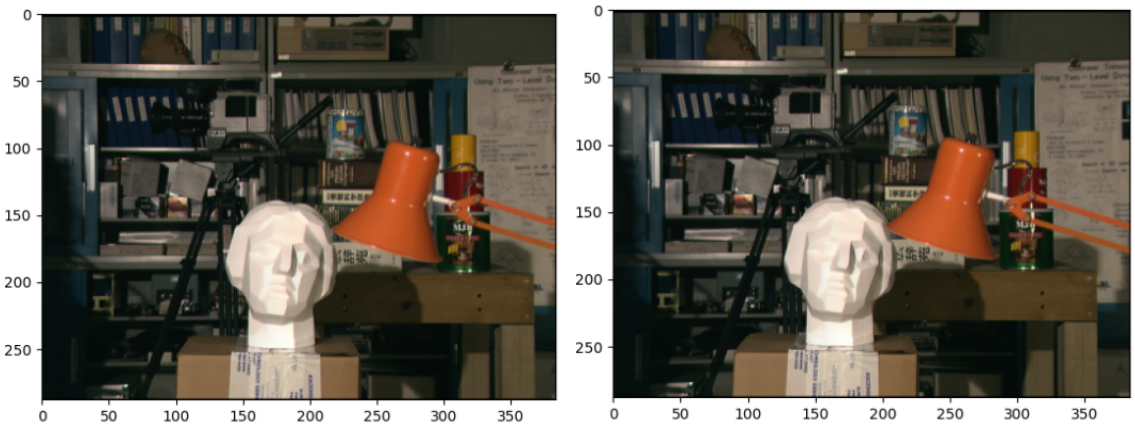
Computer Vision Exercise 2

Tomer Geva Maor Naftali

December 2021



(a) Left image used in the depth map computation *(b) Right image used in the depth map computation*

Figure 1: Source images used for the depth map computation

Part A: Distance Tensor Computation

1. Toy examples

In this subsection we are requested to compute the Sum of Squared Differences Distance (SSDD) between the matrices shown in (1) below with a window size of 3, i.e. for every location the SSDD will be computed including the 8 neighboring pixels.

$$Left = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{pmatrix} \tag{1a}$$

$$Right = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \end{pmatrix} \tag{1b}$$

The notation of the SSDD will be $(row, column, disparity\ value)$, i.e. SSDD(1,2,1) corresponds to row 1, column 2 and disparity value of 1, as depicted in the instructions. Assuming that the disparity range is 2, there are 5 possible values of shift: $-2, -1, 0, 1, 2$ with disparity values of $0, 1, 2, 3, 4$ respectively. The requested computation is shown in (2) below.

$$SSDD(1,2,2) = \left\| \begin{matrix} 2-1 & 3-1 & 4-1 \\ 7-2 & 8-2 & 9-2 \\ 12-3 & 13-3 & 14-3 \end{matrix} \right\|^2 = 1^2 + 2^2 + 3^2 + 5^2 + 6^2 + 7^2 + 9^2 + 10^2 + 11^2 = 426 \tag{2a}$$

$$SSDD(1,2,3) = \left\| \begin{matrix} 2-1 & 3-1 & 4-1 \\ 7-2 & 8-2 & 9-2 \\ 12-3 & 13-3 & 14-3 \end{matrix} \right\|^2 = 1^2 + 2^2 + 3^2 + 5^2 + 6^2 + 7^2 + 9^2 + 10^2 + 11^2 = 426 \tag{2b}$$

$$SSDD(2,3,0) = \left\| \begin{matrix} 8-2 & 9-2 & 10-2 \\ 13-3 & 14-3 & 15-3 \\ 18-4 & 19-4 & 20-4 \end{matrix} \right\|^2 = 6^2 + 7^2 + 8^2 + 10^2 + 11^2 + 12^2 + 14^2 + 15^2 + 16^2 = 1191 \tag{2c}$$

$$SSDD(2,3,1) = \left\| \begin{matrix} 8-2 & 9-2 & 10-2 \\ 13-3 & 14-3 & 15-3 \\ 18-4 & 19-4 & 20-4 \end{matrix} \right\|^2 = 6^2 + 7^2 + 8^2 + 10^2 + 11^2 + 12^2 + 14^2 + 15^2 + 16^2 = 1191 \tag{2d}$$

2. SSDD computation

A function was written to compute the SSDD between two images under the "ssd_distance" API. The function receives as input the two images, the wanted window size and the one-sided disparity range and outputs a 3-d matrix containing the Sum of Squared Distances (SSD) of eahc pixel w.r.t each disparity value is the two sided disparity range.

Part B: Naive Depth Map

3. Naive depth map computation

In this section we used the SSDD calculation to create a depth map. This was done by simply taking the disparity label matching the smallest SSD per pixel out of all the possible disparity values. The function was implemented under the "naive_labeling" API which is inputted by a 3-dimensional tensor depicting the SSDD values of two images and outputs a depth map.

4. Naive depth map results

The source images used for the depth map creation are shown in Figure (1) and the results of the naive depth map can be seen in Figure (2) below. We can see that the resulted depth map somewhat shows the different objects in the picture and locates the depth. Nevertheless, the same object is allocated to different depths and the results is not sufficient. One can see that the forward mapping of the different objects in the left photo to their respective location in the right photo is different and is dependant on the distance of the object from the camera. Therefore, As a side note, this effect is negligible in the panorama case since the distance from the camera to the scenery is extremely large.

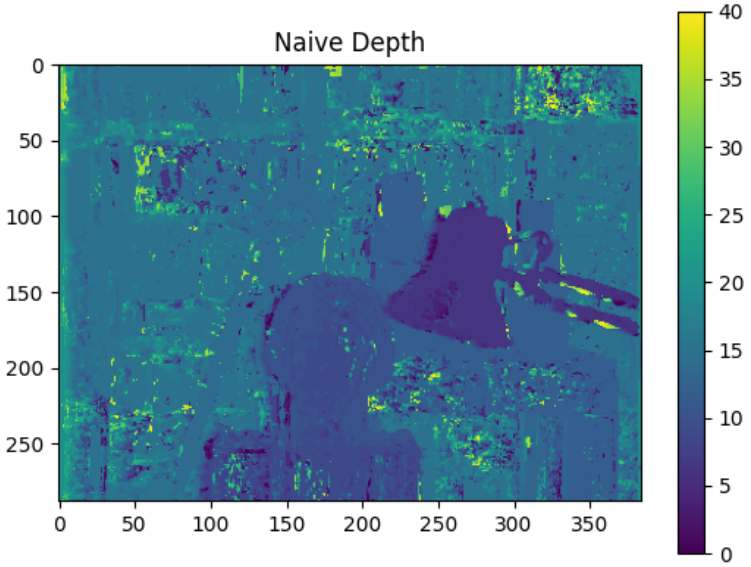


Figure 2: Depth map using the naive approach

Some of the problems in this method is the rapid change in the allocated depth for adjacent pixel locations designated to the same object. This may stem from a few reasons. Focusing on the lamp, we can see that the depth map allocates the majority of pixels in the depth map in the lamp location to the same depth but there are drastic changes in a few of the pixels. The drastic change in this may stem from the fact that when comparing the left image to the right image shifted to the left by some x pixel values we might land on an orange pixel in the shifted image that is very similar to the orange pixel of the left image which in turn yields low SSD. A slight change in lighting between the taken photos is sufficient to tip the scales towards the wrong decision in this case. Similar effect can be seen w.r.t the bookcase in the background and the head figure.

Another reason for bad results can be that the wrong shift might end by chance in a place matching another object but with a similar tone. This effect can also cause a false low SSD and tip the scale of the naive approach towards a wrong result.

Part C: Depth Map Smoothing Using Dynamic Programming

5. Loss function implementation

In this subsection a score function was implemented. The operation principles of the score function are as follows.

- Slicing the 3D SSDD tensor into 2D slices along the 1^{st} dimension, i.e. each slice hold all the labels for all the columns for a single row

$$S \in \mathbb{R}^{1 \times nCols \times nLabels} \tag{3}$$

  Where S denotes the SSDD slice, $nCols$ denotes the width of the image and $nLabels$ denotes the number of labels, in our case 41

- For each column in the slice we compute a loss function per label $L(d, col)$ such that

$$L(d, col) = \begin{cases} S(col, d) & col = 0 \\ S(col, d) + M(d, col) - \min\{L(:, col-1)\} & col > 0 \end{cases} \tag{4a}$$

$$M(d, col) = \min \begin{Bmatrix} L(d, col-1) & Same\ label\ as\ previous \\ P1 + \min\{L(d-1, col-1), L(d+1, col-1)\} & Difference\ of\ 1 \\ P2 + \min\{L(d+k, col-1) \forall k : |k| >= 2\} & Difference\ g.t.\ 1 \end{Bmatrix} \tag{4b}$$

Since each column is dependant on the previous column, this computation must be done iteratively via for loops and could not be done using vector calculations. This function was implemented in the "dp_grade_slice" API, which is inputted a slice and $P1, P2$ values and the API returns the loss score for each column and each disparity label.

6. Depth map using dynamic programming

The API used for the dynamic programming depth map was "dp_labeling" and is inputted by a 3D SSDD tensor and P_1, P_2 values. The API computes the depth map using the API from the previous section. The results can be seen in Figure (3) below.

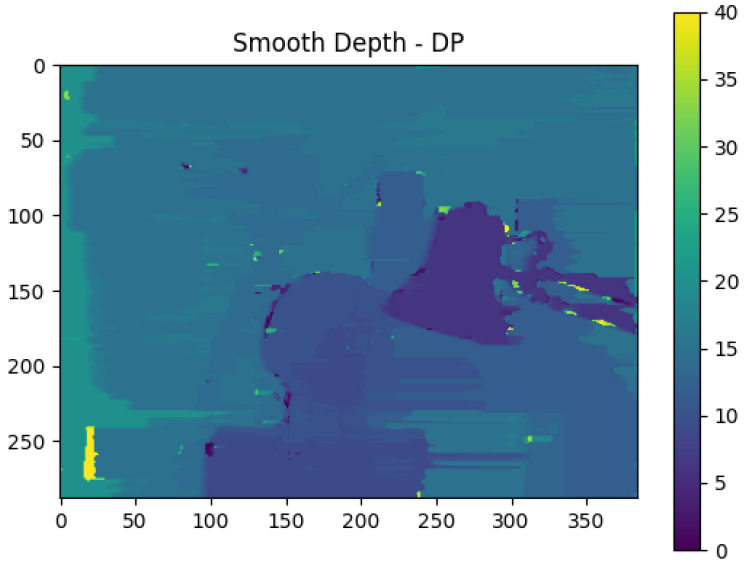
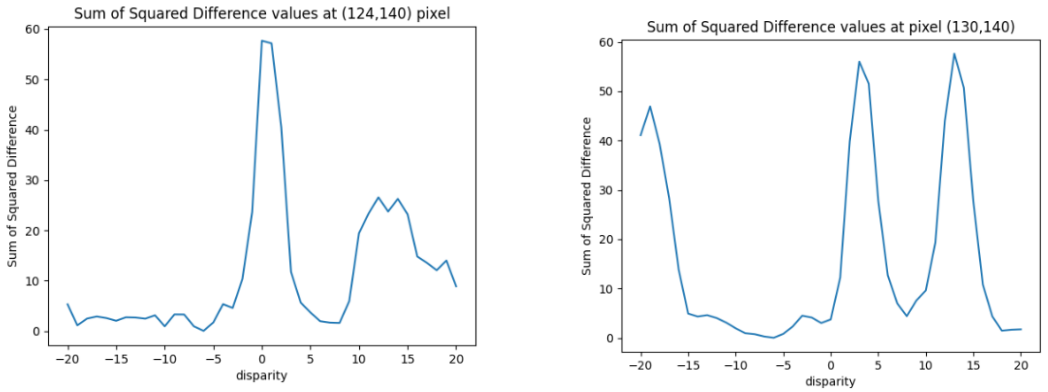


Figure 3: Depth map using $left \rightarrow right$ smoothing

7. Discussion - differences between dynamic programming and naive methods

Creating the depth-map, we try to locate the 'best' disparity value to adjust for each of the pixels. As one can see in figures (4a),(4b), the minimal SSD value may correspond with several disparity values, moreover, a local minimum might be the required disparity, for achieving continuity and coinciding with pixels' surrounding neighbors.

Using the naive approach (Figure (2)), we pair for each pixel the disparity value achieves the minimal SSD value, ignoring the discontinuity it may lead to. For each pixel the SSD was calculated for each disparity option. Figure (4a) shows an example of a pixle with low SSD values match different disparity options. An arbitrary change in lighting can favor one disparity over the other in this case.



(a) Sum of Squared Difference values at pixel (124,140) *(b) Sum of Squared Difference values at pixel (130,140)*

Figure 4: Source images used for the depth map computation

One can see that what seems as a 'high frequency noise', which was substantial in the naive method, is now gone using the DP smoothing (Figure (3)) and only the major artifacts remain (the different depths of the lamp rod for example). Moreover, in the absence of high frequency noise in the depth map one can more easily locate the desk and tripod which are closer than the bookcase but further away from the head figure (although the object are "smudged".

Nevertheless, we can see in Figure (3) significant inaccuracies in the smoothed depth map. Since the smoothing was performed from left to right, the changes from closer objects to object further away is not performed accurate enough. This causes the noticeable "smudges" of the different depths from the left to the right.

The last significant point is the computation time. The naive depth map computation took approximately 8 [*msec*] long whereas the dynamic programming smoothing computation took around 4.5 [*sec*] to complete. This degradation in runtime might not be worth the difference in results, depending on the use-case. Let us note that the runtime of the single direction dynamic programming can be sped up numerically using decorators as well as by computing the dynamic programming for all the rows vectoricaly.

Part D: Depth Image Smoothing Using Semi-Global Mapping

8 + 9. Slice extractor function implementation

In this subsection we will provide an explanation regarding the slice extractor function. The exercise defined 8 directions depicted in Figure (5) below. Let us note that direction 1 is essentially the direction used in Part C above.

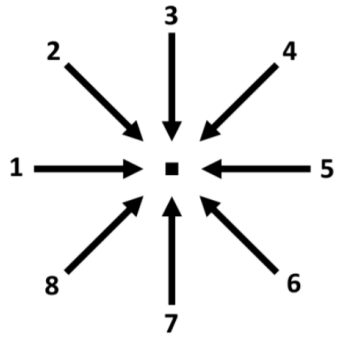


Figure 5: Slice direction definition and enumeration

The slice extractor was implemented using two APIs.

extract_slices API

The "extract_slices" API which receive a 3D SSDD tensor, a direction option depicted in the enumeration below and three Boolean values {transpose, fliplr, flipud}

1. : $left \rightarrow right$ rows
2. : $top\ left \rightarrow bottom\ right$ diagonals

The slice extractor then slices the SSDD tensor according to the direction inputted.

1. : slicing rows from left to right and extracting the coordinates of horizontal slice
2. : slicing rows diagonally, extracting the coordinates of each diagonal slice

The "extract_slices" API then returns two dictionaries "slices_dict, indices_dict" with numerical keys. The slices_dict holds the slices, which in the diagonal case may vary in length and the indices_dict holds the matching (row, col) coordinates of the slice. Let us note that before adding the indices of the slice to indices_dict, one or more of the following operations is done for all the coordinates in the slice, according to the three Boolean inputs:

- If transpose is true, switches the (x, y) coordinate tuple to (y, x)
- If fliplr is true, switches the (x, y) coordinate tuple to $(width - 1 - x, y)$
- If flipud is true, switches the (x, y) coordinate tuple to $(x, height - 1 - y)$

Let us note that if multiple flags are true the actions stack upon each other and the flip functions happen first. See examples below.

- If transpose and fliplr are true, switches the (x, y) coordinate tuple to $(y, width - 1 - x)$
- If transpose and flipud are true, switches the (x, y) coordinate tuple to $(height - 1 - y, x)$

The "extract_slices" API is used by the second API mentioned above to extract the slices in the wanted direction.

orient_direction_and_extract_slices API

The "orient_direction_and_extract_slices" API receives two inputs. The first is an SSDD tensor and the second is the wanted direction index. This API transforms the SSDD tensor before calling to the "extract_slices" API with the needed index transformations. The transformation to the SSDD tensor is done to match the wanted direction and the direction which is supported by the "extract_slices" API, and is doing as follows per direction.

1. Does not change the SSDD tensor and calls "extract_slices" for horizontal slices
2. Does not change the SSDD tensor and calls "extract_slices" for diagonal slices
3. Re-ordring the dimensions of SSDD tensor $[1, 2, 3] \rightarrow [2, 1, 3]$ and calls "extract_slices" for horizontal slices with "transpose" flag
4. Flips the 1^{st} dimension and calls "extract_slices" for diagonal slices with "fliplr" flag
5. Flips the 1^{st} dimension and calls "extract_slices" for horizontal slices with "fliplr" flag
6. Flips the 1^{st} *and* 2^{nd} dimensions and calls "extract_slices" for diagonal slices with "fliplr" and "flipud" flags
7. Flips the 2^{nd} dimension, then re-orders the dimensions $[1, 2, 3] \rightarrow [2, 1, 3]$ and calls extract_slices" for horizontal slices with "transpose" and "fliplr" flags
8. Flips the 2^{nd} dimension and calls "extract_slices" for diagonal slices with "flipud" flag

12. Debugging the directional results

To debug the API written in subsection 8 + 9, additional API was implemented called "dp_labeling_per_direction". This API receives the SSDD tensor and the two penalties P_1, P_2 and returns a dictionary with integer keys matching their respective directions, and the values are the depth maps as computed in the respective directions. The results of all 8 directions can be seen in Figure (6) below. We can see that for direction 1, the depth map coincides with Figure (3) and the "smudges" of each depth map match the directional orientation of the dynamic programming used.

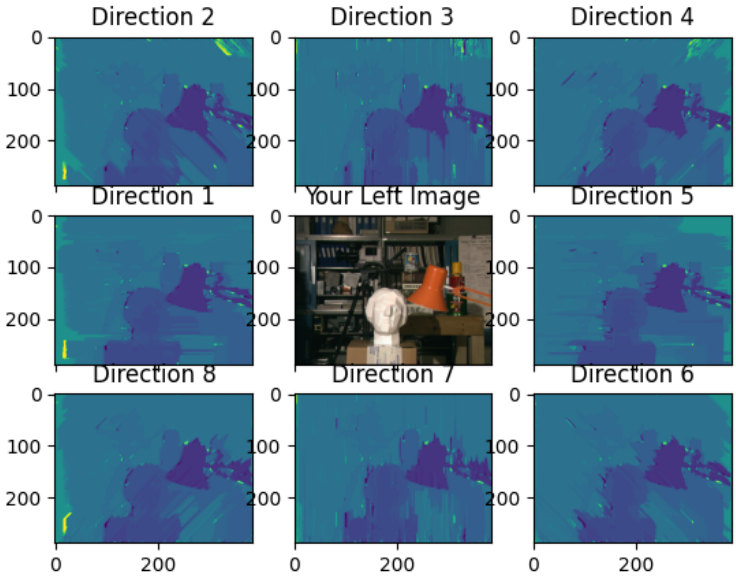


Figure 6: Depth map per dynamic programming direction

10. Semi-Global-Mapping (SGM) depth map computation

A methid with computes the depth map using SGM was implemented under the "sgm_labeling" API. This method computes the L score matrix using dynamic programming in each of the 8 directions, averages them and uses the averaged L score matrix to derive the depth map. The API is inputted with an SSDD tensor and P_1, P_2 values and the API returns a depth map, mapping each pixel in the image to a different disparity value using the SGM method with the 8 directions defined in the previous sections. The result can be seen in Figure (7) below.

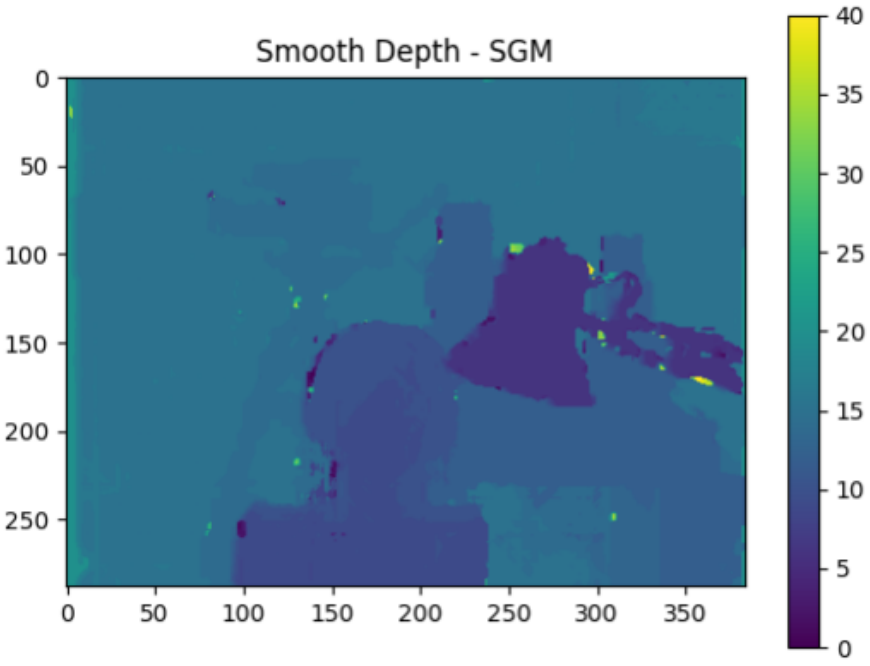


Figure 7: Depth map, Semi Global Mapping

11. Discussion - difference between SGM and naive approaches

The SGM depth map computation (Figure (7)) shows significant improvement over the naive depth map computation (Figure (2)). In figure (6) we can see the depth maps resulting in smoothing w.r.t the 8 directions defined above. We note "smudges" in the respective direction of the smoothing but when averaging the loss from the 8 directions, the "smudges" cancel out and what is left is the resulted SGM depth map. We note that the high frequency noise is almost entirely canceled, for example by seeing that the bookcase's depth is uniform and the tripod + camera is clearly separated from the bookcase. We can further see that the face sculpture is classified in multiple depths, which makes sense because the sculpture has some depth and due to the close distance of the sculpture this depth can be separated. However, the SGM result still has some room for improvement. This can be seen in the high label artifacts, mostly found at the perimeter of the lamp.

One place where the naive approach out-performed the SGM is at the right side of the depth map. We can see that the lamp is supported by two orange rods. These rods are classified better in the naive approach than the SGM and the space between them is better classified as background. In addition, wjen taking into account the time aspect of the computation, the naive approach takes approximately 8 $[msec]$ to compute whereas the SGM does dynamic programming 8 times, which takes approximately 40 $[sec]$ to compute, which is a significant difference, but the results seems to justify the additional time.

Part E: Our Own Images

We chose the following pair of rectified images, taken from middlebury stereo data.

(a) Our left image used in the depth map computation *(b) Our right image used in the depth map computation*

Figure 8: Source images used for the depth map computation

The results of the naive depth map can be seen in Figure (9) below. One can see that the resulted depth map somewhat shows the different objects in the picture and locates the depth.
We have found that for this very set of images, other parameters' values better be used, and we used these for this entire "Our Own" Part.

- P1 = 0.1
- P2 = 1.5
- Window size = 11
- Disparity range set to 12

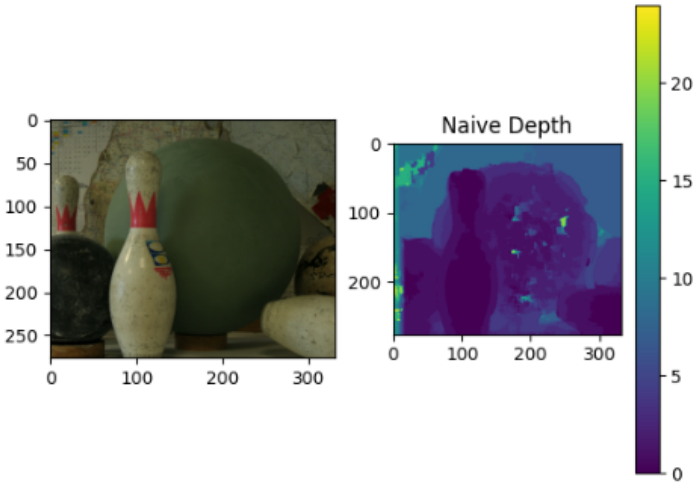


Figure 9: Depth map using the naive approach

The SGM depth map computation (Figure (11)) shows significant improvement over the naive depth map computation (Figure (9)). In figure (10) we can see the depth maps resulting in smoothing w.r.t the 8 directions defined above.

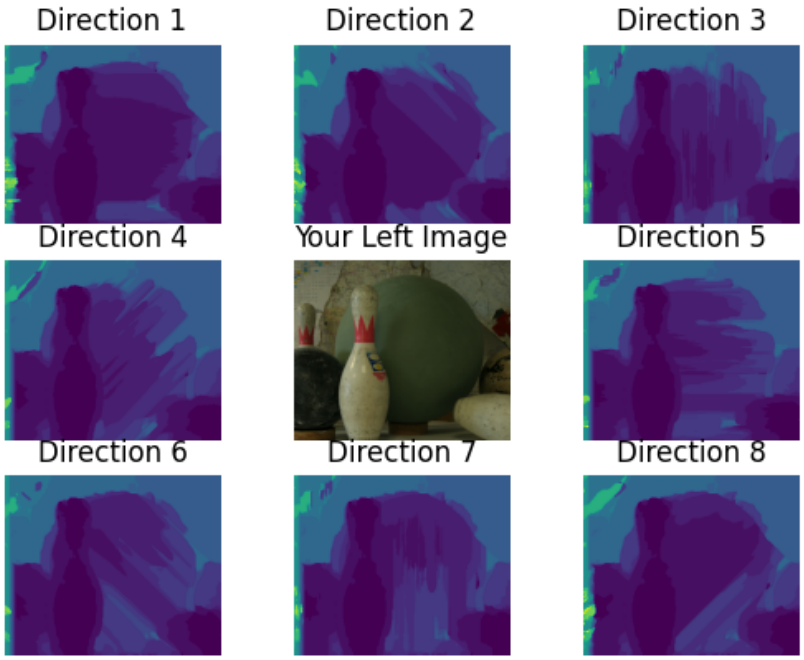


Figure 10: Depth map per dynamic programming direction on our images

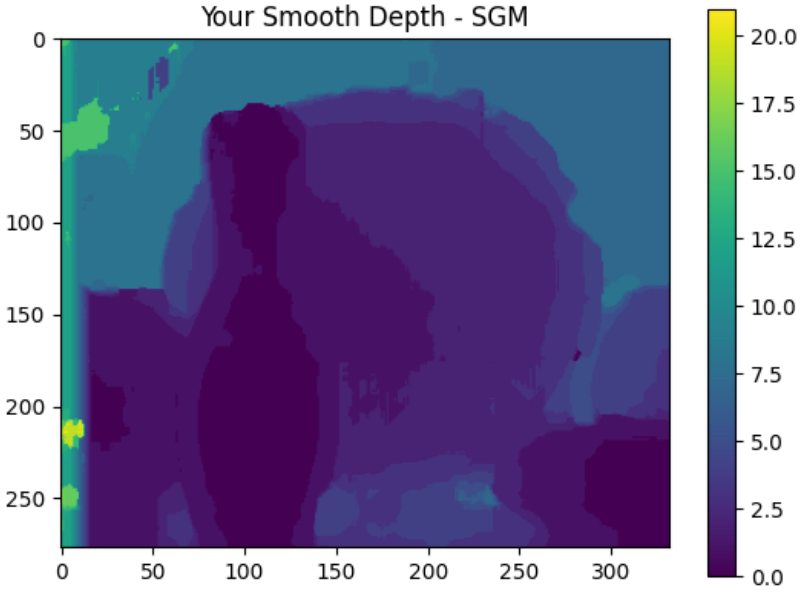
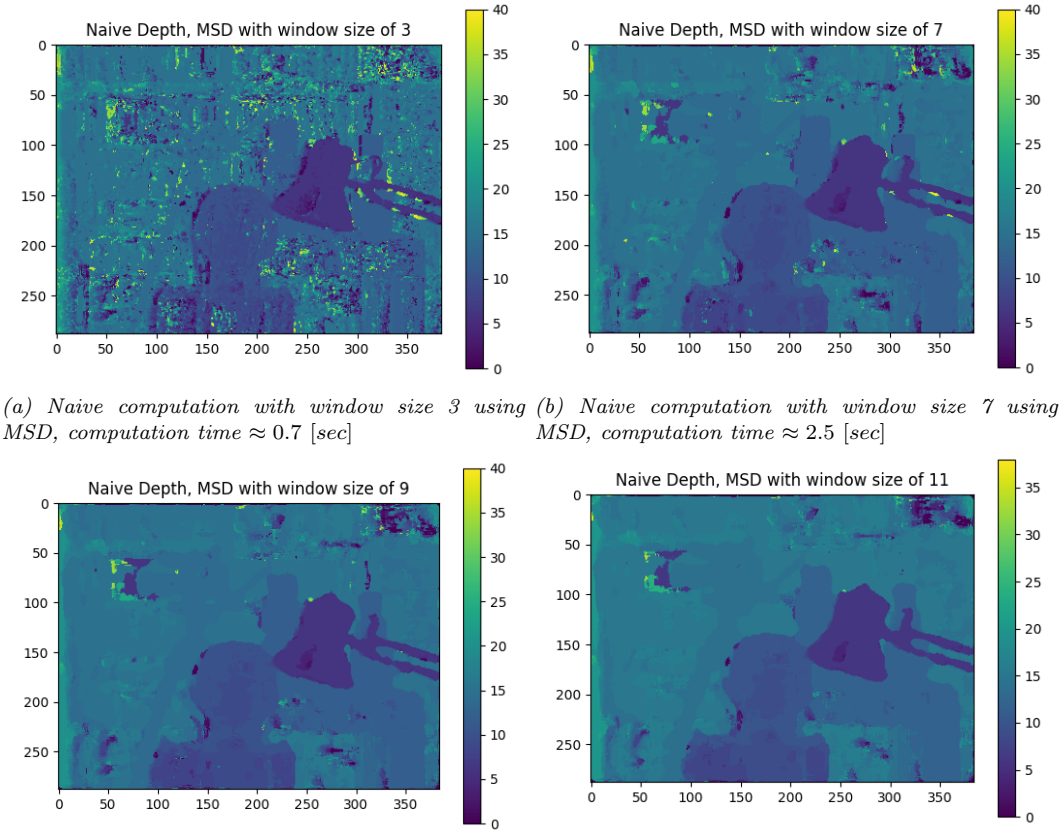


Figure 11: Depth map, Semi Global Mapping

Bonus part

14. Median square distances metric

As the title suggests, we propose to replace the SSD metric with the Median of Square differences (MSD) metric. The main advantage of the median over the SSD is that in the window where the SSD computation is done most of the pixels seem to match but the background pixels in each case are drastically different. This may cause an artificial spike in the SSD and we might prefer a different disparity value because of that. The median operator over the window resolves this problem by via performing the square distances if each pixel in the window and choosing the disparity according to the minimal median value. Using the median we can practically ignore all the very noisy difference of the different depths and match the appropriate disparity. Let us note that for the median operator to perform well, one needs to use large window size to allow statistical accuracy for the median operation. The results of the naive labeling using MSD and different window sizes is seen in Figure (12) below. We can see that for a window size of 3, the SSD in Figure (2) out performs the MSD and this is due to the fact that the median operator performs very poorly with 9 samples. As the window size increases, the MSD gradually out-performs the SSD operator. Nevertheless, when increasing the window size too much, the window might contain too much pixels which may lead to miss-classification. As a result, a bigger window than 3 should be used but the increase should not be substantial and a window size of 9 should suffice. For a window size of 9 the MSD criterion successfully categorizes most of the pixels and the high frequency noise which is present in the SSD naive depth map is successfully canceled. More over, we can see that for window size of 9 most of the big artifacts are cancelled using the MSD.



(a) Naive computation with window size 3 using MSD, computation time ≈ 0.7 *[sec]*

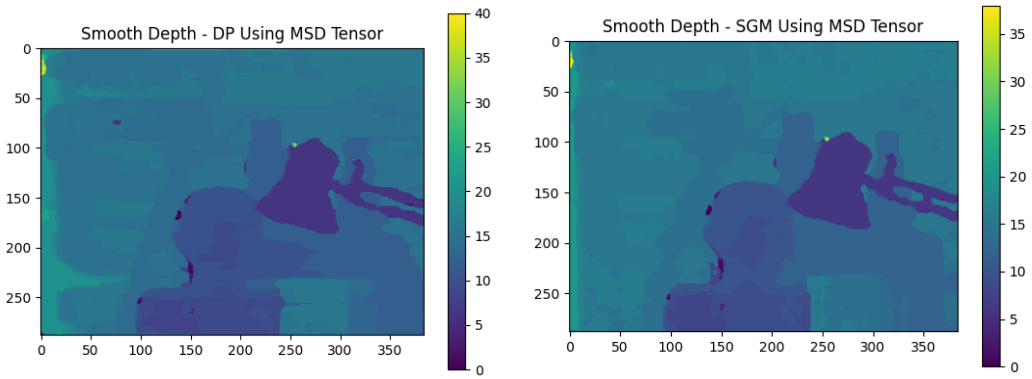
(b) Naive computation with window size 7 using MSD, computation time ≈ 2.5 *[sec]*

(c) Naive computation with window size 9 using MSD, computation time ≈ 3.5 *[sec]*

(d) Naive computation with window size 11 using MSD, computation time ≈ 5 *[sec]*

Figure 12: Naive depth maps using MSD with different window sizes

Let us note that since the naive result is significantly better than the SSD method, we can reduce the penalties P_1, P_2 from 0.5 and 3 to 0.05 and 0.3 respectively. When applying smoothing with direction 1 to the MSD tensor, the results can be seen in Figure (13a) below. We can see that for the left to right smoothing the high frequency noise is gone, as in the SSD tensor result of the left to right smoothing. We further note that we see similar "smudges" in the direction of the smoothing similar to the SSD tensor result. Similar to the naive approach, we can see that all the artifacts of high disparity values which were present using the SSD tensor are not seen using the MSD tensor, except the single artifact seen on the top right of the lamp. In addition, we see on Figure (3) a big artifact on the bottom left corner of the depth map which is not seen when using the MSD tensor and the orange bars of the lamp are extracted as two separate bars in the MSD case.



(a) Left → right smoothing using MSD *(b) Semi Global Mapping using MSD tensor*

Figure 13: DP smoothing (13a) and SGM (13b) using MSD tensor

When comparing the SGM results we can see that the SST tensor result (Figure (7)) has a better separation of the background (bookcase) than the MSD tensor result. In addition, we can see that the string of the lamp is successfully classified as part of the lamp in the SSD tensor result whereas only part of the string is classified with the same disparity as the lamp using the MSD result.

However, we can see that using the MSD tensor, all the large disparity label artifacts in Figure (7) are not seen in Figure (13b) except a small single artifact at the top of the lamp and another relatively small artifact at the top left of the depth map. In addition, orange bars of the lamp are almost completely separated in the SGM result using the MSD.

To use extract the MSD tensor instead of the SSD tensor, an additional input was added to "ssd_distane" API. the input is named "get_median" and has a default value of False for backward compatibility. when calling the "ssd_distane" with "get_median" as False, the API returns the SSD tensor and if "get_median" is True, the API returns the MSD tensor.

15 + 16. SGM MLSE smoothing

The method we suggest performs, as the name suggests, performs Maximum Liklihood Sequence Estimation (MLSE) of the depth map. We begin with a forward pass over the SSDD tensor (or MSDD tesnsor) slices in similar 8 directions as depicted in Figure (5) above. Each slice has a size of $1 \times N \times D$ where N might vary for diagonals and D denotes the number of disparity labels. For each direction we compute the cumulative loss shown in equation (5) below, denoting the SSDD (or MSDD) tensor slice as $S(\cdot,\cdot)$.

$$L(d,col) = \begin{cases} S(col,d) & col = 0 \\ S(col,d) + M(d,col) & col > 0 \end{cases} \tag{5a}$$

$$M(d,col) = \min \left\{ \begin{array}{rl} L(d,col-1) & Same\ label\ as\ previous \\ P1 + \min\{L(d-1,col-1), L(d+1,col-1)\} & Difference\ of\ 1 \\ P2 + \min\{L(d+k,col-1) \forall k : |k| >= 2\} & Difference\ g.t.\ 1 \end{array} \right\} \tag{5b}$$

The novelty is that we also create an additional indexing matrix, holding the path taking to reach the loss computed in (5). This indexing matrix is denoted as $\tilde{I}$ and has a size of $N \times D$ per slice, and holds the disparity label of the previous cell in the slice that led to the minimal cost.

$$\tilde{I}(col, d) = \arg\min\{L(d, col)\} \tag{6}$$

After passing through the entire slice, we find the disparity label of the lest cell in the slice by the minimal loss criterion, and start a backward pass through the indexing matrix, extracting the indices of the slice which led to the minimal cumulative loss. This can be seen in the example Figure (**??**) with a toy example with 4 possible disparity values and 5 columns in the slice.

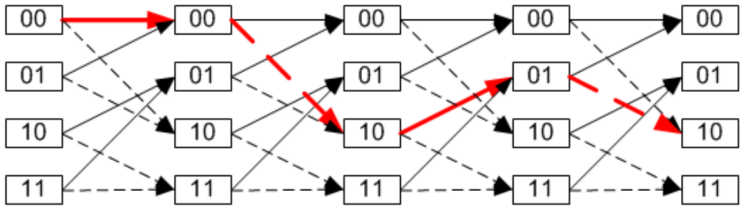


Figure 14: MLSE toy example. picture taken from "https://commons.wikimedia.org/wiki/File:Convolutional_code_trellis_diagram.png"

Here we can see that the maximum liklihood sequence is highlighted in red, and for this toy example, the slice disparity labels will we $\{0, 0, 2, 1, 2\}$. This MLSE method is particularly useful in communications, hence the example disparity labels being represented in their binary form.

This method should, in theory, results in twice the computations time, since fore each of the 8 directions we perform a forward and backward pass over the slice. Nevertheless, this SGM MLSE computation results in approximately similar computation time due to the fact that we managed to compute all the sliced in parallel using vectoring computations.

The results of the $left \rightarrow right$ MLSE smoothing can be seen in Figure (15) below.

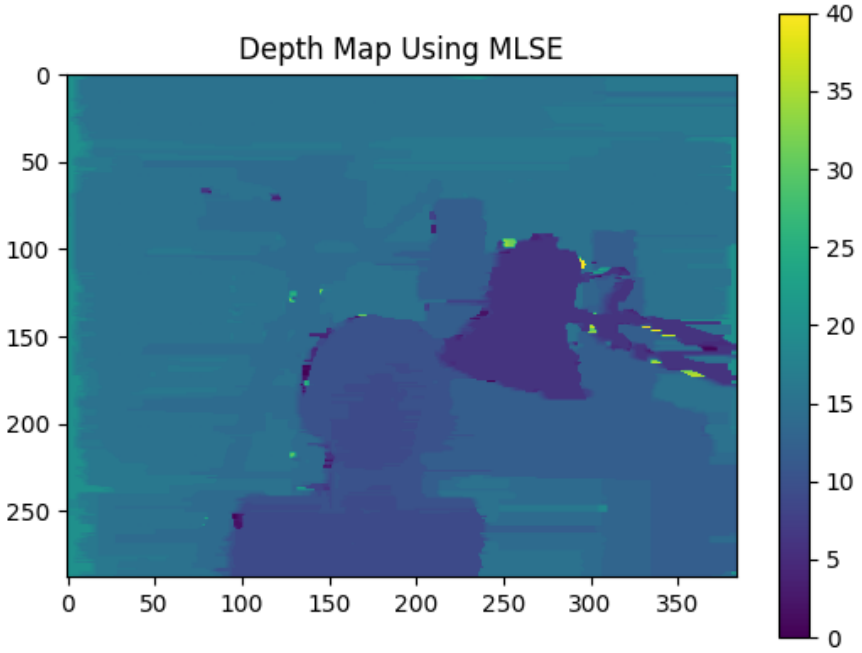


Figure 15: MLSE results using $left \rightarrow right$ smoothing

When comparing to the dynamic programming in Figure (3) we can see that the MLSE resulted with less "smudges" and less artifacts. Im addition, we see that the background is more uniform and the tripod, table and cans can be better seen. Moreover, we see that the face sculpture is better isolated and the form of the sculpture appears closer, which he is due to the 3D shape of the face. We can further note that the orange bars of the lamp are better classified as part of the lamp, and the space both around and between the bars is better classified as background.

The computation time for Figure (15) was approximately 3 [*sec*] which is $\frac{2}{3}$ of the time is took to compute the dynamic programming smoothing due to the parallel computation for all slices.

The result of the uni-directional smoothing might suffice but the presence of the "smudges" can be improved using the same principle of the SGM approach. The MLSE smoothing was done in the same 8 directions and the resulting depth maps were averaged and passed through a median filtering block with window size of 7 to overcome small artifacts, outliers and to smooth the different layers. The results of the SGM MLSE are s een in Figure (16) below.

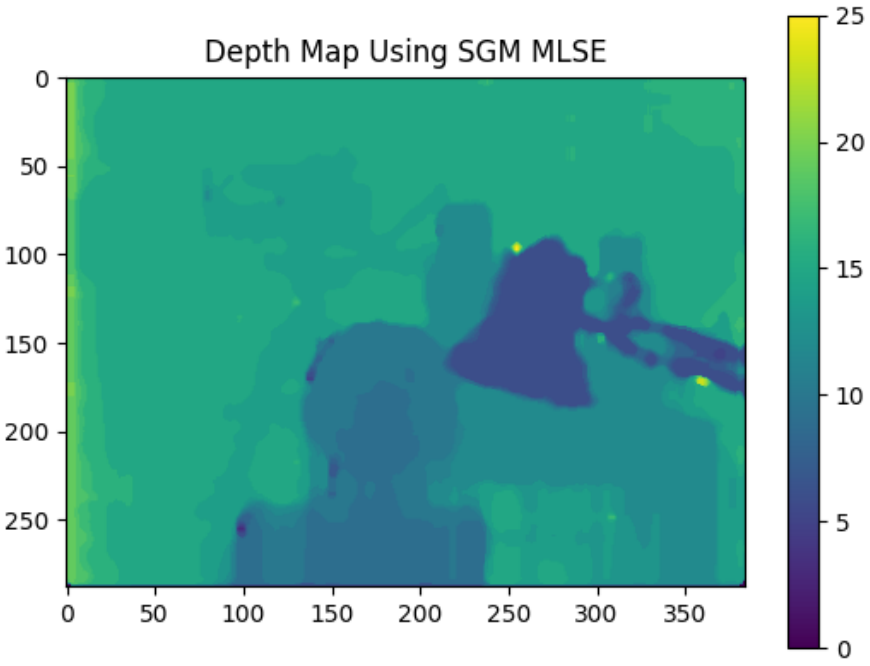


Figure 16: SGM MLSE depth map results

We can see that compared to the SGM results shown in (7), the high label artifacts are almost gone with a few exceptions at the vicinity of the lamp. We can see That the background is almost uniform which is a signal that the entire bookcase is at the same depth. Moreover, we can now see that the orange bars of the lamp have been separated and we can identify two bars. Nevertheless, we can see a slight degradation in the tripod w.r.t. Figure (7).

When taking into account the time computations, we can now see that due to the paralleling of the computation, the 8 directions was done in approximately 25 [*sec*] which is significantly faster than the $\approx$ 40 [*sec*] of the SGM dynamic programming used to compute Figure (7).

To use the custom SGM MLSE algorithm, we implemented a new "sgm_labeling_custom" API. This API receives a 3D tensor (SSDD or MSDD or any other) and two penalty values P_1, P_2 and outputs the depth map computed using the SGM MLSE algorithm.