SN: 19055854

COMP0005 - Algorithms

9 March 2020

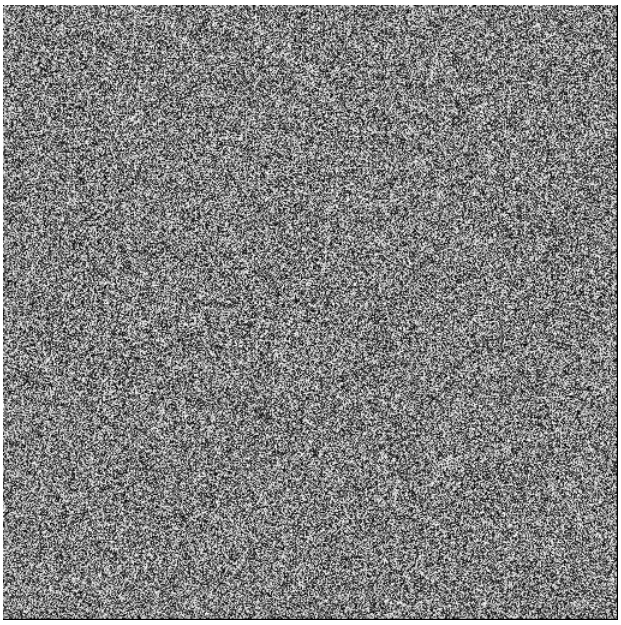
Stereo Matching Algorithm Using Dynamic Programming

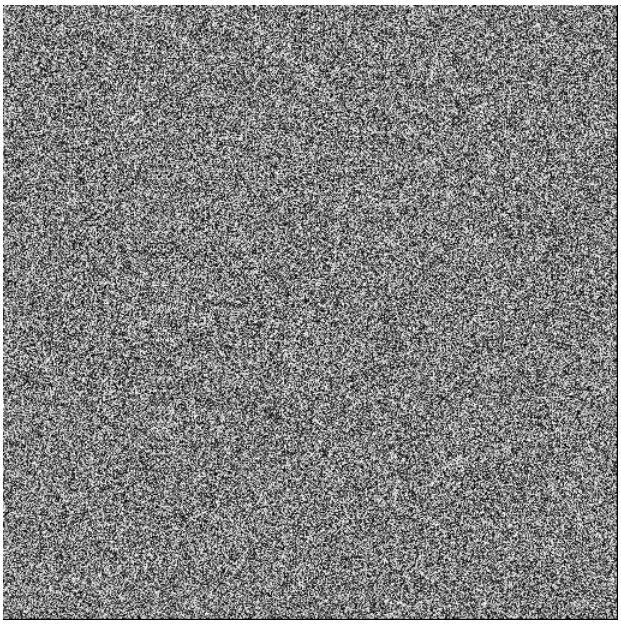
The following report is regarding an attempt to implement a dynamic programming algorithm (Cox et al. 341).

Implementation: The algorithm will be implemented using Python programming language version 3.7. The code for the algorithm can be found at Appendix 1.

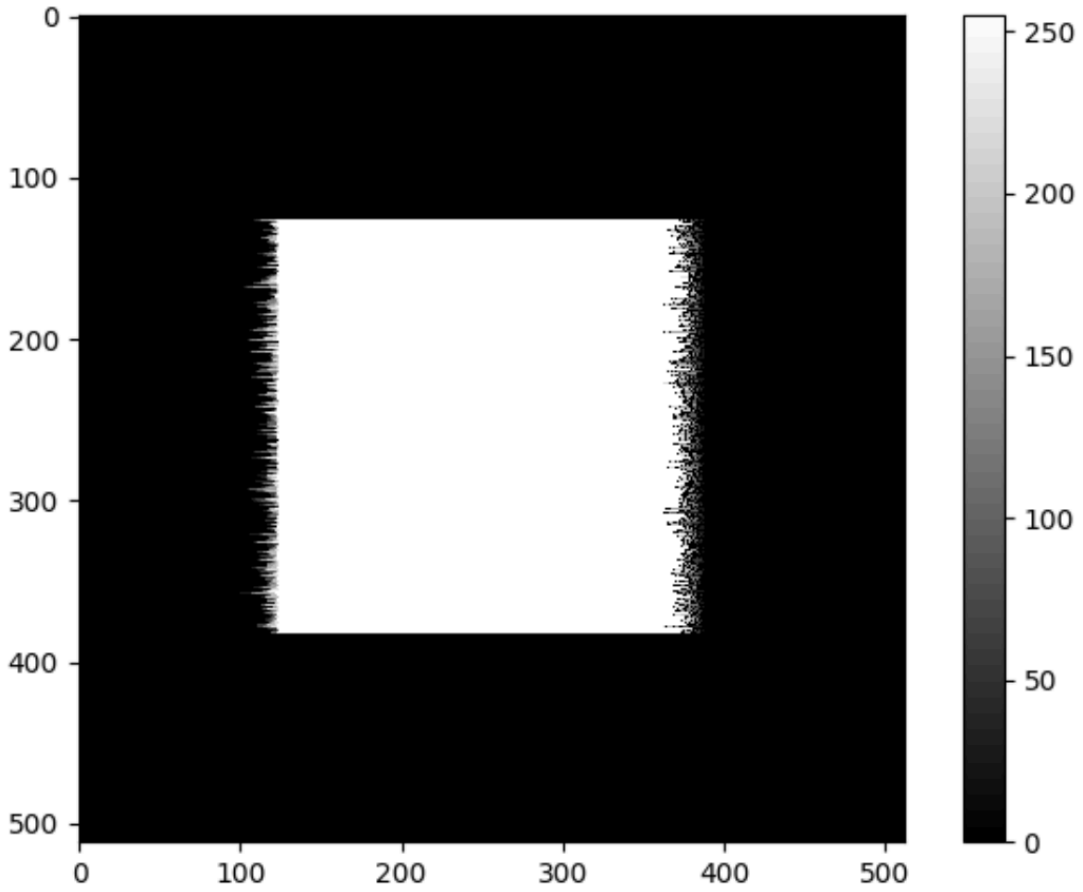
Q1: Why do matching errors occur for the binary random dot stereograms?

To examine this question, first, let us create synthetic the random-dot images, using the code specified in Appendix 2.



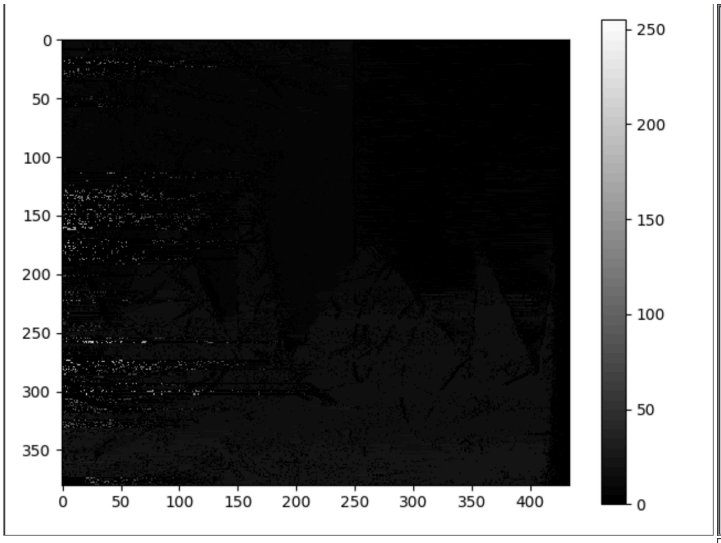


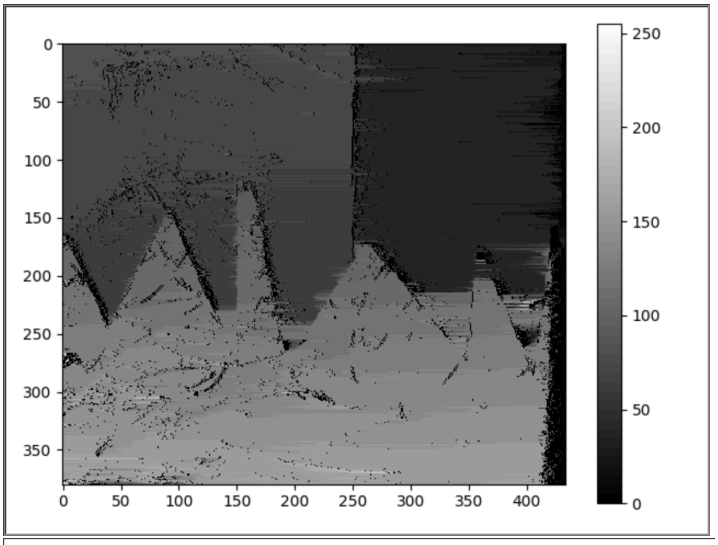
Both images contain a smaller random-dot image, however, in the right image, that sub-image has been slightly shifted. This would allow us to run it through stereo-matching algorithm, demonstrating that one can infer depth, even from images comprised of entirely random greyscale pixels (255). Below we can see the image produced by a stereo-matching algorithm, taking in the random-dot images as its parameters:

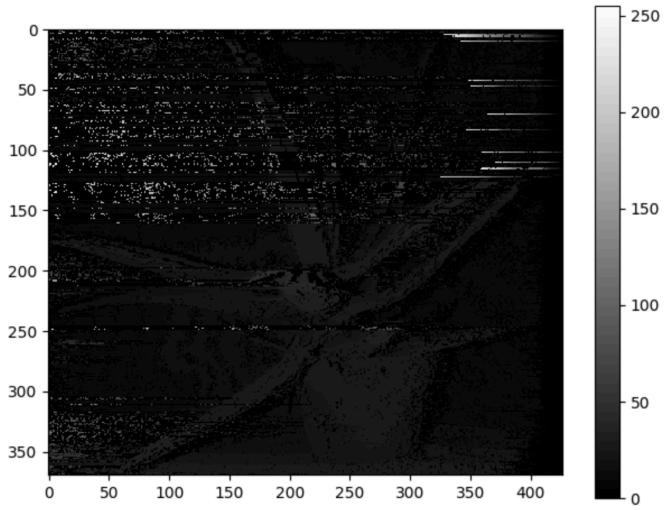


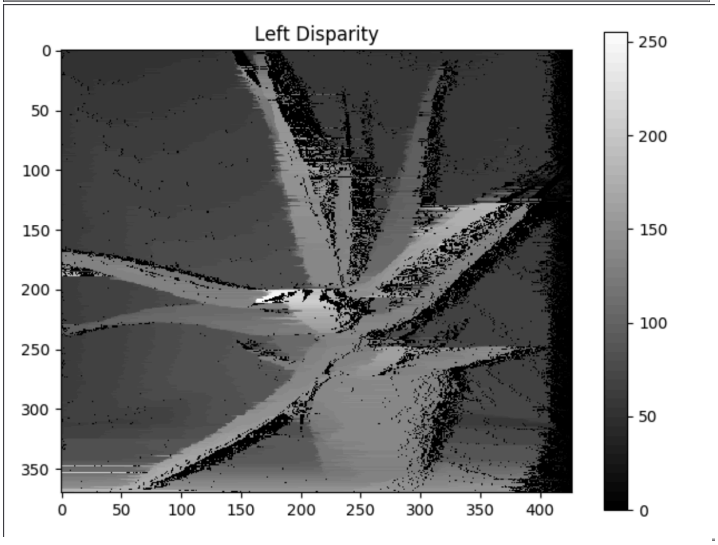
The reason why errors may occur in the random-dot stereograms could potentially be due to the fact that when comparing the pixel values across single rows of both images, there may be more than one value that qualifies as a match across both rows. Therefore, the choice the algorithm “faces”, is to determine which is the minimum, the sum of the cost and occlusion, or given lower difference in pixel intensities, the cost matrix (on a diagonal path) with the matching cost, consequently, resulting in distortions in the final disparity map.

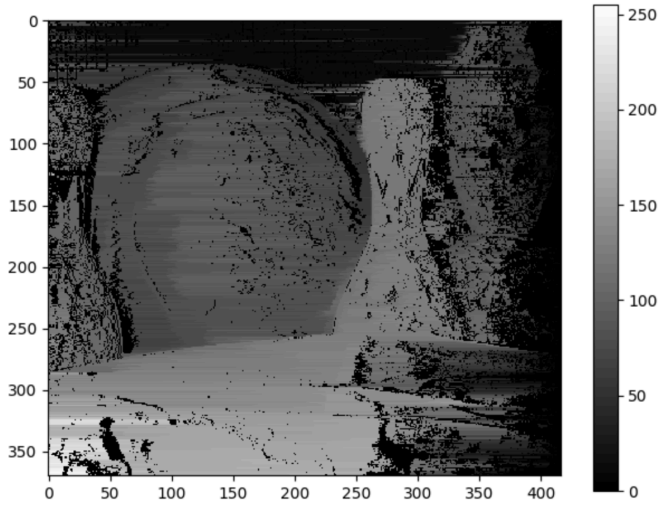
Q2: Investigate how the algorithm performs on other images as the occlusion cost is varied.

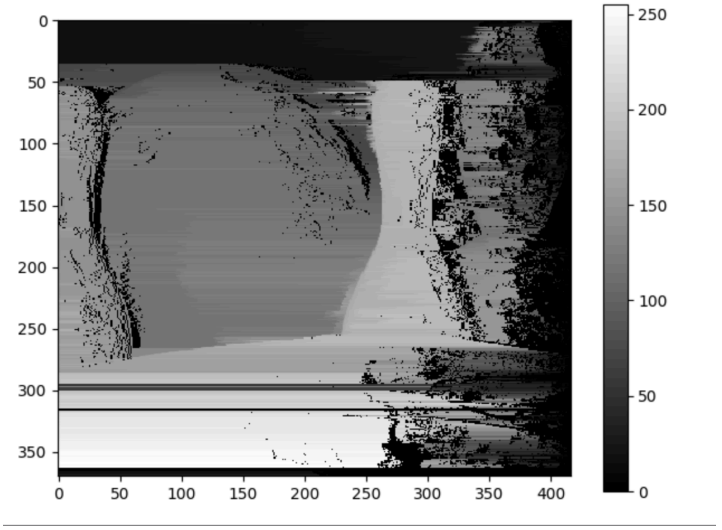






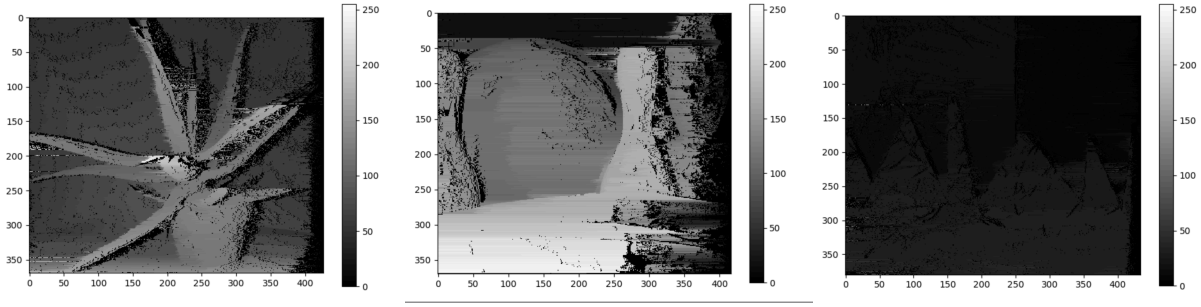






Observations resulting from examination of Pair 1, 2, and 3 respectively:

Images above, on the left were produced using the initially given occlusion value of 3.8. As one can observe, the image is rather dark when compared to the image on the right, produced with an occlusion value of 12. Testing demonstrated that the depth can be clearly inferred (particularly in disparity maps with higher occlusion constant), in accordance to the greyscale value intensity (varying from 0 to 255) - the higher the value, the closer an actual object is relative to the camera position. In particular, clearer images were produced with an occlusion value around 8-12:



(Disparity maps with occlusion value of 7)

Q3: For string matching, what is the equivalent of occlusion?

A potential equivalent of occlusion in string matching could be any possible change done to either string but not the other, in particular, a shift combined with substitution of values in an opposite direction of the shift. It hence results in new characters being introduced, that are not present in another string, similar to an object hidden away by one image, but revealed (or partially revealed) by another image.

Work Cited

Cox, Ingemar J., et al. Stereo without Disparity Gradient Smoothing: A Bayesian Sensor Fusion Solution. Princeton, NEC Research Institute. The British Machine Vision Association and Society for Pattern Recognition, www.bmva.org/bmvc/1992/bmvc-92-035.pdf.

Appendices

Appendix 1: Python Code for Stereo-Matching Algorithm

```python
import numpy as np
import cv2
from matplotlib import pyplot as plt


def Matching(num_of_columns, occlusion_value):
    CostMatrix = np.zeros((num_of_columns, num_of_columns))
    for i in range(0, num_of_columns):
        CostMatrix[i][0] = i * occlusion_value
        CostMatrix[0][i] = i * occlusion_value
    return CostMatrix

def calculate_STEREO(ImageRight, ImageLeft):
    num_of_rows = ImageRight.shape[0]
    num_of_columns = ImageLeft.shape[1]

    # Initiate the disparity arrays with 0's
    disparity_left = np.zeros((num_of_rows, num_of_columns))
    disparity_right = np.zeros((num_of_rows, num_of_columns))

    # Set occlusion value (given as 3.8)
    occlusion_value = 3.8

    for all_rows in range(0, num_of_rows):
        # Inform the user of the process made so far
        print("processing " + str(all_rows) + " / " + str(num_of_rows))
        DisparityMatrix = np.zeros((num_of_columns, num_of_columns))
        CostMatrix = Matching(num_of_columns, occlusion_value)

        # Iterate through single rows of each of the images, comparing their greyscale value intensities
        for i in range(0, num_of_columns):
            for j in range(0, num_of_columns):
                average = (ImageLeft[all_rows][i] + ImageRight[all_rows][j]) / 2
                cost_of_match = 0.5 * ((((average - ImageLeft[all_rows][i]) ** 2) / 16) + (((average - ImageRight[all_rows][j]) ** 2) / 16))
                minimum_one = CostMatrix[i - 1][j - 1] + cost_of_match
                minimum_two = CostMatrix[i - 1][j] + occlusion_value
                minimum_three = CostMatrix[i][j - 1] + occlusion_value
                # Keep the minimum cost in memory
                CostMatrix[i][j] = min(minimum_one, minimum_two, minimum_three)
                c_minimum = min(minimum_one, minimum_two, minimum_three)

                # Mark each of the pixels with some value. One, two, or three in this case
                if minimum_one == c_minimum:
                    DisparityMatrix[i][j] = 1
                elif minimum_two == c_minimum:
                    DisparityMatrix[i][j] = 2
                elif minimum_three == c_minimum:
                    DisparityMatrix[i][j] = 3

        i = num_of_columns - 1
        j = num_of_columns - 1

        # Backward pass, search for the shortest path (from bottom right corner, to top left corner)
        while i != 0 and j != 0:
            if DisparityMatrix[i][j] == 1:
                disparity_left[all_rows][i] = abs(i - j)
                disparity_right[all_rows][j] = abs(j - i)
                i = i - 1
                j = j - 1
            elif DisparityMatrix[i][j] == 2:
                disparity_left[all_rows][i] = 0
                i = i - 1
            elif DisparityMatrix[i][j] == 3:
                disparity_right[all_rows][j] = 0
                j = j - 1

    Draw(disparity_left)
    # Write(disparity_left)

def Draw(disparity_left):
    disparity_left = disparity_left
    depth_left = (disparity_left * 255) / np.amax(disparity_left)
    plt.imshow(depth_left, cmap='gray')
    plt.colorbar()
    plt.show()

def Write(disparity):
    filename = 'result1.png'
    cv2.imwrite(filename, disparity)

def main():
    # First pair of test images
    StereoPair_1_1 = "/Users/albert.ov11/Desktop/OneDrive - University College London/Algorithms/CW2/Stereo Pairs/Pair 1/view1.png"
    StereoPair_1_2 = "/Users/albert.ov11/Desktop/OneDrive - University College London/Algorithms/CW2/Stereo Pairs/Pair 1/view2.png"

    StereoPair_2_1 = "/Users/albert.ov11/Desktop/OneDrive - University College London/Algorithms/CW2/Stereo Pairs/Pair 2/view1.png"
    StereoPair_2_2 = "/Users/albert.ov11/Desktop/OneDrive - University College London/Algorithms/CW2/Stereo Pairs/Pair 2/view2.png"

    # Third pair of test images
    StereoPair_3_1 = "/Users/albert.ov11/Desktop/OneDrive - University College London/Algorithms/CW2/Stereo Pairs/Pair 3/view1.png"
    StereoPair_3_2 = "/Users/albert.ov11/Desktop/OneDrive - University College London/Algorithms/CW2/Stereo Pairs/Pair 3/view2.png"

    # Random Dot pair of test images
    StereoPair_4_1 = "/Users/albert.ov11/Desktop/OneDrive - University College London/Algorithms/CW2/venv/left.png"
    StereoPair_4_2 = "/Users/albert.ov11/Desktop/OneDrive - University College London/Algorithms/CW2/venv/right.png"

    # Read and process both images
    leftIMAGE = cv2.imread(StereoPair_4_1, 0)
    leftIMG = np.asarray(leftIMAGE, dtype=np.uint8)
    rightIMAGE = cv2.imread(StereoPair_4_2, 0)
    rightIMG = np.asarray(rightIMAGE, dtype=np.uint8)

    calculate_STEREO(leftIMG, rightIMG)

main()
```

Appendix 2: Python Code For Random-Dot Image Production

```python
import numpy as np
import cv2
from matplotlib import patches, pyplot, pyplot as plt
import random
import PIL

small_image_size = 256
big_image_size = 512

def CreateBig():
    big_arr = np.random.randint(0, high=255, size=(big_image_size, big_image_size))
    cv2.imwrite("big.png", big_arr)

def CreateSmall():
    small_arr = np.random.randint(0, high=255, size=(small_image_size, small_image_size))
    cv2.imwrite("small.png", small_arr)

def create():
    positionY = 128
    positionX = 124
    small = cv2.imread("small.png", 0)
    bigR = cv2.imread("big.png", 0)
    bigL = cv2.imread("big.png", 0)

    bigL[positionX:positionX + small_image_size, positionY:positionY + small_image_size] = small
    bigR[positionX:positionX + small_image_size, positionY + 8:positionY + small_image_size + 8] = small

    cv2.imwrite("left.png", bigL)
    cv2.imwrite("right.png", bigR)

def main():
    CreateBig()
    CreateSmall()
    create()
main()
```