1. **Stereo Correspondence. (100 points)**

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 ifwe 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 pixelsin 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) 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 (ie, 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).

2. Mosaics

The goal of this problem set is to write code to form a mosaic of two images. We begin with two images that have overlapping fields of view (see below) and stitch them together into a single, larger image. Our strategy is to use SIFT to locate feature points and their descriptors. Each feature in one image is matched to a feature in the second image with the most similar descriptor. Note that many, but not all of these matches will be correct. To find a good set of matches we use RANSAC. We randomly pick three matches, compute the affine transformation that relates these matches, and then apply these to all the feature points. We repeat many times and pick the transformation that maps the most feature points in one image near their matching feature point in the other image. Finally, we use this transformation to combine the images.

We will give you code that computes SIFT features and descriptors for a single image. (Actually, we give you instructions on retrieving this code). This code includes a Matlab wrapper you can use to call SIFT. We also give you two test images, LR1 and LR2 (above), to use in testing your code.

DOWNLOAD SIFT CODE:

The code for finding SIFT features is due to Andrea Vedaldi. Go to www.vlfeat.org and download the latest version of vlfeat code (0.9.16). Download the binary package. Then follow the instructions under Matlab install on the menu on the left. Note that there is a lot of documentation for everything, including all functions in VLfeat. You should just need to execute the command: `run('VLFEATROOT/toolbox/vl_setup')`

Problems:

1. 10 points: Run the code to detect SIFT features in a white square on a black background, and in the two images above. The results should look like this:

Include the resulting images in your write-up.

2. 10 Points: Find Best Match. vl_sift returns two values, F and D. Each column of F contains the x and y coordinates, orientation and scale of each feature. Each corresponding column of D contains a descriptor for that feature. Write code that takes every SIFT feature in image 2 and finds the feature in image 1 with the most similar descriptor. You should compare two SIFT descriptors (which are histograms) using SSD. That is, just compute the SSD of the appropriate columns of the D that is computed for each image. If there are *n* features in image 2, the output of this might be an *n*x5 array, in which each row contains the (x,y) coordinates of a point in image 2, the point in image 1 that matches it best, and the score of this match.

Now, using this code, write a function, find_best_match, which finds the three matches that have the lowest SSD. Display both images with these points shown in three different colors. That is, the images might each have a white disk, indicating the location of the two points that match best, a green disk, showing the points that match second best, and a blue disk, showing the points that match 3rd best. Run this on LR1 and LR2. Include the resulting images in your write-up. The results should look like this (it's a little hard to see the blue dot without enlarging the image):

3. 10 points: Affine Transform. Write a function that takes three matching points as input, and computes an affine transformation that maps three points from one image to the matching three points in the second image. So executing:

```
A = affine_transformation(p1,p2);
```

computes a 2x3 affine transformation, A, that maps the points in p2, represented as a 2x3 matrix in which each column is a point, so that they match the points in p1.

4. 30 Points: RANSAC. Now, perform RANSAC to find the best affine transformation that maps the most points from image 2 to image 1. To do this, perform the following steps:

a. Randomly pick three matching points, in the format computed in part 2.
b. Compute the affine transformation, A, that relates them, using part 3.
c. Apply A to all points in image 2.
d. For each point, p, in image 2, that has been matched to point q in image 1, find out whether Ap is close to q ("close" might mean their Euclidean distance is less than 2).
e. Count the number of points that are mapped by A to be close to their matching point.
f. Repeat steps a-e many times (maybe a thousand?) and choose the A with the highest total.

5. 20 Points: Stitching. Now 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 as part of the challenge problem. Test your code on LR1 and LR2 using two affine transformations. First, create an affine transformation that will translate LR2 100

pixels in the x direction, to the right of LR1. Next, create an affine transformation that will rotate LR2 by pi/16, scale it by .8, and translate it 100 in the x direction and 50 in the y direction. The results of my program are shown below:

Hint: Suppose I and J are the first and second image, and A is the affine transformation. For a point, (x,y), let (x',y') = round(A(x,y)). Then we could just set I(x',y') = J(x,y). One of the things that might make this tricky is that x' or y' might be less than 1. However, the examples in this project are constructed so that you won't have to worry about that. This is something you might want to handle as part of the challenge problem.

Note: Combining two images can be done much more efficiently using the Matlab function imtransform. I'm not recommending that, because I think it will be trickier, though. However, you are free to use imtransform if you want.

6. 10 points Now write a mosaic program that puts this all together. It runs SIFT on each image, finds the best matching points, runs RANSAC to compute an affine transformation, and uses this transformation to stitch the images together. The results of my program are shown below.

7. 10 points: Now, take your own photos and run your program on them. Notice that the affine transformation you computed in part 4 is almost a pure translation. This is because I moved the camera very little between pictures. If you move the camera too much, even an affine transformation won't work well. See if you can take two pictures that are well aligned by a more complicated affine transformation. Show the pictures you started with, the result, and the affine transformation that was computed.

8. Up to 60 Points: Challenge problem. Enhance this in a significant way. Suggestions:
a. Improve your program so that you can provide it with many images, and it will stitch them all together.
b. Improve the program so that you use bilinear interpolation to fill up all the pixels in the image. Show an example where this improves the results. For example, if the second image is smaller than the first image, this will be necessary.
c. Often, due to automatic gain control or other factors, the overall lightness of the images will be different. This looks bad when they are stitched together. Even in the result in Part 5 this is a factor. You can see a slight line where the images overlap, because the left image is a little lighter. Find a way to normalize the images to remove this effect.
d. If you know about projective transformations, enhance the program so that it uses a full homography to align the images.