Problem Set 3
CMSC 733
Assigned October 14, 2014
Due October 28, 2014

E-M

The purpose of this problem is to implement the Expectation-Maximization algorithm for the problem of grouping points into lines. That is, we assume that we are given a set of points, and we want to find two lines that explain them. In the expectation step, we find the line that minimizes the weighted sum of squares distance from points to lines. The variance is estimated using the distance between each line and the points. In the maximization step we assign (probabilistically) each point to each line based on its distance to the lines. You can base your implementation on Yair Weiss' notes:

http://www.cs.huji.ac.il/~yweiss/emTutorial.pdf

1. **Line Fitting (10 points)** Write a function that fits a line to data. Note that Weiss describes a method for doing this using weighted least squares, which essentially only looks at error in the y direction. This fits the examples below, in which noise is added to the y coordinate. If you are interested, you can also implement, for a small amount of extra credit, a total least squars method, that takes account of the Euclidean distance between each point and the line, and see what difference this makes. Test your function with the following set of points (I'm using Matlab notation, in which 'a:b:c' means to select numbers from a to c, in increments of b, and randn generates a random variable with Gaussian distribution, zero mean and variance of 1):

   (i) x=0:0.05:1; y=2*x+1
   (ii) x=0:0.05:1; y=2*x+1+0.1*randn(size(x)).
   (iii) x=0:0.05:1; y=(abs(x-0.5) < 0.25).*(x+1)+(abs(x-0.5) >=0.25).*(-x);
   In all cases plot the data and the best fitting lines.

2. **E-M (40 points)** Write a function that estimates the parameters of two lines using EM. It should get as input vectors x,y and return (a1,b1,c1), (a2,b2,c2) the parameters of the two lines as well as the weight vectors w_1 and w_2. (Set the free parameter sigma^2=0.1.)
   a. Test your function on the data in part (iii) of the previous question. Plot the data and the two fitted lines as estimated after each of the first five iterations. Also, show in separate plots the membership vectors after every iteration.
   b. Experiment with adding Gaussian noise to the y coordinates. How much noise can you add before the algorithm breaks.

Mosaics

The goal of this problem set is to write code to form a mosaic of 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 (0.9.19). 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')
```

If you seem to need a compiler for the matlab mex file, it can be found at:
http://www.mathworks.com/support/compilers/R2014b/index.html

We include some skeleton code and a helper function, in the file ps3.m.

Problems:

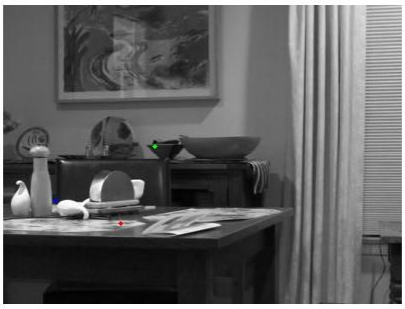
1. **5 points:** Run the skeleton code to detect SIFT features in a white square on a black background, and in the two images above. The results should look like this:

   Essentially all the code you need to do this is given to you. The point of this problem is just to get you started running SIFT. Include the resulting images in your write-up.

2. **5 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. **5 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. Test this by running the code in ps6. You might get results like:

```
>> ps3(3)

ans =

  1.0e-15 *

   0.3331    0.2220    0.1110
  -0.0555   -0.2220   -0.4441
```

4. **5 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.

Test your code using part 4 of ps3. In this code, I construct a test case with random points in p1 and p2, some of which are related by an affine transformation. For example, you might get:

A =

```
  70.1550   77.2314   14.4582
  36.9972    9.5638    9.4105
```

Ares =

```
  70.1550   77.2314   14.4582
  36.9972 9.5638    9.4105
```

5. **10 Points: Stitching.** Now write a function, `stitch(J, K, A)`, that will stitch two images, J and K, together, using the affine transformation A. You must apply the affine transformation A to K, and then combine it with J in a single image. One way to think about this is to first create a new image, L, which you know will be big enough to hold J and K together. Then, for every pixel in L, you can ask whether this pixel lies inside J and/or whether, applying the inverse of A to this pixel, it would lie in side K. If the answer to either question is yes, you can retrieve the appropriate pixel value from the relevant image. If the value you need comes from K, it will generally lie in between several pixels, so you could use bilinear interpolation to retrieve the appropriate value. However, you might find it easier to use the Matlab function, imtransform.

   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.

6. **5 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. **5 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. **10 points:** 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 6 this is a factor. You can see a slight line where the images overlap, because the left image is a little lighter. Do some research to find a way to normalize the images to remove this effect. Take images that will demonstrate your results, and explain how you achieved them.