Assignment 3: Creating Panoramas

This assignment is out of 100 points. But there are additional bonus points in various parts of the assignment.
Please turn into Moodle, a .zip file containing the following:

1. A short report containing answers to all necessary questions and optional questions of your choice. The report should contain **all images** generated by applying your code to images listed in the questions. Additionally, each question in the report should contain names of functions / scripts that you have written for that particular question, and input and output at the MATLAB prompt (as long as the output is small: if the output is an image, do not print it to the MATLAB prompt!). If there is any hand-written part that you will provide me in person, please indicate this in the report.

2. All images generated by applying your code, written out as .jpg or .png files.

3. All scripts and functions written by you for the assignment. Important: make sure the code is commented with a help block in the beginning, and with comments throughout the code.

4. For Section 7, you can include photos of your pen-and-paper work or you can write it up electronically as well in which case turn in a PDF/Word/anything-you-like-that-can-be-reliably-opened-on-a-Mac. **Please do not turn in physical copies to me, I will not be able to keep track of them.**

Start your assignment soon! Assignment questions begin on Page 2. Happy coding!

You will be making panoramas on this assignment! As we saw in class, you can make a panorama out of two images by warping one image to match the other image, and blending the two images. Let us go through the steps of making the panorama. You will need the `transformImage` function from Assignment 1 to warp one image.

For the assignment, you will use

1. The pair of images 'Image1.jpg' and 'Image2.jpg'.
2. **Two** pairs of images each of a different scene of your own choosing (one scene per pair).

Put all code pertaining to the following sections into a single script. However, make sure you try each step of the script separately at the MATLAB prompt to see and store intermediate results.

1 TAKING PHOTOGRAPHS

[2 points] In class we saw how to use a homography to relate two images for a panorama. The homography relates the two images **only if** you take the images by **standing still and rotating your camera about its axis**, and not if you move (or translate) the camera. Take two photographs of your scene of interest by rotating the camera about its axis. Make sure your scene of interest has tons of features like windows, bricks, trees, cars, etc. so that you can mark corresponding points across the two images. Also make sure that both images **contain overlap**.
For this assignment, you will create a horizontal panorama, where you will take two images 'im1' and 'im2', with 'im1' representing the **left** image and 'im2' representing the **right** image. The following figure shows the two images 'Image1.jpg' and
'Image2.jpg'.


2 OBTAINING CORRESPONDENCES

[3 points] Once you have your photographs, it is time to obtain correspondences. Load both your images into MATLAB. For this assignment, you can convert them to grayscale. Also make

sure you convert them to double using `im2double`.

In class I mentioned several times that you will use SIFT[1] which stands for Scale Invariant Feature Transform to automatically estimate correspondences. You will actually use an approach called SURF (). The authors of SURF[2] have performed certain transformations to SIFT that provide correspondences faster. And the nice thing is that SURF is implemented in MATLAB (R2015 and later) for you to use :-).
Assuming that your images are labeled 'im1' and 'im2' in MATLAB, use the function `detectSURFFeatures` function to obtain correspondences. Do a 'help' on the function to know more about it. You call the function as follows:

```
points1 = detectSURFFeatures( im1 );
```

'points1' is an object that contains information about the features, including their location, and additional information about the points such as the scale and orientations of the points. You repeat the process to get 'points2'. To match 'points1' and 'points2', we need feature descriptors that summarize intensity information round the each point in 'points1' and 'points2'. You can extract the feature descriptors as:

```
features1 = extractFeatures( im1,points1 );
```

Repeat the process for 'points2' and 'im2' to get 'features2'. Now you match the features by calling the function 'matchFeatures'

```
indexPairs = matchFeatures( features1, features2, 'Unique', true );
```

'indexPairs' contains indices in 'points1' and 'points2' that correspond to each other. The unique flag ensures that one point in 'im1' is matched to one and only one point in 'im2'. Extract objects for the matched points as

```
matchedPoints1 = points1( indexPairs( :,1 ) );
matchedPoints2 = points2( indexPairs( :,2 ) );
```

and convert the points objects to coordinates as:

```
im1_points = matchedPoints1.Location ;
```

Repeat the above line for the 'matchedPoints2' to get 'im2_points'. If SURF detects N points, 'im1_points' and 'im2_points' are each of size $N \times 2$.

[1] David G. Lowe, "Distinctive image features from scale-invariant keypoints," International Journal of Computer Vision, 60, 2 (2004), pp. 91-110, `http://www.cs.ubc.ca/~lowe/keypoints/`

[2] Herbert Bay, Andreas Ess, Tinne Tuytelaars, Luc Van Gool "SURF: Speeded Up Robust Features", Computer Vision and Image Understanding (CVIU), Vol. 110, No. 3, pp. 346?359, 2008, http://www.vision.ee.ethz.ch/ surf/papers.html

3 ESTIMATING THE HOMOGRAPHY

[60 points] Write a function, `estimateTransform` to determine the transform between 'im1_points' to 'im2_points', i.e., to determine the transform between 'im1' to 'im2'. Your function should have the form:

```
A=estimateTransform( im1_points, im2_points );
```

In class, we saw how to estimate a 3×3 homography

$$\mathbf{A} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \tag{3.1}$$

by setting

$$\mathbf{q} = \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \end{bmatrix} \tag{3.2}$$

and creating a design matrix **P** and a vector **r**. We also saw how to estimate **q** by using homogeneous least squares, with singular value decomposition (SVD). The function `estimateTransform` should create **P** and **r** according to the direct linear transform (DLT) approach we saw in class. Note: you will get full credit for using a for loop to create **P**. However, there are ways to create both **P** without using a for loop. While **r** is relatively easy to create without a for loop, you will get 1 bonus point for figuring out how to create **P** without using a for loop! Once you create **P**, use the SVD-based method we discussed in class to obtain **q** from **P**. Then rearrange the values of **q** to get the values in **A**. For the following set of points in 'im1=Image1' and'im2=Image2':

$$\text{im1\_points} = \begin{bmatrix} 1373 & 1204 \\ 1841 & 1102 \\ 1733 & 1213 \\ 2099 & 1297 \end{bmatrix} \text{ and im2\_points} = \begin{bmatrix} 182 & 1160 \\ 728 & 1055 \\ 617 & 1172 \\ 1001 & 1247 \end{bmatrix}, \tag{3.3}$$

A should be:

$$\begin{bmatrix} -0.0004272, & -0.0002588, & 0.8379231 \\ -0.0000576, & -0.0007065, & 0.5457875 \\ -0.0000000, & -0.0000003, & 0.0001091 \end{bmatrix} \text{ or } \begin{bmatrix} 0.0004272, & 0.0002588, & -0.8379231 \\ 0.0000576, & 0.0007065, & -0.5457875 \\ 0.0000000, & 0.0000003, & -0.0001091 \end{bmatrix}, \tag{3.4}$$

i.e., where the second solution is the negative of the first solution (like scale, the negation of the homography matrix does not impact the result). Use the points and **A** listed above only to verify your function `estimateTransform`. **However, you must report results for your own set of points**, not the ones listed here. Unless you use points obtained using SURF with RANSAC as discussed below, you will not get full credit.

Note: In MATLAB, you will get a stubby matrix of size 8×9, in which case you should use the regular version of the SVD function. If your matrix were 9×9 or taller, then you would use the flag 'econ' with the SVD function.

In class, we looked at using RANSAC to sample random sets of 4 correspondences and get a near-optimal solution that throws outliers. Write a function `estimateTransformRANSAC`, that uses RANSAC to estimate the transformation between the points in 'im1_points' and 'im2_points'. In each RANSAC iteration, you will call the function `estimateTransform` using the minimum number of correspondences k. What should k be for the homogeneous least squares version?

An important consideration will be how you choose the error threshold t for RANSAC, and the number of RANSAC iterations N_{ransac}. In your write-up include how you chose the error threshold and the number of RANSAC iterations.

Once you perform RANSAC, you will get a list of points N_{agree} which is a subset of all N points. To balance the solution, you should perform a final homogeneous least squares solve on all N_{agree} points, by calling `estimateTransform` on the set N_{agree} points. Your assignment must demonstrate that you have performed this final solve to receive full points.

Note: While we looked at using regular least squares to estimate **q** using the pseudo-inverse of **P**, that method is numerically ill-conditioned. **If you implement the pseudo-inverse method, you will lose all 75 points on this section!** You must use SVD to get full points.

Apply your function `esimateTransformRANSAC` to 'im1_points' and 'im2_points' in your workspace to get the transform matrix, **A**. You will find that when you use RANSAC, **A** may be different from the one above. This is fine, as the results from RANSAC + homogeneous least squares depend upon the random samples chosen.

4 APPLYING THE HOMOGRAPHY

[4 points] Use the function `transformImage` written in Assignment 1 to transform 'im2' to match 'im1'. Call the transformed image 'im2_transformed'.

Be careful! **A** relates 'im1' to 'im2'. To transform 'im2' to 'im1', you have to apply the **inverse** of **A** to 'im2'!

For the panorama, you will find it easier to force the corners in the transformed image to be at (1,1) instead of (minx,miny).

Also, `interp2` may provide NaN (not a number) values. You can reset NaNs to zeros by calling

```
nanlocations = isnan( im2_transformed );
im2_transformed( nanlocations )=0;
```

For the example images, 'im2_transformed' will be similar to the following image:

It is quite likely that your 'im2_transformed' will not appear **exactly** like the one shown here, and will have slight differences. Again, this is fine, as your 'im2_transformed' depends on the estimate of **A** calculated using your correspondences. It should not look too different from the image shown here.

5 EXPANDING THE IMAGES TO BE THE SAME SIZE

[3 points] You will likely have to expand 'im1' to be the same size as 'im2'. Add a matrix of zeros to 'im1' to make it the same size as 'im2_transformed'. Call the expanded matrix 'im1_expanded', an example is shown here: Assume that the resulting images be of size $h \times w$, where h is height and w is the width of 'im2_transformed'.

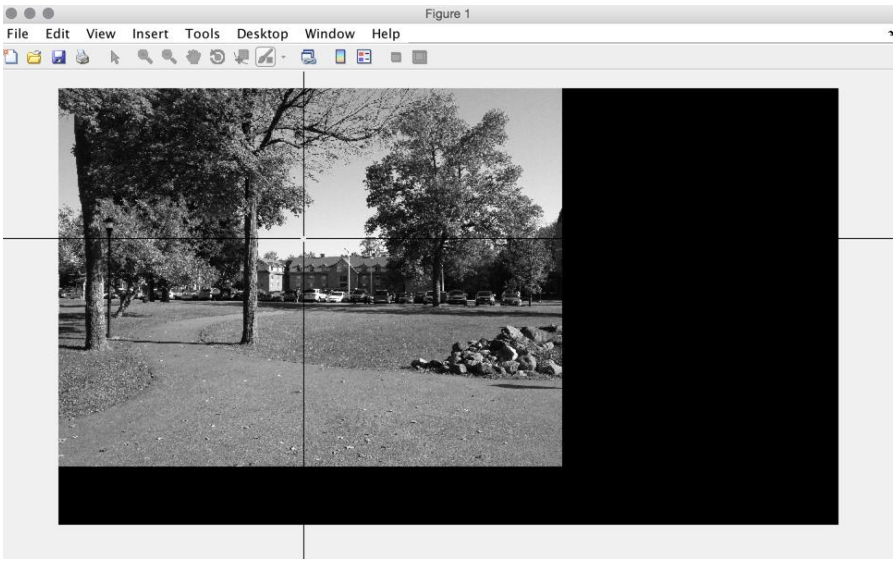
6 BLENDING THE IMAGES

[8 points] Blend the two images to hide seams between the images.

The easiest way to blend the images is to use a **ramp**. In our case, the ramp is a vector of size $1 \times w$, which contains several 0 values, followed by a linearly increasing set of values from 0 to 1, followed by several 1s. To create the ramp, manually locate two points in im1 on either side of the overlapping region. Use the 'ginput' function in MATLAB:

```
imshow(im1_expanded);
[x_overlap,y_overlap]=ginput(2);
```

The 'ginput' function gives a figure window with crosshairs, as shown below:



Since you called 'ginput' with the value 2, it expects you to click twice, so that it can give you two points. You can use this to click once on the left side of the overlap, and once on the right side of the overlap.

```
overlapleft=round(x_overlap(1));
overlapright=round(x_overlap(2));
```

The values 'overlap_left' and 'overlap_right' give you the left and right side of the ramp (the

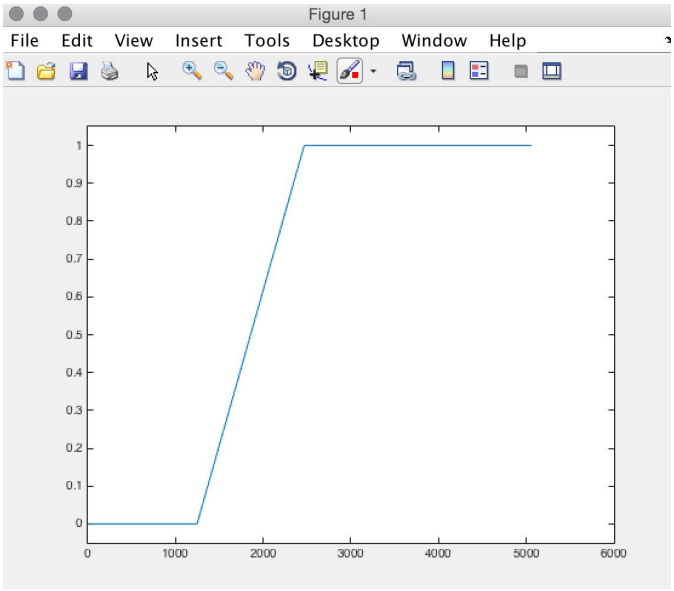
'round' function rounds to the nearest integer). Set your ramp as:

ramp=[zeros_till_overlapleft, overlapleft : stepvalue : overlapright, ones_till_overlapright];

The value of 'stepvalue' represents the step factor to go from 0 to 1 over the interval [overlapleft,overlapright]. For the images chosen, plotting the ramp using

```
plot(ramp);
```

should give you a ramp that has the following graph:



To apply the ramp for blending, you need to use it so that
(1) 'im1_expanded' is converted into an image 'im1_blend' which has values on the left that blend off to zero on the right, and
(2) 'im2_transformed' is converted into an image 'im2_blend' which has zeros on the left that blend into values on the right.

The following two images show examples of 'im1_blend' and 'im2_blend':

Apply the 1×2 ramp to the $h \times w$ images 'im1_expanded' and 'im2_transformed' to obtain

'im1_blend' and 'im2_blend'. As an example, 'im2_blend' can be obtained from 'im2_transformed' by repeating 'ramp' downwards and performing elementwise multiplication to 'im2_transformed':

```
im2_blend = im2_transformed .* repmat( ramp,h,1 );
```

How can 'im1_blend' be obtained?

Finally, the panorama can be obtained by adding 'im1_blend' and 'im2_blend':

```
impanorama=im1_blend+im2_blend;
```

Store all intermediate images: 'im2_transformed', 'im1_expanded', 'im1_blend', and 'im2_blend', as well as the final panorama 'impanorama' to png files, and submit all png files in the zip file associated with the assignment.

To receive 5 additional bonus points, provide a panorama with 3 or more images.

7 Some More Algebra

[20 points] In class, we saw how to create a design matrix for a homography. Now, let us create a design matrix for another transform, specifically, a *similarity transform*. A similarity transform combines rotation, scaling, and translation, and has the following matrix:

$$\mathbf{A} = \begin{bmatrix} a & b & t_x \\ -b & a & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{7.1}$$

Using the technique we saw in class, write out the design matrix **P** for this matrix. How many elements does **q** have, i.e., how many columns does **P** have? If we wanted to solve for a similarity transform, at least how many correspondences would we need?