Problem Set 7

CMSC 426

Assigned Tuesday April 27, Due Tuesday, May 11

1. **Stereo Correspondence.** For this problem set you will solve the stereo correspondence problem using dynamic programming, as described in class. The goal of this algorithm is to find the lowest cost matching between the left and right images, so that the matching obeys the epipolar, ordering, non-negative disparity and uniqueness constraints. First, let's define these:
    a. The epipolar constraint tells us that we can match the images one row at a time. So we have to solve a matching problem with 1D images, matching pixels in a row in the left image to pixels in the same row of the right image. Then we combine the results for every row. Note that we will use images in which the epipolar lines are horizontal.
    b. The ordering constraint means that if pixel *i* in the left image matches pixel *j* in the right image, then no pixel to the right of pixel *i* is allowed to match a pixel to the left of pixel *j*.
    c. The uniqueness constraint means that every pixel can match at most one pixel. However, a pixel might be occluded, and match nothing.
    d. Non-negative disparity means that no point should have negative disparity, because all points are in front of the camera, and have positive depth.
    e. Subject to these constraints we use a cost function to measure how good a match is. If we match pixel *i* in image *L* to pixel *j* in image *R*, the cost of this match will be $(L(i)-R(j))^2$. If any pixel is not matched, the cost of this is OC, which is some constant occlusion cost. For the experiments below, I assume that you first normalize all images so that intensities range between 0 and 1. Then an occlusion cost of OC = .01 should work well. This is about the same as the cost of matching two pixels with an intensity that differs by .1, or 25 in the original image. However, feel free to experiment with other values to try to improve the results.

These constraints allow us to find the best matching between two epipolar lines using dynamic programming. One way to see this is to think about constructing a cost table, C. C(i,j) contains the cost of the best possible set of correspondences and occlusions that accounts for the first i pixels in the left image and the first j pixels in the right image. We will build this table recursively, so that C(i,j) is computed using only values of C(i',j') for i'<=i, j'<=j.

We initialize C(0,0) = 0. This means that if we haven't accounted for any pixels, there is zero cost. Next, let's consider C(1,0). This means that a set of matchings account for the first pixel in the left image, and no pixels in the right image. This can only happen if the first pixel in the left image is occluded, so that C(1,0) = OC. Similarly, for any i, C(i,0) = i*OC, and C(0,j) = j*OC.

Next, with this initialization, we can think about how to fill in an arbitrary point in the table, C(i,j). There are three ways we can get to this point in one step. One is that we might have matched pixel i in the left image to pixel j in the right image. In that case, the cost is the cost of matching pixels i and j, plus the cost of the best way of matching all pixels in the left image up to i-1 and all pixels in the right image up to j-1. So, if we say that L(i) is pixel i in the left image, and R(j) is pixel j in the right image, then one possibility is: $C(i,j) = (L(i)-R(j))^2 + C(i-1,j-1)$. But it is also possible that the last step before we account for pixels up to i and j is that we occluded a pixel in the left or right image. So we have:
$C(i,j) = MIN((L(i)-R(j))^2 + C(i-1,j-1), OC + C(i,j-1), OC + C(i-1, j))$.

Using this recursion, we can fill an entire table of costs. If the left image has n pixels and the right image has m pixels, we keep doing this until we have found the value of C(n,m). That gives us the cost of the best possible way of matching the two images.

To find the actual disparities, though, we need to not only compute the lowest cost, but also keep track of how we got there. To do this, we can keep another table, M, which records which move we took to obtain the minimum cost matching. So, M(i,j) will tell us how we accounted for pixels in the last move that brought us to account for pixels up to i in the left image and j in the right. For example, we might use M(i,j) = 1 to indicate that pixel i matched pixel j, while M(i,j) = 2 might indicate that pixel i was occluded. Using M, we can then trace back to find all correspondences and disparities. So, if M(n,m) = 1, that means pixel n in the left image is matched to pixel m in the right image, with a disparity of n-m. It also means that we should look at M(i-1,j-1) to find the next match. But if M(n,m) = 2, this means that pixel n was occluded in the left image, and we should look next at M(n-1,m).

a. **Paper and pencil exercise**: Using the cost function above, compute all minimum cost matches that obey all the constraints listed above for the 1D images, v1 = [1 0 1 1]; v2 = [1 1 0 1]. Write your answers as a disparity map for v1. Since disparity is always non-negative, we can use -1 to indicate an occlusion. That is, a map of [0 -1 1 1] means the disparity for the first point in v1 is 0 (v1(1) = 1 matches v2(1) = 1, the second point is occluded and unmatched, the third point has a disparity of 1, (v1(3) =1 matches v2(2) = 1), and the fourth point has a disparity of 1 (v1(4) =1 matches v2(3) = 0). Of course, this example is not necessarily a minimum cost matching.
    a. **5 points** There may be one or more matches with the same minimum cost. List all of them, along with their cost.
    b. **5 points** List all minimum cost matches if we are allowed to ignore the non-negative disparity constraint.
    c. **5 points** List all minimum cost matches if we ignore the ordering constraint and the non-negative disparity constraint.

b. Using disparity. Suppose that we have two stereo cameras that have a baseline of 10 cm. These cameras have focal points on the *x* axis. Suppose further that each pixel is 1mm in length.

a. **5 points:** Suppose the cameras each have a focal length of 5mm, and are orthogonal to the z direction. If we match two points with a disparity of 10 pixels, what is their depth (that is, their distance in the Z direction).

b. **5 points:** Suppose the left camera has a focal length of 5mm, but the right camera has a focal length of 10mm. We still match two points with a disparity of 10 pixels. Do you have enough information to determine the depth of this point? If yes, what is the depth? If no, give an example of two possible matches with the same disparity and different depths.

c. **30 points** Now, write a function *stereo1D.* This function will take as input two 1D images, along with an occlusion cost, OC. For our experiments, OC will be .01. The function will compute the C matrix described above, containing the cost of best matches. If you call this function with:

Left image: 0 0 255 0 0 255
Right image: 0 255 0 0 255 0

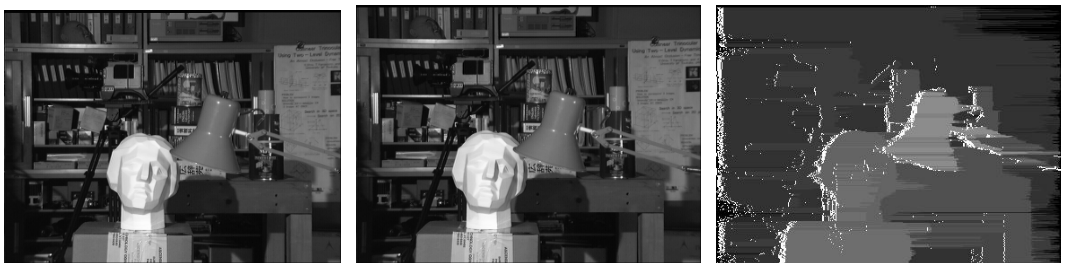
the cost matrix should look like this:

```
cost = 0.0 0.01  0.02 0.03 0.04 0.05
cost = 0.01 0.0  0.01 0.02 0.03 0.04
cost = 0.02 0.01  0.02 0.01 0.02 0.03
cost = 0.03 0.02  0.01 0.02 0.03 0.02
cost = 0.04 0.03  0.02 0.01 0.02 0.03
cost = 0.05 0.04  0.03 0.02 0.01 0.02
cost = 0.06 0.05  0.04 0.03 0.02 0.01
```

(keep in mind that we normalize intensities to be in the range (0,1)). Turn in your code and a print out showing the result.

d. **30 points:** Enhance this function so that it will also keep track of which matches or occlusions occurred to produce a matching, with an M matrix, and returns the disparities of the best match. The disparities for the example in (c) should look like: [0 -1 1 1 1 1], where -1 indicates an occlusion. Turn in your code and a print out showing the result.

e. **20 points:** Now expand your program so that you read in left and right images from a file and compute a 2D disparity map for the entire images. This is done by just running the 1D stereo code on each pair of conjugate epipolar lines (ie, each pair of rows with the same row number) and collecting the results together. Save the resulting disparity maps. Run your code on the Tsukuba images, and turn in the resulting disparity map. These are T3bw.jpg and T4bw.jpg. Below we show the images and resulting map that we obtain.



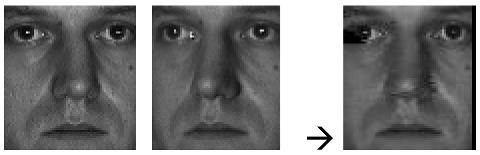
Turn in your code, and the output on these runs. The web page also includes two random dot stereograms, I1L and I1R. Run your code on these and turn in an image showing the resulting disparity map.

f. **20 points** We can also use stereo to interpolate between two images, and find intermediate images. To see this, suppose that point i in the left image has a disparity of 4. This means that it appears in the right image at pixel i – 4. So, if we had a viewpoint halfway in between the two images, this point would appear at pixel i – 2. So, we can generate a new image containing this pixel value.

   There are a couple of complications. First, the intermediate location of this pixel might not be an integer. Second, some of the pixels of the intermediate image might not be filled in by these values. (There are also occluded pixels, but we can just ignore these). This means that we must interpolate the values of all pixels in the intermediate image, using the values of matched pixels.

   For example, suppose the left image is [120 130 140 200 210] and the right image is [122 141 202 208 240]. We compute a disparity map of [0 -1 1 1 1]. We want to generate a new image, taken from halfway between these two images. We can then assign a value of 121 to the first pixel in the new image (I'm using the average of 120 and 122, since these two pixels are matched.). Next, we can determine that halfway between the second and third pixels we should have a value of 140.5. Using linear interpolation, we can calculate that the second pixel should have a value of (1/3)*121 + (2/3)*140.5 = 134.

   Using this approach, write a program to use the disparities you compute to generate an image that is halfway between the left and right images. Test this on the two face images given. Here are the results we get:



2. **Challenge Problem:**
   a. **5 points:** You can speed up your code by limiting the possible disparities. For example, try limiting the maximum disparity to 16 pixels. Explain how you can do this, and report on how much it speeded up your program.
   b. **10 points:** Dynamic programming fills up an entire cost table, C. However, this may not be necessary. If we consider how shortest path algorithms work, we can see that some entries in the table may not need to be filled, because their neighbors already have high costs. Implement a shortest path approach to computing stereo. Explain how you did this. How much does this speed thing up?
   c. **Up to 15 points:** The stereo algorithm we have implemented doesn't really handle slanted surfaces in an ideal way. This problem is explained and resolved in the paper: "Efficient Dense Stereo with Occlusions for New View-Synthesis by Four-State Dynamic Programming," by A. Criminis, J. Shotton, A. Blake, C. Rother, P.H.S. Torr. Look up this paper. For partial credit, explain the problem we have with slanted surfaces and how this paper resolves it. For full credit, implement their approach and compare to the approach in this problem set.
   d. We're open to other small projects for extra credit. Feel free to email me if there's something else you'd like to try.