1. Unsharp masking

Unsharp masking produces an edge image \( g(x, y) \) from an input image \( f(x, y) \) via

\[
g(x, y) = f(x, y) - f_{\text{smooth}}(x, y)
\]

where \( f_{\text{smooth}}(x, y) \) is a smoothed version of \( f(x, y) \).

We can better understand the operation of the unsharp sharpening filter by examining its frequency response characteristics. If we have a signal as shown in Figure 1(a), subtracting away the low pass component of that signal (as in Figure 1(b)), yields the highpass, or ‘edge’, representation shown in Figure 1(c).

![Figure 1](image)

**Figure 1** Calculating an edge image for unsharp filtering.

This edge image can be used for sharpening if we add it back into the original signal, as shown in Figure 2.

![Figure 2](image)

**Figure 2** Sharpening the original signal using the edge image.
We can now combine all of this into the equation:

\[ f_{\text{sharp}}(x, y) = f(x, y) + k \times g(x, y) \]

where \( k \) is a scaling constant. Reasonable values for \( k \) vary between 0.2 and 0.7, with the larger values providing increasing amounts of sharpening.

Consider an image of your choice.

a) Perform unsharp sharpening on the raw image using a mean filter and a Gaussian filter (with the same kernel size). How do the sharpened images produced by the two different smoothing functions compare?

b) Try re-sharpening this image using a filter with larger kernel sizes (e.g. 5×5, 7×7 and 9×9). How does increasing the kernel size affect the result? c) What would you expect to see if the kernel size were allowed to approach the image size?

2. Stereo Correspondence.

For this problem set you will solve the stereo correspondence problem using dynamic programming. 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, (25 in the original image, with values from 0 to 255). 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 in a recursive manner (not necessarily with recursive functions), in which 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-1,j), OC + C(i,j-1)).

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-1. 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) 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. We can use X to indicate an occlusion. That is, a map of [0 X 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.

1. There may be one or more matches with the same minimum cost. List all of them, along with their cost.
2. List all minimum cost matches if we are allowed to ignore the non-negative disparity constraint.
3. List all minimum cost matches if we ignore the ordering constraint and the non-negative disparity constraint.

(b) 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 .06
cost = 0.01 0.0 0.01 0.02 0.03 0.04 .05
cost = 0.02 0.01 0.02 0.01 0.02 0.03 .04
cost = 0.03 0.02 0.01 0.02 0.03 0.02 .03
cost = 0.04 0.03 0.02 0.01 0.02 0.03 .02
cost = 0.05 0.04 0.03 0.02 0.01 0.02 .03
cost = 0.06 0.05 0.04 0.03 0.02 0.01 .02
```
(keep in mind that we normalize intensities to be in the range (0,1)).

(c) Now expand your program so that you apply it to left and right images 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 (i.e., each pair of rows with the same row number) and collecting the results together. Save the resulting disparity maps. Run your code on a stereo pair from the Middlebury Stereo database (use the Tsukuba sequence).

3. Image Mosaic

The goal of this problem set is to write code to form a mosaic of two images. We begin with two images (next page) that have overlapping fields of view and stitch them together into a single, larger image. You will match manually features in one image to features in the second image. You will then compute the affine transformation A that relates these matches, and then apply this transformation to combine the images.

Write a function, stitch(J, K, A), that will stitch two images, J and K, together, using the affine transformation A. For this problem, just do this in a very simple way. For example, for every pixel in image 2, apply A to find its transformed location. Round off this location to an integer value. Place this value in the new image at this location. At the same time, place all values from image 1 into the new image at their original location. If a pixel doesn’t get a value from image 1 or image 2, make it black. If a pixel gets two or more values, you can combine them in any way you want. That is, if image 1 and image 2 overlap, you might use the average, or always use image 1’s value, or always use image 2’s value. Doing this may produce some artifacts, since there may be pixels in the middle of the new, target image, that don’t get any value assigned from either of the original images. You can fix this for extra credit.