after tuning will be analyzed.

4.1 Parameter Tuning

This section explores the effects of the changes in the function parameters, specifically the D, m, af and rf values, which represent max disparities, window size, aggregation filter size, refinement filter size.

Stage	Parameters	V1	V2	V3	General Observations
1	Max Disparities (D)	D=8	D=16	D=20	As D increases, the search space increases, but it greatly increases the time needed to compute match costs.
1	Window Size (m)	m=5	m=7	m=9	As m increases, the disparity map becomes sharper, but details will be lost at larger values.
2	Aggregation Filter Size (af)	af=0	af=1	af=3	As af increases, noise reduces and filled in, high values even out edges.
4	Refinement Filter Size (rf)	rf=0	rf=1	rf=3	As rf increases, the edges are smoother and the image smoothing effect is much greater than aggregation filter.

Table 4.1: General observations for some of the parameters trialed

4.1.1 Max Disparities (D) Tuning

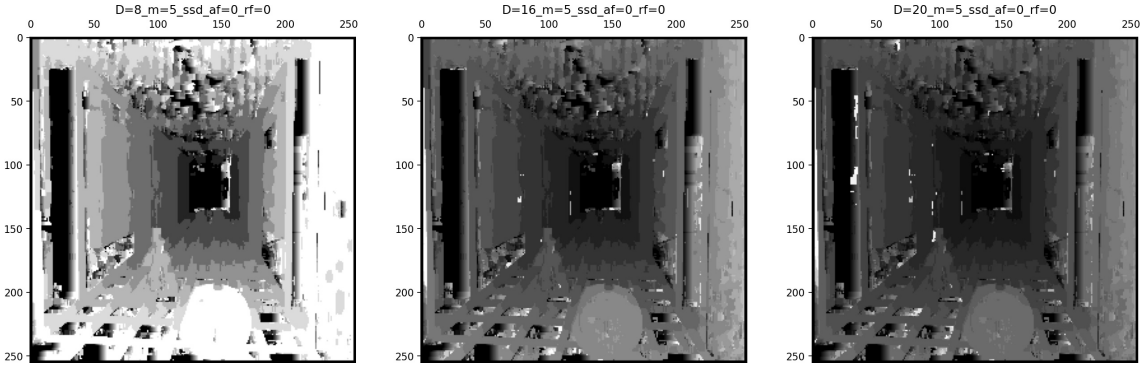


Figure 4.1: Corridor - Effects of increasing max number of disparities from 8, 16 to 20

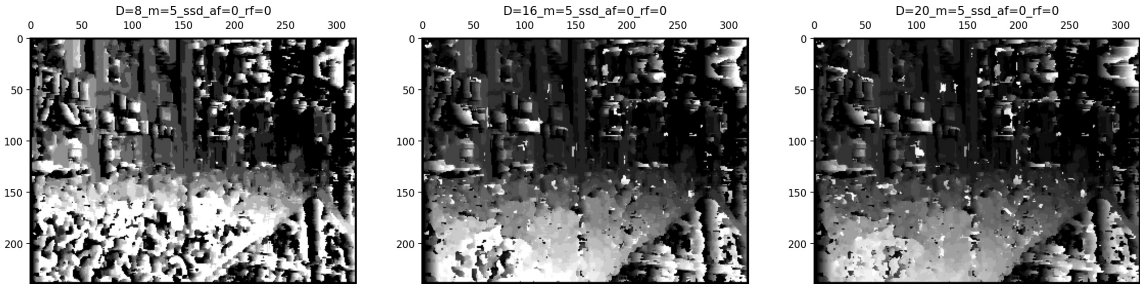


Figure 4.2: triclopsi2 - Effects of increasing max number of disparities from 8, 16 to 20

For the generation of the raw disparity maps in Figure 4.1 and 4.2, the rest of the parameters were set to constant values of m=5, af=0 and rf=0. This would mean that only stage 1 and 3 (no cost aggregation and disparity refinement) were performed on this set of images to prevent any smoothing effects that would affect the comparison of D values. When the parameter D increases, the matching cost across a larger search spaces are computed, which helped to improve the overall quality of the disparity maps. However, the increased search space also means more calculation needs to be done and greatly increases the computation time.

4.1.2 Window size (m) Tuning

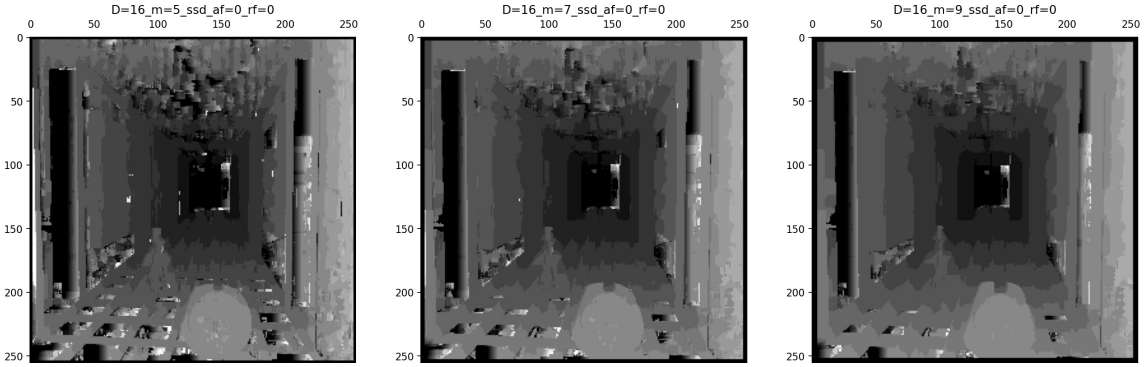


Figure 4.3: Corridor - Effects of increasing window size from 5, 7 to 9

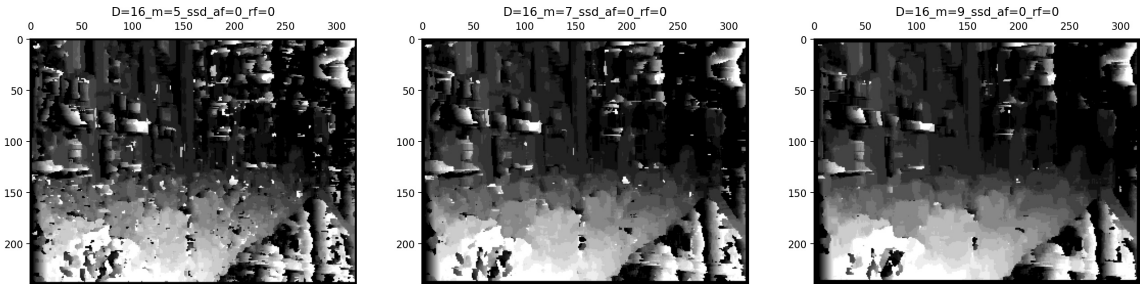


Figure 4.4: triclopsi2 - Effects of increasing window size from 5, 7 to 9

With constant d set to 16, af=0 and rf=0, Figure 4.3 and 4.4 showed that when the window size (m) increases from 5 to 7, the noticeable salt and pepper noises are removed. However, there is a need for balance to find a window size large enough to have sufficient intensity variation, yet small enough to contain only pixels with about the same disparity.

4.1.3 Aggregation Filter Size (af) Tuning

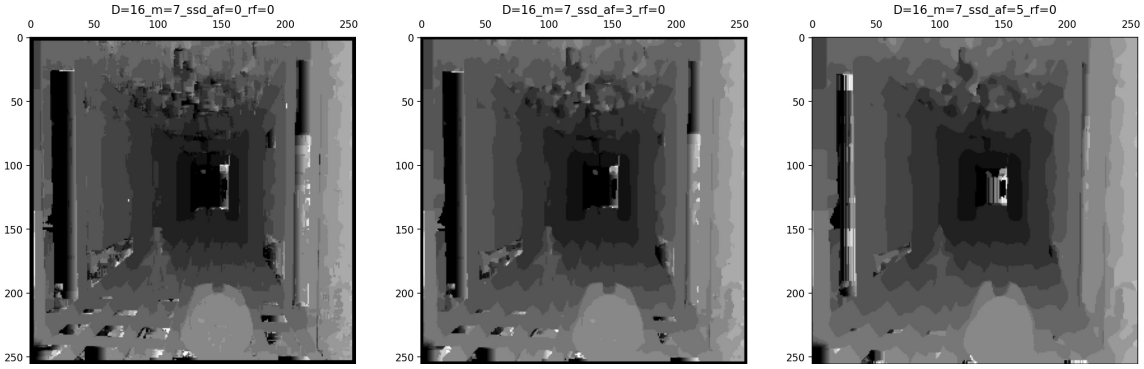


Figure 4.5: Corridor - Effects of increasing aggregation filter size from 0, 3 to 5

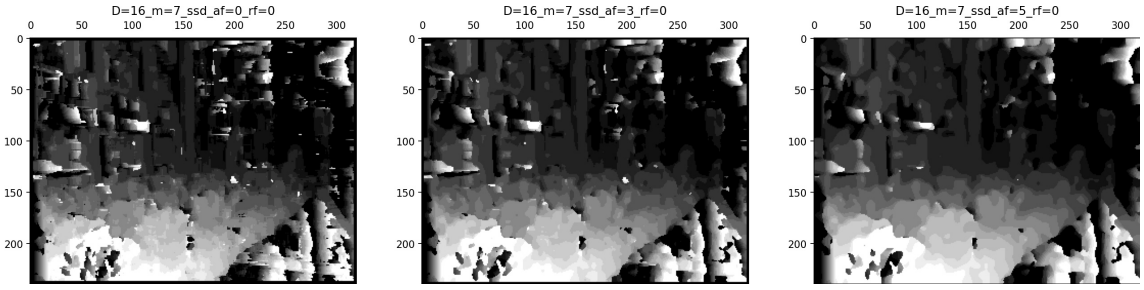


Figure 4.6: triclopsi2 - Effects of increasing aggregation filter size from 0, 3 to 5

Figure 4.5 and 4.6 are generated by setting d=16, m=7 and rf=0. When the parameter af increases, the overall noise (white spots) of the images are reduced as the central disparity values are replaced by the averaging kernel. However, it is observed that high values of af will have unwanted effects of evening out the edges.

4.1.4 Refinement Filter Size (rf) Tuning

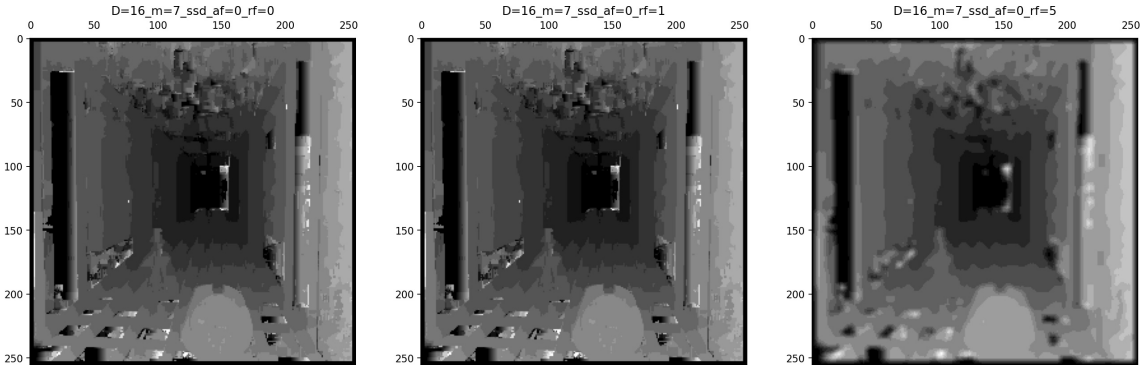


Figure 4.7: Corridor - Effects of increasing refinement filter size from 0, 2 to 5

As shown in Figure 4.7 and 4.8, when the parameter rf increases, the edges of the disparity map are smoother. However, as the convolution of the median filter is applied on the entire disparity map, the smoothing effect will be much stronger, hence setting high

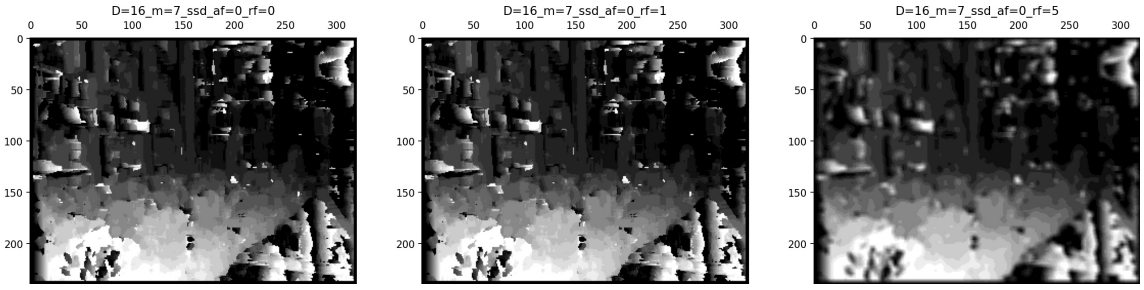


Figure 4.8: triclopsi2 - Effects of increasing refinement filter size from 0, 2 to 5

values of rf is not desirable. These disparity maps are produced with other parameters set to constant values of d=16, m=7 and af=0.

4.2 Tuned Parameter Outputs

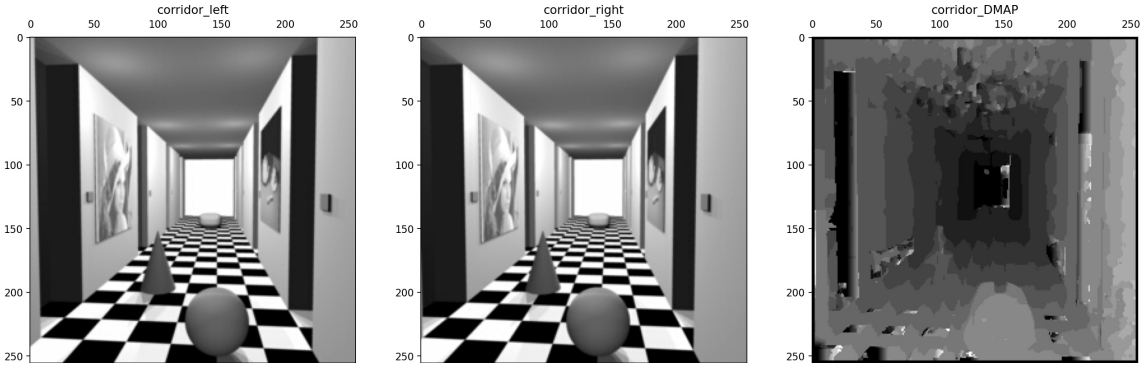


Figure 4.9: Corridor - Left, Right and Disparity map of window-based approach

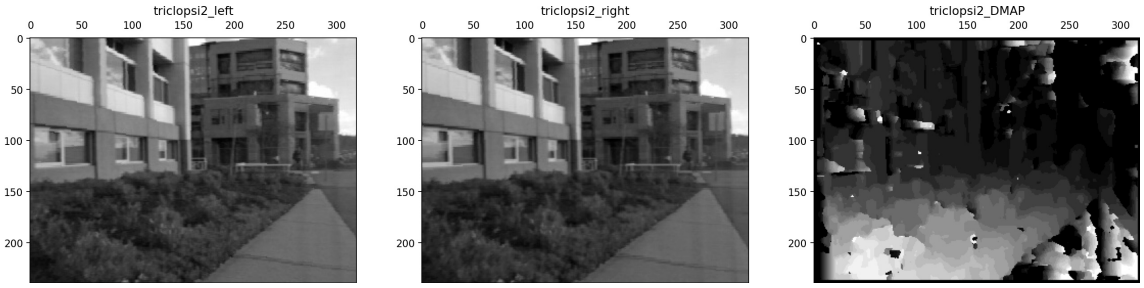


Figure 4.10: triclopsi2 - Left, Right and Disparity map of window-based approach

The final disparity maps after parameter tuning are shown in Figure 4.9 and 4.10. The best parameters found for corridor scene are D=16, m=7, af=3 and rf=1, while for triclopsi2 scene are D=20, m=9, af=3 and rf=1. Note that the parameters do not represent the most optimal settings as they are purely combination of non-exhaustive search. From the outputs, the flaws of sliding window-based approach are apparent. Due to its block shape search, it is unable to capture the cone and ball shapes perfectly in the corridor scene in Figure 4.9. Disparity refinement stage smoothed the gradient of the different disparity levels and reduces noise, but its main caveat is that the simple median filter also smooths out edges, leading to sharp lines being lost.

5. Task 4: Improvement Ideas

In this chapter, an alternative pipeline for disparity computing is derived by using better stage four refinement techniques and the outputs will be evaluated against the sliding window approach.

5.1 Guided Filter Approach

Due to the observation of lost edges in the output disparity maps, the idea is to use more sophisticated refinement methods to better improve the quality of the output maps. One of approach created by Hosini et al. in their 2012 paper [2] was to align the maps' transitions with the edges of the original image. It is performed efficiently using an edge preserving filter. This implementation also takes reference of the article by Tan et al. [5] and a Python adaption [6] of it.

Step	Stage	Method
1	Match Cost Computation	SD
2	Cost Aggregation	Median Filter
3	Disparity Selection	WTA
4	Disparity Refinement	L-R Consistency Check, Densification, Guided Weighted Filter

Table 5.1: Stereo Disparity through Cost Aggregation with Guided Filter Pipeline [5]

The match cost computation stage is modified to perform pixel-by-pixel squared differences instead of window-based. As shown in Table 5.1, the cost aggregation stage reuses the same median filter function shown in Listing 3.3. Similarly, the disparity selection stage also executed WTA strategy, hence the same function developed in Listing 3.4 can be reused. The core changes can be found in the disparity refinement process, where the simple median filter (Table 3.1) was replaced with a left-right consistency check and densification process as well as a guided weighted filter. These source code additions will be elaborated further in following sections.

5.1.1 Implementation

5.1.1.1 Stage 1 - Pixel-wise Match Computation

Listing 5.1: compute_matching_cost_pixel function

```
def compute_matching_cost_pixel(S: StereoImage,
                  D: int, l2r:bool, algo="sd"):
    """
    This function computes the pixel-wise matching cost between the LR images.

    Args:
        S (StereoImage): data class containing left and right image arrays
        D (int): max number of disparities
        l2r (bool): calculate disparities from left to right or vice versa
        algo (str): choose either 'sd' or 'ad'

    Returns:
        cost (np.ndarray): cost volume of size (H,W,D)
    """
    assert algo == 'sd' or algo == 'ad'
    cost = np.zeros((S.H, S.W, D))

    # disparities range
    for d in range(D):
        # for each scanline
        for y in range(S.H):
            for x in range(S.W):  #each x pixel
                if l2r:
                    if x-d >= 0:
                        w_l = S.I_l[y,x]
                        w_r = S.I_r[y,x-d]
                        if algo == 'sd':
                            cost[y,x,d] = (w_l - w_r)**2
                        elif algo == 'ad':
                            cost[y,x,d] = np.abs(w_l - w_r)
                else:
                    if x+d < S.W:
                        w_l = S.I_r[y,x]
                        w_r = S.I_l[y,x+d]
                        if algo == 'sd':
                            cost[y,x,d] = (w_l - w_r)**2
                        elif algo == 'ad':
                            cost[y,x,d] = np.abs(w_l - w_r)
        if d > 0: # adds padding at each scanline
            if l2r :
                for i in range(d):
                    cost[:,x,d] = cost[:,d,d]
            else:
                for i in range(d):
                    cost[:,(S.W-1)-x,d] = cost[:,(S.W-1)-d,d]
    return cost
```

The match cost computation was modified to perform pixel-wise match cost computation with reference to the design of [5]. Furthermore, other than the default left image to right image comparison, a "l2r" option was added to allow right image to left image difference calculation when set to False. Both left-to-right and right-to-left disparity maps will be used for left-right consistency checking in the latter stage. The implementation (squared differences (SD) sequence of I_L and I_R does not matter) follows the SD formula are denoted in Equation 5.1.

$$SD(x,y,d) = \begin{cases} [I_L(x,y) - I_R(x-d,y)]^2 & \text{if } l2r = True \\ [I_R(x,y) - I_L(x+d,y)]^2 & \text{if } l2r \neq True \end{cases} \tag{5.1}$$

5.1.1.2 Stage 4.1 - Left-Right Consistency Check

Listing 5.2: disparity_refinement_con_check function

```
def disparity_refinement_con_check(S: StereoImage,
                           dmap_l: np.ndarray, dmap_r: np.ndarray):
    """
    Performs consistency check on the left and right disparity maps.

    Args:
        S (StereoImage): data class containing left and right image arrays
        dmap_l (np.ndarray): disparity map of l2r differences
        dmap_r (np.ndarray): disparity map of r2l differences

    Returns:
        labels (np.ndarray): dmap_l with inconsistencies removed
        holes (array): inconsistent x, y coordinates
    """
    holes = []
    dmap_l = np.float64(dmap_l)
    for y in range(S.H):
        for x in range(S.W):
            if x - int(dmap_l[y,x]) >= 0:
                if dmap_l[y,x] != dmap_r[y, x-int(dmap_l[y,x])]:
                    # inconsistency found
                    dmap_l[y,x] = 0
                    holes.append((y,x))
    return dmap_l, holes
```

The left and right consistency checking function identifies the pixels where value of the disparity map from the left image coincides with the disparity computed from the right image. When a pixel is rejected, it is removed from the left disparity map (by setting the value to 0) and its position will be recorded in a "holes" array that will be used by the following densification function.

5.1.1.3 Stage 4.2 - Densification Process

Listing 5.3: disparity_refinement_densification function

```
def disparity_refinement_densification(labels: np.ndarray, holes: list):
    """
    Performs hole filling process of replacing the zero values in the
    disparity maps (by consistency check) with a neighbourhood value.

    Args:
        labels (np.ndarray): disparity map with inconsistencies removed
        holes (array): inconsistent x, y coordinates

    Returns:
        labels (np.ndarray): densified disparity map
    """
    for y, x in holes:
```

```
            slice_l, slice_r = labels[y,0:x], labels[y,x+1:labels.shape[1]]
            can_l, can_r = -1, -1
            i, j = len(slice_l)-1, 0
            while (can_l <= 0 and i >= 0):
                can_l = slice_l[i]
                i -= 1
            while (can_r <= 0 and j < len(slice_r)):
                can_r = slice_r[j]
                j += 1
            if can_l <= 0:
                labels[y,x] = can_r
            elif can_r <= 0:
                labels[y,x] = can_l
            else:
                labels[y,x] = min(can_l, can_r)
        return labels
```

This function refills the disparity map after the left right inconsistency check. Based on the coordinates of the rejected disparities in the "holes" array, it replaces the missing values in disparity map with the lowest disparity value of the spatially closest un-rejected pixels.

5.1.1.4 Stage 4.3 - Guided Weighted Filters

Listing 5.4: disparity_refinement_guided_filter function

```
def disparity_refinement_guided_filter(image: np.ndarray, dmap: np.ndarray,
                                       w_i:int=15, w_d:int=5):
    """
    Performs guided weighted median filtering on original image
    and its disparity.

    Args:
        image (np.ndarray): Image array for reference
        dmap (np.ndarray): Disparity map
        w_i (int): Weights for image array pixels
        w_d (int): Weights for disparity map
    Returns:

        r_dmap (np.array): refined disparity map
    """
    c_image = np.array(image, dtype='uint8')
    dmap = dmap.astype('uint8')
    r_dmap = cv2_ximg.weightedMedianFilter(c_image,dmap,w_i,w_d,cv2_ximg.WMF_JAC)
    return r_dmap
```

Using the original image as reference, this function uses a weighted mechanism to filter the noise from the pixel-by-pixel comparison. The guided filter preserves the edges by calculating the weighted L_2 norm between the pixel in the reference image and the disparity map. CV2's ximgproc library provides an efficient computation of this weighted median kernel across the two arrays.

5.1.1.5 Overall function

Listing 5.5: guided_filter_disparity function

```
def guided_filter_disparity(S: StereoImage, D: int, algo='sd',
                    agg_filter=3, guided_filter:bool=True,
                    lr_check:bool = True,
                    plot=True, save_dir=None):
    """
    Implements the guided filter disparity computing of two stereo images.
    Args:
        S (StereoImage): data class containing left and right image arrays
        D (int): max number of disparities
        algo (str): choose either 'sd' or 'ad'
        agg_filter (int): aggregation filter size
        guided_filter (int): applies guided filters
        lr_check (int): applies left-right consistency check
        plot (bool): show disparity map
        save_path (str): save directory, set None to not save
    Returns:
        S (StereoImage): object with updated disparity map
    """
    cost_l2r = compute_matching_cost_pixel(S, D=D, l2r=True, algo=algo)

    if agg_filter > 0:
        cost_l2r = cost_aggregation_avg(cost_l2r, f=agg_filter)

    dmap_l2r = disparity_selection_wta(cost_l2r)

    if guided_filter:
        dmap_l2r = disparity_refinement_guided_filter(S.I_l, dmap_l2r)

    if lr_check: # create another disparity map from right to left
        cost_r2l = compute_matching_cost_pixel(S, D=D, l2r=False, algo=algo)
        if agg_filter > 0:
            cost_r2l = cost_aggregation_avg(cost_r2l, f=agg_filter)
        dmap_r2l = disparity_selection_wta(cost_r2l)
        if guided_filter:
            dmap_r2l = disparity_refinement_guided_filter(S.I_r, dmap_r2l)

        dmap_l2r, holes = disparity_refinement_con_check(S, dmap_l2r, dmap_r2l)
        dmap_l2r = disparity_refinement_densification(dmap_l2r, holes)
        if guided_filter:
            dmap_l2r = disparity_refinement_guided_filter(S.I_l, dmap_l2r)

    S.set_disparity_map(dmap_l2r)
    if plot:
        save_path = save_dir
        if save_dir is not None:
            save_path = os.path.join(save_dir,
            ↪  f"GDMAP_{S.name}_D={D}_{algo}_af={agg_filter}_gf={guided_filter}.jpg")
            print(save_path)
        S.show(save_path=save_path)
    return S
```

When the guided_filter_disparity function (Listing 5.5) is called, the stereo images will pass through the four stages in the disparity mapping pipeline. Similar to the overall function in Listing 3.6, the output is a StereoImage data class containing the left, right and disparity maps.

5.1.2 Results Discussion

5.1.2.1 Effects of Guided Filters and L-R Consistency Check

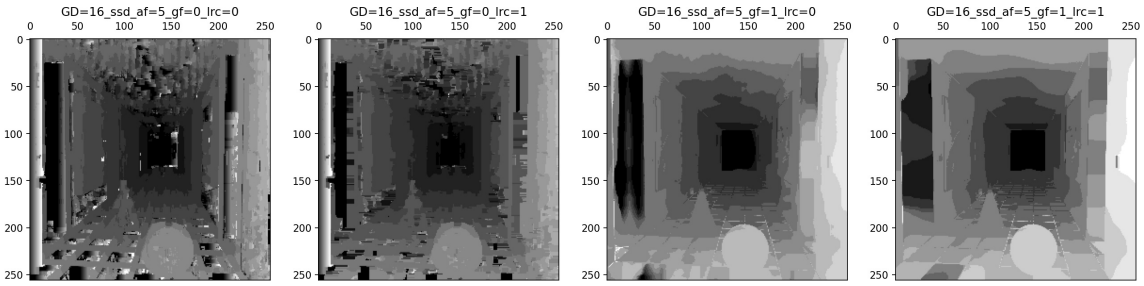


Figure 5.1: Corridor - Disparity maps after aggregation filters (1st), only l-r consistency check (2nd), only guided filters (3rd), both guided and L-R consistency check (4th)

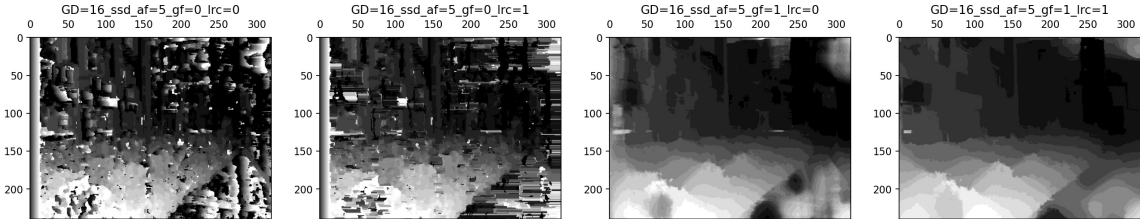


Figure 5.2: triclopsi2 - Disparity maps after aggregation filters (1st), only l-r consistency check (2nd), only guided filters (3rd), both guided and L-R consistency check (4th)

From Figure 5.1 and 5.2, the first image would represent the sliding window approach with a window size of 1, which is very noisy and the edges are rough. The second image shows the effect of disparity refinement with left-right consistency check and densification. It helps to fill in the white areas that were originally found in the first image, but the filled outlines are still very coarse. The third image is the output with only guided filters applied. By performing the weighted convolutions with the original image, the edges of shapes can be recovered, thus leading to better visualizations. The best disparity maps were produced with left-right consistency checking, densification and guided filters turned on. These maps not only have smooth color gradients, the outlines of shapes can also be vividly seen.

5.1.2.2 Tuned Parameter Outputs

The final disparity maps after parameter tuning are shown in Figure 5.3 and 5.4. The best parameters found for corridor scene are D=16, af=3 and gf=1, while for triclopsi2 scene are D=20, af=3 and gf=1. Note that the parameters do not represent the most optimal settings as they are purely combination of non-exhaustive search.

As compared to the results of the window-based approach as seen in Figure 4.9 and 4.10, the disparity maps from the guided filter approach are visually more well-defined. The images have lesser salt and pepper noises and the color gradients are smoother without loss of edges. The left-right consistency check also removed the same-valued vertical borders caused by the max disparity range. The outlines of the ball and cone in the corridor scene and the layers of grass along the road in the triclopsl2 scene can now be seen clearly.

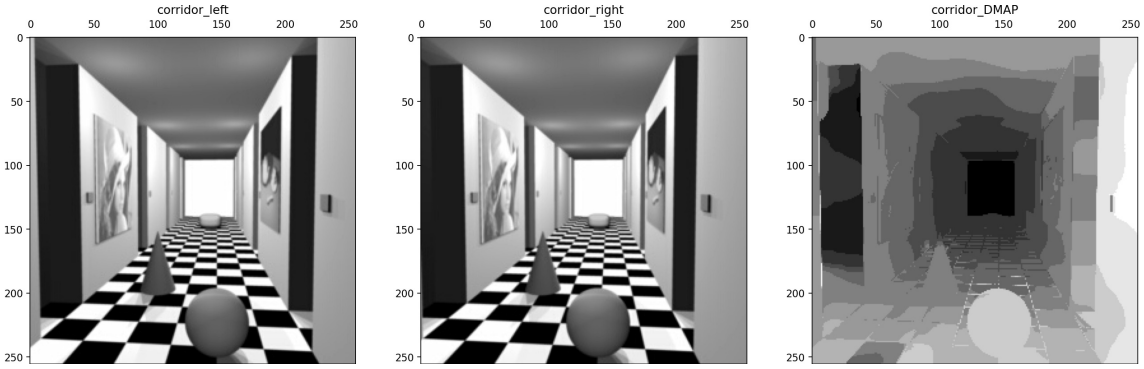


Figure 5.3: Corridor - Left, Right and Disparity map after Guided Approach

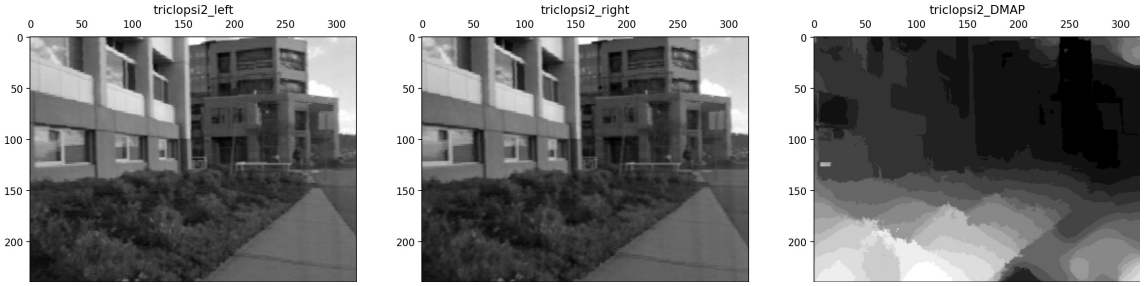
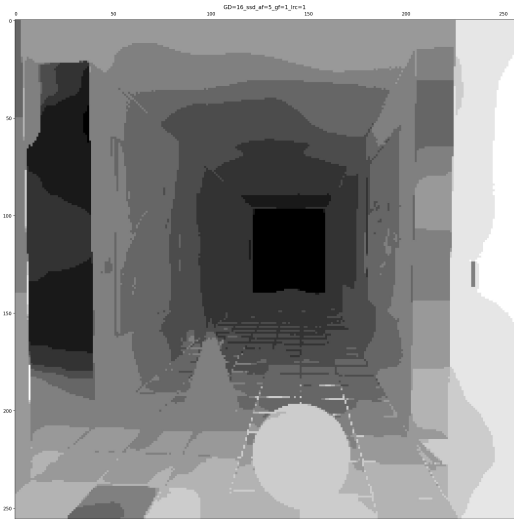
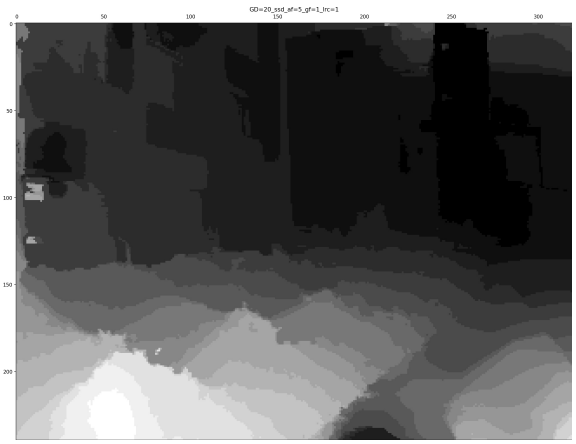


Figure 5.4: triclopsi2 - Left, Right and Disparity map after Guided Approach



(a) Corridor



(b) triclopsi2

Figure 5.5: Final Disparity Maps of guided filter approach

6. Conclusion

In this report, the general pipeline stages of disparity map computation were enumerated for the first task. It comprises of the matching cost computation step, cost aggregation step, disparity selection step and lastly, the disparity refinement step. For the second task, a traditional sliding window-based approach was implemented by closely following the procedure documented in task one. The source codes were written to adhere to the Python's PEP8 style guidelines for readability and re-usability.

In the third task, the outputs of the disparity maps from the sliding window approach were analyzed. As the approach is highly sensitive to the parameters used, the tuning process and the observations made for each parameter tuned were thoroughly discussed. After several rounds of non-exhaustive search, the best outputs based on the trials were displayed and the flaws of using the simple approach were listed.

Finally, the motivation of the implementation of the last task is to improve the quality and overcome some of the flaws found in the output of the simple approach. More sophisticated methods were used in the disparity refinement stage. The most critical changes were the addition of the left-right consistency check as well as the guided weighted filter [2] against the original image. The final disparity maps produced by the improvised approach are a lot more aesthetically appeasing with less noise and preserved edges.

References

[1] Rostam Affendi Hamzah and Haidi Ibrahim. "Literature survey on stereo vision disparity map algorithms". In: *Journal of Sensors* 2016 (2016).

[2] Asmaa Hosni et al. "Fast cost-volume filtering for visual correspondence and beyond". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.2 (2012), pp. 504–511.

[3] David G Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[4] Daniel Scharstein and Richard Szeliski. "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms". In: *International journal of computer vision* 47.1-3 (2002), pp. 7–42.

[5] Pauline Tan and Pascal Monasse. "Stereo disparity through cost aggregation with guided filter". In: *Image Processing On Line* 4 (2014), pp. 252–275.

[6] Le-Duo Yang. *Disparity_Estimation_Guided_Filter*. 2018.

Appendices

A StereoImage DataClass

Listing 6.1 shows the dataclass for loading the pairs of stereo images.

Listing 6.1: StereoImage Dataclass Object

```
from dataclasses import dataclass, field
@dataclass
class StereoImage:
    name: str
    I_l: np.ndarray = field(repr=False)
    I_r: np.ndarray = field(repr=False)
    W: int = field(init=False)
    H: int = field(init=False)
    DMAP: np.ndarray = field(init=False, repr=False)

    def __post_init__(self):
        self.H, self.W = self.I_l.shape
        self.I_l = self.I_l.astype('float32')
        self.I_r = self.I_r.astype('float32')
        self.DMAP = None

    def set_disparity_map(self, d_map):
        self.DMAP = d_map

    def show(self, save_path=None):
        subplot_value = 120 # show 2 images
        if self.DMAP is not None:
            subplot_value = 130 # show 3 images
        plt.figure(1, figsize=(20, 20))
        plt.subplot(subplot_value+1)
        plt.title(self.name + "_left")
        plt.tick_params(axis='both', which='major', labelsize=10,
                        labelbottom = False, bottom=False,
                        top = False, labeltop=True)
        plt.imshow(self.I_l, cmap='gray')

        plt.subplot(subplot_value+2)
        plt.title(self.name + "_right")
        plt.tick_params(axis='both', which='major', labelsize=10,
                        labelbottom = False, bottom=False,
                        top = False, labeltop=True)
```

```
plt.imshow(self.I_r, cmap='gray')

if self.DMAP is not None:
    plt.subplot(subplot_value+3)
    plt.title(self.name + "_DMAP")
    plt.tick_params(axis='both', which='major', labelsize=10,
                    labelbottom = False, bottom=False,
                    top = False, labeltop=True)
    plt.imshow(self.DMAP, cmap='gray')

if save_path is not None:
    plt.savefig(save_path, dpi=300)
plt.show()
```