CS4495 Fall 2014 — Computer Vision
Problem Set 4: Harris, SIFT, RANSAC

DUE: **Wednesday, October 29 - 11:55pm**

The focus of this problem set is on feature computation and model fitting. As defined in class features must be (1) fairly repeatable - will tend to show up in both images even with changes in lighting or imaging; (2) well localizable - their location in the imagery should be easily and relatively precisely determined; (3) fairly common without being dense in the imagery; (4) characterizable such that it is possible to find likely matches. Once we have such features and their *putative* matches, we use RANSAC as a form of global checking to find a likely alignment. You will do each of these steps below, though for some of them code will be provided.

There are several extra credit elements given. Enjoy as you wish.

As a reminder as to what you hand in: A Zip file that has

1. Images (either as JPG or PNG or some other easy to recognize format) clearly labeled using the convention PS<number>–<question number>–<question sub>–counter.jpg
2. Code you used for each question. It should be clear how to run your code to generate the results. Code should be in different folders for each main part with names like PS1-1-code. For some parts – especially if using Matlab – the entire code might be just a few lines.
3. Finally a PDF file that shows all the results you need for the problem set. This will include the images appropriately labeled so it is clear which section they are for and the small number of written responses necessary to answer some of the questions. Also, for each main section, if it is not obvious how to run your code please provide brief but clear instructions. **If there is no Zip file you will lose at least 50%!**

This problem set uses files stored in the directory `http://www.cc.gatech.edu/~afb/classes/CS4495-Fall2014/problemsets/ps4` The images transA.jpg and transB.jpg are a apir of images that are related by a pure translation. Likewise simA.jpg and simB.jpg are images related by a similarity transformation (more on this later). You will use both of these sets for evaluation and testing of your code. Additionally we provide you with two simple images of a checkerboard check.bmp and check_rot.bmp which may be easier to use in testing your corner detection code.

1 Harris corners

In class and in the text we have developed the *Harris* operator. To find the Harris points you need to compute the gradients in both the X and Y directions. These will probably have to be lightly filtered using a Gaussian to be well behaved. You can do this either the "naive" way - filter the image and then do simple difference between left and right (X gradient) or up and down (Y gradient) - or you can take an analytic derivative of a Gaussian in X or Y and use that filtter. The

scale of the filtering is up to you. You may play with the size of the Gaussian as it will interact with the window size of the corner detection.

1.1 Write functions to compute both the X and Y gradients. Try your code on both `transA` and `simA`. To display the output, adjoin the two gradient images(X and Y) to make a new, twice as wide, single image (the "gradient-pair"). Since gradients have negaitve and positiive values, you'll need to produce and image that is gray for 0.0 and balck is negative and white is positive.

Output: The code, and the gradient-pair image for both `transA` and `simA`.

Now you can compute the Harris value for the image. As a reminder the Harris value was define as:

$$R = det(M) - \alpha \,\mathrm{trace}(M)^2$$

where

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

The only design decisions are the size of the window (the sum), the windowing function that controls the weights, and the value of α in the Harris scoring function. Remember you can check you code on the checkerboard images, though those might use a different optimal window size than the real ones.

1.2 Write code to compute the Harris value. You can try the weights just equal to 1. But it might work better with a smoother Gaussian that is higher at the middle and falls off gradually. Your output is a scalar function. Apply to `transA`, `transB`, `simA`, and `simB`.

Output: The code and the Harris value output image for each of the images. To display the output reasonably you will have to scale the image values to be in a range of 0-255 or 0.0 to 1.0, depending upon how you deal with images.

Finally you can find some corner points. To do this requires two steps: thresholding and non-maximal suppression. You'll need to choose a threshold value that eliminates points that don't seem to be plausible corners. And for the non-maximal suppression, you'll need to choose a radius (could be size of a side of a square window instead of a circle radius) over which a pixel has to be a maximum.

1.3 Write a function to threshold and do non-maximal suppression on the Harris output. Surprise, huh? Adjust the threshold and radius until you get a "nice" set of points, probably on the order of a hundred or two (or three?). But use your judgment in terms of getting enough points. Are there any points that are visible in both images but **not** found as corners in both?

Output: The code. Apply your function to both image pairs: (`transA`, `transB`) and (`simA`, `simB`). Mark the corners visibly in each of the four result images and provide those images. Also, describe the behavior of your corner detector including anything surprising, such as points not found in both images of a pair.

2 SIFT features

Now that you have keypoints for both image pairs, we can compute descriptors. You will be glad to know that we do not expect you to write your own SIFT descriptor code. Instead you'll use either a MATLAB package called VLFeat, the Python interface to VLFeat or the SIFT or SURF classes in OpenCV. There will be a supplemental document posted on the website with details. It is posted now (when this PS is distributed) and it may evolve as we figure out more about the Python VLFeat inteface which is new for us this year.

The standard use of a SIFT library consists of you just providing an image and the library does its thing: finds interest points at various scales and computes descriptors at each point. We're going to use the library code **only to compute the orientation histogram descriptors** for the interest points you have already detected from Problem 1. To do so, you need to provide a scale setting and an orientation for each feature point as well as the gradient magnitude and angle for each pixel. The scale we'll fix to 1.0 (see accompanying SIFT software usage tutorial). The orientation needs to be computed from the gradients:

$$\text{angle} = \text{atan2}(I_y, I_x)$$

But, you already have the gradient images! So you can create an "angle" image and then for a given feature point at $< u_i, v_i >$ you can get the gradient direction.

2.1 Write the function to compute the angle. Then for the set of interest points you found above, plot the points for all of `transA, transB, simA` and `simB` on the respective images and draw a little line that shows the direction of the gradient. In MATLAB or Python (or even C) if you want you can use the VLFeat function `vl_plotframe` to draw the feature points locations and the angle. You'll need to figure it out - look at `http://www.vlfeat.org/overview/sift.html` and also the documentation for `vl_plotframe`. In OpenCV you can use the method `drawKeypoints()`.

Output: The code. And both of the drawn on pairs (`transA`, `transB`) and (`simA`, `simB`).

Now we're going to call the SIFT descriptor code. You need to pass in each keypoint's location along with its scale and orientation. This process is covered in more detail in the accompanying supplemental document.

Once we have the descriptors, we need to match them. This is what we called *putative* matches in class. Given keypoints in two images, get the best matches. Both VLFeat and OpenCV have functions for computing matches (e.g. for VLFeat it's called `vl_ubcmatch`. But I'm not sure there is a python interface to that function. We'll update that in the supplemental documentation shortly.) You will call those and then you will make an image that has both the A and B version adjoined and that draws lines from each keypoint in the left to the matched keypoint in the right. We'll call this new image the putative-pair-image. Now, both of the SIFT packages have functions to do this. *****But you are not permitted to use them!!!***** This way you will have to identify the location of each keypoint and its match and explicitly draw the line. This tells us you have a good handle on the data structures that you are using.

2.2 Write the function to call the appropriate SIFT descriptor extraction function with the necessary input data structures. Do this for all the keypoints in both pairs of images. Then call the matching functions of VLFeat or OpenCV to compute the best matches between the left and right images of each pair. Then create the putative-pair-image for both `transA`–`transB` and `simA`–`simB` pair. You must write your own drawing function (note you may use OpenCV's `line()` function or MATLAB's `plot()` function).

Output: The code. And both of the putative-pair-images.

3 RANSAC

We're almost there. You now have keypoints, descriptors and their putative matches. What remains is RANSAC. To do this for the translation case is easy. Using the matched keypoints for `transA` and `transB`, randomly select one of the putative matches. This will give you an offset (a translation in X and Y) between the two images. Find out how many other putative matches agree with this offset (remember, you may have to account for noise, so "agreeing" means within some tolerance). This is the *consensus* set for the selected first match. Find the best such translation - the one with the biggest consensus set.

3.1 Write the code to do the translational case on `transA` and `transB`. Draw the lines on the adjoined images of the biggest consensus set.

Output: The code and the drawn upon adjoined image pairs. Also, say what the translation vector was and what percentages of your matches was the biggest consensus set.

For the other image pair we need to compute a similarity transform. Recall that a similarity transform allows translation, rotation and scaling. We can represent this transform with a matrix as:

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} a & -b & c \\ b & a & d \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

In other words, there are four unknowns. Each match gives two equations - so you need to pick two matches to solve, which is why similarity is called a two point transform. Clever.

3.2 Do the same as above but for the similarity pair `simA` and `simB`. Write code to apply RANSAC by randomly picking two matches, solving for the transform, and determining the consensus set. Draw the lines on the adjoined images for the biggest consensus set.

Output: The code and the drawn upon adjoined image pairs. Also, say what the transform matrix is for best set and what percentages of your matches was the biggest consensus set.

EXTRA CREDIT. Do as many as you wish, though 3.5X relies on 3.3X

For the second image pair we told you that the transform to compute was a similarity transform. But suppose you didn't know that. You might have guessed that it was an affine transform,

expressed as follows:

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

Here there are six unknowns. Again, each match gives two equations - so this time you need to pick three matches to solve, which is why affne is called a three point transform. Still clever?

3.3X Try estimating the affine transform between `simA` and `simB`. Write code to apply RANSAC by randomly picking three matches, solving for the transform, and determining the consensus set. Draw the lines on the adjoined images for the biggest consensus set.

Output: The code and the drawn upon adjoined image pairs. Also, say what the transform matrix is for best set and what percentages of your matches was the biggest consensus set.

Finally, given these transforms, you should be able to warp the second image to the first. We're not going to tell you about how to that here except we did talk about warping (remember backward warping?) earlier.

3.4X Create a new version of `simA` by warping `simB` "back" to the coordinate system of `simA` using the 3.2 transform you found. Call the new image `warpedB`. Show the two images (`simA` and `warpedB` overlaid by either blending them or by makng a pseudo color image wihere you put `simA` in the red channel and `warpedB` in the green channel of a color image (both (`simA` and `warpedB` are grayscale images).

Output: The code, `warpedB`, and the overlay image.

3.5X Do 3.4X again but this itme using the affine transform recovered in 3.3X.

Output: The code, `warpedB`, and the overlay image. And comment as to whether using the similarity transform or the affine one gave better results, and why or why not.