Problem Set 5: Harris, SIFT, RANSAC

Description

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. SIFT libraries you can use are described in this supplemental document (see question 2 for instructions).

What to submit

Download and unzip: ps5.zip

`ps5/`

- input/ - input images, videos or other data supplied with the problem set
- output/ - directory containing output images and other files your code generates
- ps5.py - code for completing each part, e.g. reading/writing images, function calls
- *.py - Python modules, any utility code
- ps5_report.pdf - a PDF file with all output images and text responses

Zip it as `ps5.zip`, and submit on T-Square.

Guidelines

1. Include all the required images in the report to avoid penalty.
2. Include all the textual responses, outputs and data structure values (if asked) in the report.
3. Make sure you submit the correct (and working) version of the code.
4. Include your name and GTID on the report.
5. Even if the code is not working, submit the code as the instructors can read through the algorithms to give partial credit. Comment your code appropriately.
6. Late submissions should be emailed to the TAs to be graded for partial credit.

Questions

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 filter. 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.

a. 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").

   Note: Since gradients have negative and positive values, you'll need to produce an image that is gray for 0.0, black for the negative extremum, and white for the positive extremum.

   Functions:
   - `gradientX(image) -> Ix`
   - `gradientY(image) -> Iy`
   - `make_image_pair(image1, image2) -> image_pair`

   Output: The gradient-pair image for both transA and simA:
   - transA gradient-pair image as `ps5-1-a-1.png`
   - simA gradient-pair image as `ps5-1-a-2.png`

b. Now, you can compute the Harris response for the image, defined 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 you have are the size of the window, the windowing function that controls the weights (together defining the *kernel*), and the value of α in the Harris scoring function. Remember, you can check your code on the checkerboard images, though those might use a different optimal window size than the real ones.

   Write code to compute the Harris response map. You can try with equal weights (uniform kernel), but it might work better with a smoother Gaussian that is higher at the middle and falls off gradually. Your output is a scalar map (single value per pixel). Apply your code to transA, transB, simA, and simB.

   Note: To display/write the output reasonably, you will have to scale the response values to be in a range of [0, 255] or [0.0, 1.0], depending upon how you deal with images.

   Function: `harris_response(Ix, Iy, kernel, alpha) -> R`

   Output: The Harris response image for:
   - transA image as `ps5-1-b-1.png`
   - transB image as `ps5-1-b-2.png`
   - simA image as `ps5-1-b-3.png`
   - simB image as `ps5-1-b-4.png`

c. Finally, you can find 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 half the side of a square window instead of a circle radius) over which a pixel has to be a maximum.

   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?

   Functions:
   - `find_corners(R, threshold, radius) -> corners`
   - `draw_corners(image, corners) -> image_out`

   Output: Apply your function to both image pairs: (transA, transB) and (simA, simB). Draw the corners visibly (in color) on each of the four result images and provide those images.
   - transA image with Harris corners marked as `ps5-1-c-1.png`
   - transB image with Harris corners marked as `ps5-1-c-2.png`
   - simA image with Harris corners marked as `ps5-1-c-3.png`
   - simB image with Harris corners marked as `ps5-1-c-4.png`

   Output (Textual Response): 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 the SIFT or SURF classes in OpenCV. Please check out the supplemental document for instructions on using SIFT in OpenCV/Python.

The use of a standard SIFT library consists of you just providing an image and the library does its thing: finding interest points at various scales and computes descriptors at each point. We're going to use the library 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 neighborhood size, scale setting (octave), and an orientation for each feature point (gradient angle at that pixel). We'll use the full-scale image, and hence fix octave to 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 X and Y gradient images! So you can create an *angle* image, and then for a given feature point at $<x_i, y_i>$ you can look up the desired angle.

a. Write a function to compute the angle image from X and Y gradients. Write another function to create OpenCV KeyPoint objects given interest points (corners) and response and angle images.

   Note that to instantiate a KeyPoint object, you'll need to supply two additional parameters: `_size` indicating the diameter of the region in which the keypoint was found (you can use double the radius supplied to `find_corners()` above), and `_octave` for the scale at which the feature was found (we fix this at 0 indicating full-scale image).

   Apply this to the set of corners you found above, and plot keypoints for all of transA, transB, simA, and simB on the respective images, with a little line/arrow that shows the direction of the gradient. In OpenCV, you can use the `drawKeypoints()` method to draw these points and directions.

   Functions:
   - `gradient_angle(Ix, Iy) -> angle`
   - `get_keypoints(points, R, angle, _size, _octave) -> keypoints`

   Output:
   - Interest points with angles shown on (transA, transB) pair as `ps5-2-a-1.png`
   - Interest points with angles shown on (simA, simB) pair as `ps5-2-a-2.png`

b. Now we're going to call the SIFT descriptor code. Pass in the list of keypoints found (each with location, response, angle, size, and octave defined) along with the *original* (grayscale) input image to OpenCV's SIFT.compute() method to extract descriptors. This process is covered in more detail in the accompanying supplemental document.

   Once we have the descriptors, we need to match them. In class, this is what we called *putative* matches. Given keypoints in two images, get the best matches. OpenCV has functions for computing matches. 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, the SIFT package has functions to draw matches. *****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.

   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 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).

Functions:
- `get_descriptors(image, keypoints) -> descriptors`
- `match_descriptors(desc1, desc2) -> matches`
- `draw_matches(image1, image2, kp1, kp2, matches) -> image_out`

Output:
- putative-pair-image for (transA–transB) as `ps5-2-b-1.png`
- putative-pair-image for (simA–simB) as `ps5-2-b-2.png`

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.

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

Function: `compute_translation_RANSAC(kp1, kp2, matches) -> translation, good_matches`

Output: Biggest consensus set lines drawn on pair (transA, transB) as `ps5-3-a-1.png`

Output (Textual Response):
- What translation vector was used?
- What percentage of your matches was the biggest consensus set?

b. 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.

Do the same as above but for the similarity pair <u>simA</u> and <u>simB</u>. 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.

Function: `compute_similarity_RANSAC(kp1, kp2, matches) -> transform, good_matches`

Output: Biggest consensus set lines drawn on pair (<u>simA</u>, <u>simB</u>) as `ps5-3-b-1.png`

Output (Textual Response):
- What is the transform matrix for the best set?
- What percentage of your matches was the biggest consensus set?

EXTRA CREDIT QUESTIONS (3-c, 3-d, 3-e)

c. 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 affine is called a three point transform. Still clever?

Try estimating the affine transform between <u>simA</u> and <u>simB</u>. 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:
- Biggest consensus set lines drawn on pair (<u>simA</u>–<u>simB</u>) as `ps5-3-c-1.png`

Output (Textual Response):
- What is the transform matrix for the best set?
- What percentage of your matches was the biggest consensus set?

d. 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.

Create a new version of simA by warping simB "back" to the coordinate system of simA using the 3-b transform you found. Call the new image warpedB. Show the two images (simA and warpedB overlaid by either blending them or by making a pseudo color image where 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:
- warpedB image as `ps5-3-d-1.png`
- the overlay image as `ps5-3-d-2.png`

e. Do 3-d again but this time using the affine transform recovered in 3-c.

Output:
- warpedB image as `ps5-3-e-1.png`
- the overlay image as `ps5-3-e-2.png`

Output (Textual Response):
Comment as to whether using the similarity transform or the affine one gave better results, and why or why not.