# Stereo-matching Algorithm Report

Implementing the dynamic programming approach to a maximum likelihood stereo algorithm given in the paper ‘Stereo without Disparity Gradient Smoothing: A Bayesian Sensor Fusion Solution’.

A brief explanation of the algorithm

The algorithm aims to match corresponding pixels between a stereo pair (an image taken from a left and right perspective). For each line it assumes several constraints:

* Matching can only occur along epipolar lines (comparable to the perspectives being captured from the same height), which, for processing purposes, are taken to be coplanar with the image
* Uniqueness: each pixel in one image matches at most one pixel in the other
* Monotonic ordering: if a pixel in the left image at position along the row matches a pixel in the right image at position j in the row, a subsequent measurement in the left image (at position where ) may only match pixels at positions where . This ensures the pixels stays ‘in order’ and don’t end up jumbled

For each row we compute a cost matrix, a table filled with values for every pair of pixels from the stereo images. Each value within the cost matrix depends upon its upper, left and upper-left neighbours and corresponds to either one of the pixels being occluded or the pair being matched. Recording which of the three costs we choose for each cell makes it simple to reconstruct the optimal match by performing a backwards pass through the matrix. The final result is the disparity map which we can output visually as an image.

To implement the algorithm, we have focused on grayscale images meaning that we can simply take the single scalar intensity of each pixel.

Results with random dot stereograms

A set of random-dot stereograms (RDS) were created using the code given in appendix 1, according to the instructions within the coursework description so the squares are offset by 8 pixels. My implementation of algorithm, given by the code in appendix 2, was applied to the RDS pair and produced the disparity map as seen below.

|  |  |  |
| --- | --- | --- |
| A picture containing rain, people, air, group  Description automatically generatedLeft RDS | A picture containing rain, people, air, group  Description automatically generatedRight RDS | Left Disparity Map |

Evidently, the result is very dark, and it is hard to distinguish the different disparities and occlusions. To remedy this, 128 was added the disparities, so only the occluded pixels are left with a value of 0 and the overall photo becomes lighter. To enhance the difference in disparities, the values have also been multiplied them by 10 and interpolated to ensure they were all within the range 0 and 255. These additional calculations produced the outputs shown below for the left and right disparity maps:

|  |
| --- |
| A close up of a logo  Description automatically generated  Left and right disparity maps after multiplying disparities by 10, adding 128 to values and then interpolating within the range [0,255] |

As you can see, this processing made a significant difference and it is now clear to see exactly which pixels are being occluded and discern between the disparities. The results for RDS pairs are discussed in more detail within the Additional Questions, 1.

Further results for images provided

We expect our results with natural imagery to be much less accurate due to the introduction of a range of intensities so greater levels of noise. When applying the same conversions as above to the disparities and using an occlusion cost of 3.8, the outputs for the other stereo pairs are as below:

|  |  |  |
| --- | --- | --- |
| A picture containing snow, flock, group, tall  Description automatically generated | A picture containing outdoor, water, snow, covered  Description automatically generated | A picture containing rain  Description automatically generated |
|  |  |  |

The results produced when varying the occlusion cost is, again, discussed further in the Additional Questions, 2.

Additional questions:

1. Why do matching errors occur for the binary random dot stereograms?

Random dot stereograms have no noise. This is apparent as when our occlusion cost is within the range 0 to 65,205, the result is exactly the same (Note that this is not the case for natural scenes as discussed in the next section). The reason why this occurs is because whilst the occlusion cost is less than 65,205 (equal to the square of the maximum intensity, 255), the matching cost for matching a black pixel to a white pixel is always greater than the occlusion cost, so black pixels are only matched to other blacks and the same goes for white pixels. Therefore, we might expect our stereograms to produce a perfect result, with disparity values of either 0 or 8 constructing a perfect square and a rectangle of occlusion on one side. However, just because a black is never matched with a white, it still doesn’t signify that it is matched to the *correct* black pixel, or indeed occluded correctly. Given that there are only two possible values that each pixel could take, it should be obvious that some pixels in the smaller square will actually be the same value as the pixel it replaces in the larger square. Furthermore, for each pixel there is a 50% chance that the subsequent measurement is the same colour (if you believe that python’s random library is indeed random), so it is highly likely that a cheap match could be found for a pixel as an alternative to occluding it. This is why the edges of our square are misaligned, with some pixels being matched to the background itself and others with a disparity other than 0 or 8.

Developing this idea about how each row gets a different solution leads us to the possibility of multiple global minima. Even though our algorithm finds the global minimum for each row, it does not guarantee that it is the *only* solution for this minimum. For different scanlines, it may choose a different global minimum, which is what leads to the rough occlusion edge, or ‘misalignment of vertical depth discontinuities’ as the paper describes. The paper discusses a way to reduce these by performing a second optimisation to select the set of minima for all rows that contains the least discontinuities, providing a better, though still not perfect, solution.

To investigate the possibility of multiple minima, initially all I did was simply create a variable named pathChoices (initialised as 0 for each pair of images) and before assigning the direction matrix value I added these two lines of code:

if (min1==min2==minimum) or(min2==min3==minimum)or(min1==min3==minimum):  
 possible += 1

The value it outputted represents how many times there were two possible routes in our cost matrix. The result was much higher than I initially would have thought - 67,272,050

(Funnily enough, I originally tried to print a statement instead of adding to a variable - no wonder it crashed!). When computing the same value but for each line rather than the image as a whole, this number varied but was in the 130,000s.

By changing the code around slightly, we can force the algorithm to choose a different path and see the different results produced by different minima. Simply switching the if statements means the possible cases are compared in a different order so different minimums are chosen. Evaluating case 2 - a pixel in the left being occluded (as given in the coursework description) - before case 1 - a match in pixels - means that occluding the left pixel will be favoured over matching them. In essence, this encourages a greater number of shifts over matching to the corresponding pixel which is visibly evident in the results between the first and second images below when looking at the left-hand side of the square. When evaluating case 3 before any of the above, the occlusion of the right pixels is favoured above all, so we see these ‘invading’ inwards of the square rather than outwards. The results of these experiments are shown below; Pay particular attention to the vertical edges of the square (note too that the images have been cropped).

|  |  |  |
| --- | --- | --- |
| A close up of a logo  Description automatically generated  The original ordering: Case 1-2-3 as given in the coursework document | A close up of a logo  Description automatically generated  Output when switching case 1 and 2 so the order is 2-1-3 | A close up of a logo  Description automatically generated  Output when the order of evaluation of cases is 3-2-1 |
| The images to the right simply highlight the pixel differences between each the image directly above them and the original ordering 1-2-3. | A screenshot of a social media post  Description automatically generated | A close up of a logo  Description automatically generated |

To reduce the number of global minima, we can make it less likely for a match to occur by introducing noise. One experiment I carried out is creating my images using overlaid black to white gradients rather than RDS pairs. This actually produces a near perfect match, with the exception of a few pixels where the intensities along the edges of the square are very similar. If you are able to overlay the gradients in some way such that the colours at the edges between the small square and background are dissimilar, you can produce an exactly perfect result for both the left and right disparity maps as shown by the second set of images below!

|  |  |
| --- | --- |
| A picture containing dark, black, screen, sitting  Description automatically generatedA picture containing dark, photo, black, screen  Description automatically generated  Left and right gradient images that produce a near perfect disparity map (a few pixels are incorrectly matched/occluded) | A picture containing table, room  Description automatically generated |

|  |  |  |
| --- | --- | --- |
| A picture containing light  Description automatically generated A close up of a logo  Description automatically generated  An example of gradient images that produce perfect disparity maps | A picture containing bird  Description automatically generated | A picture containing bird  Description automatically generated |

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

As the occlusion cost is varied, the number of matches that are made over occlusions changes. This is shown by the images below (where all disparities have again been multiplied by 10, added to 128 and interpolated).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A close up of a logo  Description automatically generated  Occlusion cost = 0 | A group of tall trees  Description automatically generated Occlusion cost = 1 | A picture containing outdoor, flock, group, snow  Description automatically generated Occlusion cost = 2 | A group of trees  Description automatically generated Occlusion cost = 3 | A group of trees in the background  Description automatically generated Occlusion cost = 4 |
| A flock of birds flying in the sky  Description automatically generated Occlusion cost = 10 | A flock of birds flying in the sky  Description automatically generated Occlusion cost = 15 | A flock of birds flying in the sky  Description automatically generated Occlusion cost = 20 | A flock of birds flying in the sky  Description automatically generated Occlusion cost = 30 | A flock of birds flying in the sky  Description automatically generated Occlusion cost = 35 |
| A flock of white map  Description automatically generated Occlusion cost = 200 | A flock of seagulls flying in the sky  Description automatically generated Occlusion cost = 250 | A flock of white map  Description automatically generated Occlusion cost = 500 | A picture containing boat, white, stop, light  Description automatically generated  Occlusion cost = 750 | A close up of a logo  Description automatically generated Occlusion = 100,000 |

When the occlusion cost is 0, it is always cheaper to occlude a pixel rather than match it, so all pixels are occluded meaning our final image is simply black. On the other hand, when our occlusion cost is sufficiently large, we match every pixel to the corresponding pixel in the other image and never occlude anything, creating a completely white disparity map (would be grey due to adding 128 but since we have interpolated so that the maximum value becomes 255, all pixels are mapped to 255, white).

In between these we can see that the higher the occlusion cost, the more and more noise we tolerate, so the more matches that are made between pixels that have greater intensity differences. Thus, increasing the occlusion cost represents a trade-off between smoothness and incorrect matches. For example, for the results given in the table above, even though the lines are more distinct when the occlusion is 1, there are more dark areas where pixels have been occluded; When the occlusion is 10 the result is much smoother but we see some smears between the distinct peaks of the triangles so it is less accurate for edges.

Depending on the level of noise and the textures in the photos, we may prefer a smaller or larger occlusion cost. Compare the above set of images with the results provided below for the bowling pins. In the bowling pin images, the bottom section is comprised of pixels with very similar intensities, so even with an occlusion cost of 2, it matches these pixels with much more ease than the others (if you look carefully it is all a grey shade, meaning all pixels have been matched but not occluded, this would be completely black). Therefore, for similar images, a high occlusion cost may be unsuitable and a very low occlusion cost is actually desirable.

|  |  |  |  |
| --- | --- | --- | --- |
| A picture containing indoor, sport, table, sitting  Description automatically generated | A picture containing covered, snow, riding, water  Description automatically generatedOcclusion cost = 0.5 | A picture containing covered, water, man, riding  Description automatically generatedOcclusion cost = 1 | A picture containing rain, snow  Description automatically generatedOcclusion cost = 2 |
| A picture containing rain  Description automatically generatedOcclusion cost = 2.5 | A picture containing outdoor, water, flock, bird  Description automatically generated  Occlusion cost = 5 | A picture containing water, window, flock, flying  Description automatically generatedOcclusion cost = 15 |

3. For string matching, what is the equivalent of occlusion?

In string matching, we replace pixel values with values representing characters. If two characters at a specified position do not match, there are several operations we can perform: replacement, insertion and deletion. The edit distance is the number of these operations that we make. The algorithm for this problem is very similar to that of stereo matching. The difference is that each cell in the matrix that we perform the backwards pass on, D[i,j], will now contain the edit distance between length-i prefix of the first string and length-j prefix of the second.

This value is in fact equal to the minimum of its left neighbour (the case in which the ith letter is deleted from the left substring), its upper neighbour (the case where we should insert the jth character from the right substring into the left substring between the ith and i+1th positions), and its upper left neighbour (either a match or replacement operation).

By reviewing what each direction represents, it is clear to see that the equivalent of occlusion is an insertion or deletion. When producing the left disparity match, if a pixel is present in the left image but occluded in the right, this is the same as an insertion, and pixels that aren’t present in the left that are in the right would represent a deletion. The case of a replacement could be compared to the case where two corresponding pixels have slightly different intensities in each image but nonetheless are a match.

Appendix:

1. Python code for creating the two random-dot stereograms

import random  
import numpy as np  
from PIL import Image  
  
# Initialise four arrays: a 512x512 and a 256x256 image of random black and white pixels, and the left/right view  
imageA = np.zeros([512, 512], dtype=np.uint8)  
imageB = np.zeros([256, 256], dtype=np.uint8)  
leftView = np.zeros([512, 512], dtype=np.uint8)  
rightView = np.zeros([512, 512], dtype=np.uint8)  
  
for i in range(512):  
 for j in range(512):  
 imageA[i][j] = random.choice([0, 255])  
  
for i in range(256):  
 for j in range(256):  
 imageB[i][j]= random.choice([0, 255])  
  
# Both the left and right views share Image A as a common background  
leftView = imageA.copy()  
rightView = imageA.copy()  
  
# Modify the left image so that the top-left corner of the image B starts at (124,128)  
for i in range(256):  
 for j in range(256):  
 leftView[i + 128][j + 124] = imageB[i][j]  
  
# Modify the right image so that the top-left corner of the image B starts at (132,128)  
for i in range(256):  
 for j in range(256):  
 rightView[i + 128][j + 132] = imageB[i][j]  
  
Image.fromarray(leftView).save("randomdotleft.png")  
Image.fromarray(rightView).save("randomdotright.png")

1. Python code implementing the stereo-vision algorithm, as described in the paper ‘Stereo without Disparity Gradient Smoothing: A Bayesian Sensor Fusion Solution’.

import numpy as np  
from PIL import Image  
import os  
dirname = os.path.dirname(\_\_file\_\_)  
variance = 16  
  
  
def createDisparityMap(leftImage, rightImage, occlusionCost, leftDisparityMap, rightDisparityMap):  
 *"""  
 This function takes two arrays and computes their left and right disparity maps  
 To do this it creates a cost matrix and direction matrix for each row to enable  
 a backwards pass to be completed  
  
 Args:  
 leftImage (np.ndarray): array representing left image  
 rightImage (np.ndarray):array representing right image  
 occlusionCost (float): cost of occlusion  
 leftDisparityMap (np.ndarray): empty array to be filled with left disparity values  
 rightDisparityMap (np.ndarray): empty array to be filled with right disparity values  
 Returns:  
 leftDisparityMap (np.ndarray): result for the left disparity map for the given images  
 rightDisparityMap (np.ndarray):result for the right disparity map for the given images  
 """* rowCount = leftImage.shape[0]  
 columnCount = leftImage.shape[1]  
  
 for rowNo in range(rowCount):  
 leftRow = leftImage[rowNo]  
 rightRow = rightImage[rowNo]  
  
 costMatrix = np.zeros((columnCount + 1, columnCount + 1))  
 directionMatrix = np.ones((columnCount + 1, columnCount + 1), dtype=np.uint8)  
  
 # Initialise the topmost row and leftmost column of the costMatrix, inclusive  
 for i in range(0, columnCount + 1):  
 costMatrix[i, 0] = i \* occlusionCost  
 costMatrix[0, i] = i \* occlusionCost  
  
 for i in range(1, columnCount + 1):  
 z1 = leftRow[i - 1]  
 for j in range(1, columnCount + 1):  
 z2 = rightRow[j - 1]  
 matchingCost = pow((z1 - z2), 2) / variance  
 # Case 1 - pixels i and j match  
 min1 = costMatrix[i - 1][j - 1] + matchingCost  
 # Case 2 - pixel i is unmatched  
 min2 = costMatrix[i - 1][j] + occlusionCost  
 # Case 3 - pixel j is unmatched  
 min3 = costMatrix[i][j - 1] + occlusionCost  
  
 # The value in the cost matrix is the minimum of the above cases  
 minimum = min(min1, min2, min3)  
 costMatrix[i][j] = minimum  
  
 if minimum == min3:  
 directionMatrix[i][j] = 3  
 elif minimum == min1:  
 directionMatrix[i][j] = 1  
 elif minimum == min2:  
 directionMatrix[i][j] = 2  
 else:  
 print("Error matching minimum")  
 print(i, j)  
 print(costMatrix[i][j], int(min1), int(min2), int(min3), min1, min2, min3)  
  
 # Perform a backwards traversal using the directionMatrix to construct optimal match  
 i = columnCount #i represents the rows in our directionMatrix (left image pixels)  
 j = columnCount #j represents the columns (right image pixels)  
  
 while (i != 0) and (j != 0):  
 if directionMatrix[i][j] == 1:  
 # Disparity here is multiplied by 10 and added to 128  
 # This makes disparity values more distinct and occlusions more obvious  
 disparity = abs(i - j) \* 10 + 128  
 leftDisparityMap[rowNo][i] = disparity  
 rightDisparityMap[rowNo][j] = disparity  
 i -= 1  
 j -= 1  
 elif directionMatrix[i][j] == 2:  
 i -= 1  
 elif directionMatrix[i][j] == 3:  
 j -= 1  
 else:  
 print("Error within direction matrix")  
  
 # Interpolate values to map them between range [0,255]  
 leftDisparityMap \*= 255.0 / leftDisparityMap.max()  
 rightDisparityMap \*= 255.0 / rightDisparityMap.max()  
 leftDisparityMap = leftDisparityMap.astype(np.uint8)  
 rightDisparityMap = rightDisparityMap.astype(np.uint8)  
 return leftDisparityMap, rightDisparityMap  
  
  
def go(occlusionCost):  
 *"""  
 This function does the bulk of the image processing, importing images, converting them to greyscale and storing  
 as arrays. It calls a function to compute their disparity maps for the given occlusion cost before outputting   
 the results  
   
 Args:   
 occlusionCost (float): cost of occlusion  
 Outputs:  
 2 images representing the left and right disparity maps  
  
 """* leftImage = np.array(Image.open(os.path.join(dirname, 'pair2/view1.png')).convert('L'))  
 rightArr = np.array(Image.open(os.path.join(dirname, 'pair2/view2.png')).convert('L'))  
  
 leftDisparityMap = np.zeros(leftImage.shape, dtype='float64')  
 rightDisparityMap = np.zeros(leftImage.shape, dtype='float64')  
  
 leftDisparityMap, rightDisparityMap = createDisparityMap(leftImage, rightArr, occlusionCost, leftDisparityMap, rightDisparityMap)  
  
 Image.fromarray(leftDisparityMap).save(os.path.join(dirname, "out2/Displeft" + str(occlusionCost) + ".png"))  
 Image.fromarray(rightDisparityMap).save(os.path.join(dirname, "out2/Dispright" + str(occlusionCost) + ".png"))  
  
  
for i in np.arange(0.5, 5, 0.25):  
 go(round(i, 1))  
 print(i)