Assignment 07

Please refer to MyCourses/Gradescope for assignment deadlines.

The goal of this assignment is to give you hands-on practice with the following concepts:

- Corner detection (Harris algorithm)
- Feature detection and descriptors (SIFT algorithm)
- The Random Sample Consensus (RANSAC) algorithm
- Feature matching with RANSAC to automate the panorama-stitching process from A2

Part 1 of 2: Harris Corners

Your mission is to implement the Harris corner detection algorithm using numpy and some "low level" OpenCV functions. Recall from class that the corner response function at each pixel is given by

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$$

where λ_1 and λ_2 are the eigenvalues of the second moment matrix. The second moment matrix is defined at every pixel, and is given by

$$M(i,j) = \sum_{k,l} \begin{bmatrix} G_x(i+k, j+l)^2 & G_x(i+k, j+l)G_y(i+k, j+l) \\ G_x(i+k, j+l)G_y(i+k, j+l) & G_y(i+k, j+l)^2 \end{bmatrix}$$

where G_x and G_y are the image gradients in the x and y directions (i.e. outputs of Sobel operations). You may use `cv.Sobel()` for computing `Gx` and `Gy`.

A useful way to think about this is that, rather than having a 2x2 matrix at every pixel, we can instead keep track of all the $M[0,0]$ values as a 2D array, all the $M[0,1]$ values as a 2D array, and all the $M[1,1]$ values as a 2D array (we don't need $M[1,0]$ because it equals $M[0,1]$). This is the recommended approach (see the following diagram):

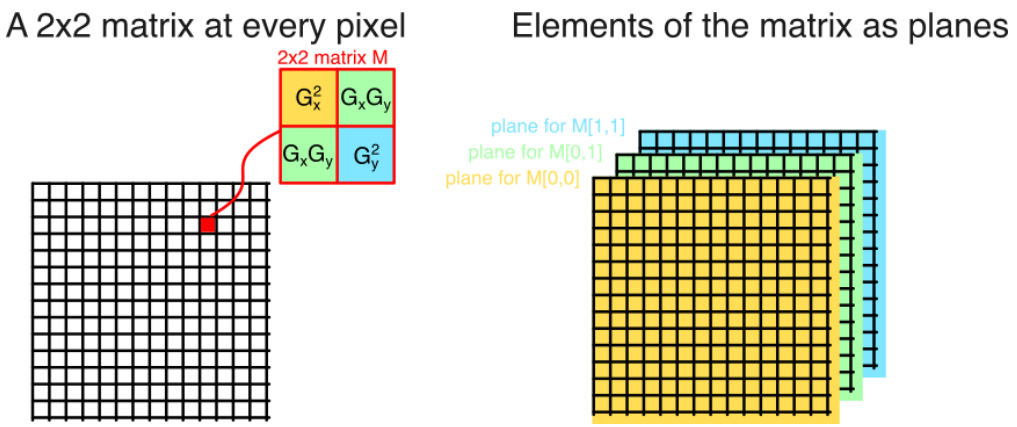

Figure 1: Rather than a "2x2 matrix at each pixel", think about having a separate plane for the elements of the 2x2 matrix.

The following identities about sums and products of eigenvalues will also be useful:

- $\lambda_1 + \lambda_2 = \text{trace}(M)$
- $\lambda_1 \lambda_2 = \det(M)$

...and, you can easily look up what the trace and determinant of a 2x2 matrix are in terms of its elements. In other words, you can compute R using simple multiplies and sums of the three yellow/green/blue image planes in the diagram above.

One last hint: the $\sum_{k,l}$ in the definition of $M(i,j)$ is a sum over a window of pixels and may be implemented using a simple box filter!

What you need to do: Fill in the missing code in `harris_corners.py`. When you run the file, it will display an interactive window with OpenCV corners on the left and yours on the right. If you do everything correctly, they should match.

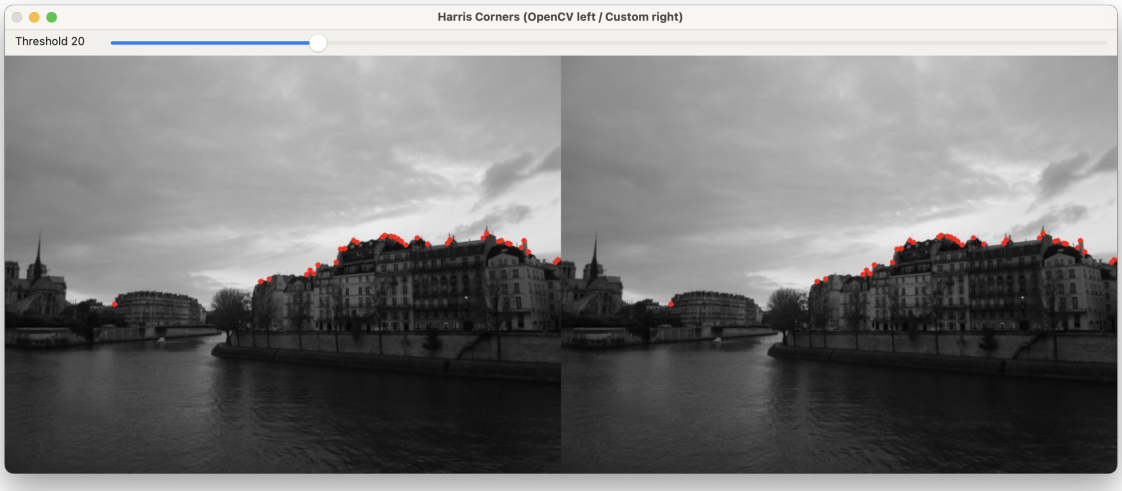

Figure 2: Screenshot of a working solution

Part 2 of 2: Improved Panorama: Matching SIFT features with RANSAC

Go watch **this 8-minute video** on feature-matching for panorama-stitching using the RANSAC algorithm. You may also want to watch **this 12-minute follow-up video** if you want to see some more details on how the underlying `warpPerspective()` function works using a backwards mapping and where the edge weighting comes from.

You've been provided with (a slightly modified) version of the answer key file for `panorama.py` from Assignment 2. Back then, you had to manually click on corresponding points in a pair of images to align them. We're now going to update the procedure using SIFT feature matching, which will also enable us to automatically align more than two images at once.

The main differences from Assignment 2 are:

- The `get_clicked_points_from_user()` function is gone
- All the buggy A2 code is now correct, so you don't need to touch any of the old functions.
- Three new functions have been added:
  - `get_matching_points()` - this function uses SIFT feature matching to find corresponding points between two images. This involves (i) detecting SIFT keypoints and getting their descriptors using a `cv.SIFT` object, then (ii) finding matches between the descriptors across two images using the `cv.FlannBasedMatcher`. You may use any CV functions you like here.

- `my_find_homography()` - this function is essentially pure-numpy implementation of the `cv.findHomography()` function. It is (should be) bug-free and provided for you, since we will be asserting in test cases that `find_homography_ransac()` makes no calls to any `cv` functions.
- `find_homography_ransac()` - this function is where you will implement the RANSAC algorithm to find the best homography between two sets of matching points. This is the only function you need to implement for this part of the assignment.

The main steps of the overall panorama stitching algorithm are:

1. Choose a 'reference' image. We'll let the user specify this by providing a list of image files and an index like `-r=1` such that the `images[r]` image is the reference.
2. For each `images[i]`, use SIFT feature matching to find a set of `xy` points whose descriptors match between `images[i]` and the reference image `images[r]`. This is provided for you in the function `get_matching_points()`. Importantly, the returned points are such that `points_ref[i]` is some `xy` point in the reference image and `points_q[i]` is the "best matching" `xy` point in the second "query" image, *but the "match" only takes into account the descriptors.* Thus, we still need another step of the algorithm to figure out which subset of the matching points are 'inliers' and which are 'outliers.' This is where RANSAC comes in.
3. For each pair of `(points_src, points_dst)` points, find the best homography that lines up the greatest number of corresponding points in the two images. This will return a homography matrix that maps *from* the src coordinate system *to* the dst coordinate system. **You need to implement this part in the `find_homography_ransac()` function.** As in past assignments, this is something that would normally be done by a call to OpenCV functions, but you will be implementing RANSAC yourself using numpy and python. The pseudocode for this algorithm is as follows:

```
for N iterations:
    sample 4 random indices `idx` [0..n-1] and select
        ... points_src[idx] and points_dst[idx]
    compute the homography H that maps between them using
        ... my_find_homography()
    use H to map *all* points_src to points_dst
    compute the distance for all `n` pairs of points
    count the number of 'inliers' (i.e., points whose distance is
       ... less than some threshold)
    if the inlier count is greater than the best inlier count so far:
           record the indices of all the inliers
recompute H using all the inliers from the best set of inliers
```

```
    return H
```

4. Pass the resulting list of images and homographies into the stitching code from Assignment 2

Example output:

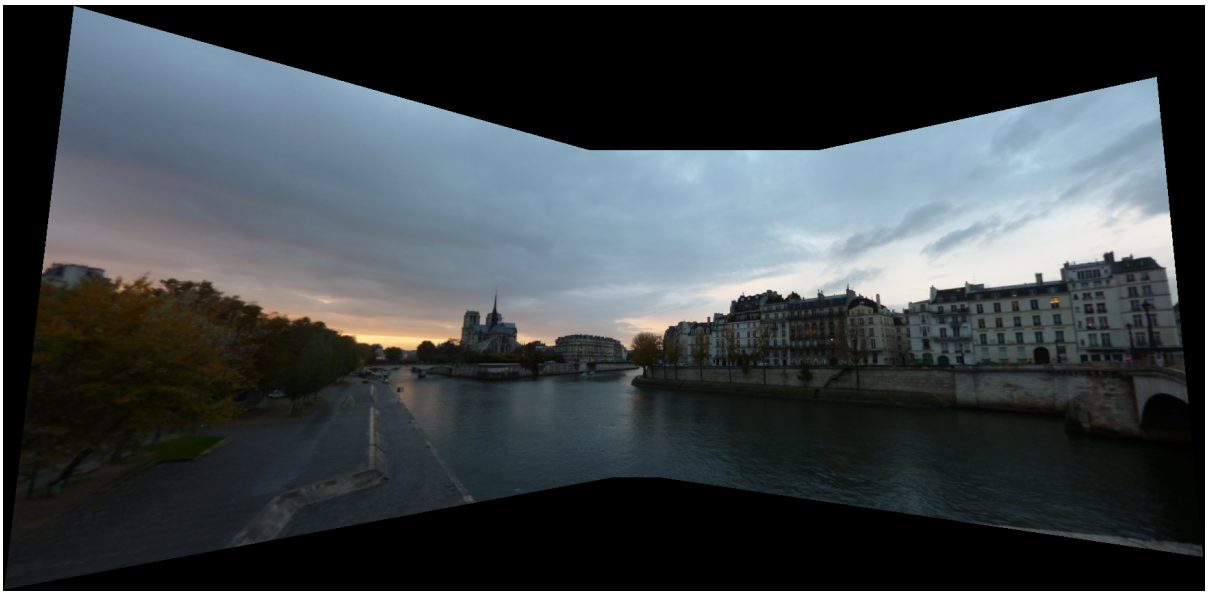

Figure 3: Result of `python panorama.py src/images/paris_a.jpg src/images/paris_b.jpg src/images/paris_c.jpg --reference-index=1`

Pay attention: While `get_matching_points()` is provided, future quizzes/tests may ask some questions about how SIFT works. We may also ask questions about the RANSAC algorithm for line or circle detection (as an alternative to the Hough Transform), so make sure you understand each step of what you're implementing and how it might apply in other contexts!

Collaboration and Generative AI disclosure and Assignment Reflection

Did you collaborate with anyone? Did you use any Generative AI tools? How did the assignment go? Briefly explain what you did in the `collaboration-disclosure.txt` file. **Completing the collaboration disclosure for each assignment is required.** Completing the reflection part of the file is optional, but it is a good way to give feedback to the instructor and grader about how the course is going.

Submitting

As described above, use `make submission.zip` to generate a zip file that you can upload to Gradescope. You can upload as many times as you like before the deadline. To account for late penalties, contact the instructor or grader directly if you plan on uploading any further revisions after the deadline.