CS 558: Homework Assignment 3
Due: November 7, 6:00pm – 21 days from now!

Philippos Mordohai
Department of Computer Science
Stevens Institute of Technology
pmordoha@stevens.edu

Collaboration Policy. Homeworks will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems. Use of the Internet is allowed, but should not include searching for previous solutions or answers to the specific questions of the assignment. I will assume that, as participants in a graduate course, you will be taking the responsibility of making sure that you personally understand the solution to any work arising from collaboration.

Late Policy. A total of **3 late-days** will be allowed in the **entire semester** without penalty. Days will be counted by rounding up, e.g. being 5 minutes late will be counted as one of the late-days. Assignments will no longer be accepted after all late-days have been used.

Deliverables. In a zip file include:

1. The complete **Jupyter notebook** with all cells executed. **Make sure that the filename starts with your Stevens username.**
2. The above in **pdf format**.
3. A **report in pdf format** explaining what you did and answering the questions in each problem.

Please read the following list carefully:

- You should not change the provided function specifications, but you can add your own functions as needed.
- You are not allowed to use **any** library functions that operate on the images. Functions for reading and writing from and to file and for displaying images on the screen have been provided. (See the NumPy tutorial for examples.) Math functions from NumPy are allowed, but no filtering, convolution, resizing, edge detection etc. functions that have not been implemented by you may be used. **Speficic OpenCV function that are explicitly named in this documents or the comments in the skeleton code should be used.**

Guidelines and Hints

- Specify the path to the input as in the skeleton code and append specific filenames in separate lines, so that we can replace them with our own. In other words, preserve the structure of the skeleton code.
- If you develop using a different Python setup, you still have to submit a notebook by populating the boxes. Under the *Runtime* menu, there is an option for executing all cells.
- Do not activate *playground mode* so that your changes to the notebook are preserved.
- You can use `gimp` or `pixspy.com` to read the coordinates and RGB values of a pixel.
- **Keep track of what x and y correspond to.** Which one spans the rows and which one spans the columns of the image. Inconsistent notation and usage can cause trouble. The way I keep my notation consistent is:
  1. In math operations, x comes before y. For example the homography is applied as $H \cdot [x,\ y,\ 1]^T$.
  2. y is always used as the row index and x as the column index, as in `image[y, x]`.
  3. I avoid mixing in other letters and having to manage that i behaves like x etc.
- All vectors are column vectors.

Problem 1: Homography Fitting (25 points)

Inputs: 9 files named `points_ case_k.npy` for $1 \leq k \leq 9$ in directory `hdata`.

These files contain matrices where each row contains four integers representing two corresponding pixels: $[x_i, y_i, x'_i, y'_i]$. The first eight files are transformed letters M. The correspondences in all 9 files are perfect, but with rounded pixel coordinates.

The objective of this problem is to fit a homography $H \in \mathbb{R}^{3\times3}$ that satisfies $\mathbf{p'}_i = H\mathbf{p}_i$ where $\mathbf{p}_i = [x_i, y_i, 1]$ and $\mathbf{p'}_i$ is defined similarly. In other words, the homography maps points from the coordinate system of the first two columns to the coordinate system of the third and fourth column. See Fig. 1 for an example.

Each pixel correspondence contributes two equations to be stacked in the data matrix A, as follows:

$$A\mathbf{h} = \begin{bmatrix} & \cdots & \\ \mathbf{0}^T & -\mathbf{p}_i^T & y'_i\mathbf{p}_i^T \\ \mathbf{p}_i^T & \mathbf{0}^T & -x'_i\mathbf{p}_i^T \\ & \cdots & \end{bmatrix} \begin{pmatrix} \mathbf{h}_1 \\ \mathbf{h}_2 \\ \mathbf{h}_3 \end{pmatrix} = 0 \tag{1}$$

where $\mathbf{0}$ is a column vector with three zeros. $\mathbf{h}_1$ is the transpose of the first row of H (i.e. a column vector) while $\mathbf{h}_2$ and $\mathbf{h}_3$ are defined similarly.

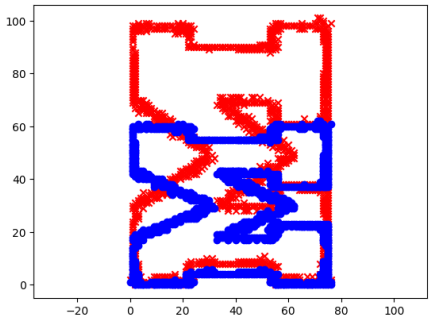

Figure 1: One of the inputs for Problem 1. The first two columns of the matrix stored in the `npy` file represent the red points, while the third and fourth columns are the coordinates of the blue points. The objective is to fit a homography that maps each red point to the blue point in the same row of the input.

After stacking all equations in A, the desired homography is eigenvector of $A^T A$ corresponding to smallest eigenvalue, and can be found using `np.linalg.svd(A)`. This returns a 9D vector which should be reshaped to a 3×3 matrix to yield H.

Hints

- **Keep track of what x and y correspond to.** Which one spans the rows and which one spans the columns of the image. Inconsistent notation and usage can cause trouble. (Cannot repeat enough times.)
- In general, do not assume that the last coordinate of $\mathbf{p}_1$ is 1. However, when constructing such a vector using the inhomogeneous coordinates (x and y), you should set it to 1.
- Be aware at all times of the direction of the mapping of your homography. Does it map from the first to the second set, or the other way around? Use the inverse if needed.
- Before dividing a homography matrix by its last element, you should check that it is not 0, in general. This should not happen on these data. If it does, something has gone wrong.
- Once you have computed a homography, it may be useful to verify that one point from the source maps correctly to the destination. This can be done by looking at one row of the input matrix.
- Do not use a minimal set of *consecutive* input rows to fit H because the points are too close to each other and rounding errors may cause trouble. You can pick points far apart to do this.
- The transpose is also the inverse only for rotation matrices. `np.linalg.inv()` will compute the inverse of a matrix.
- I generated the scatter plots in Fig. 1 using:

```
plt.scatter(X[:,0], X[:,1], s=2, c='r', marker='x')
plt.axis('equal')
```

After the transformation, compute the distance between each mapped point and its ideal destination. For example, if point $[3,\ 4]$ corresponds to point $[1,\ 3]$ (they are in the same row of the input), and is mapped to point $[1.5,\ 2.5]$ by the homography, the distance is $\sqrt{(1-1.5)^2+(3-2.5)^2} = \sqrt{0.5^2+0.5^2} = 0.7071$. Average all these distances for each case and include them in the report.

Report:

1. Include H points_case_3.npy and points_case_5.npy. You must normalize the last entry to 1.
2. Also include visualizations of the original points $[x_i, y_i]$, target points $[x_i', y_i']$ and the points after applying the provided function `homography_transform(X, H)` in one figure. Please do this for points_case_4.npy and points_case_9.npy. There should be two plots, each containing 3 sets of N points. Figure 1 above contains two sets of points. (The mapped points of the third set should overlap the second set. This is an indication of success.)
3. Finally, include a table with the average distances between mapped and destination points for *all* 9 cases, as described above. These are the fitting errors.

Problem 2: Image Stitching (75 points)

Inputs: two images to be stitched from six different scenes are provided in the `eynsham`, `florence2`, `florence3`, `mertonchapel`, `mertoncourtyard` and `vgg` subdirectories of the `stitch` directory in the zip file.

In this problem, the objective is to stitch images by fitting a homography to correspondences detected by matching keypoint descriptors. Since these correspondences may contain mismatches, RANSAC will be used to find the best homography. Then, one image should be warped and blended with the other, extending the canvas as much as necessary to fit all pixels. See Fig. 2.

Since implementing a detector and descriptor such as the ones in SIFT is beyond the scope of a homework, you will use the AKAZE detector and descriptor that comes with OpenCV. Relevant functions are also provided in the Jupyter notebook and can be used directly as described in the steps below.

A keypoint has a location p_i and descriptor d_i. The calling convention is:
`keypoints, descriptors = get_AKAZE(image)`
where `get_AKAZE()` is a function provided to you. It returns an $N \times 4$ matrix `keypoints` and an $N \times F$ matrix `descriptors` for the N detected keypoints. The first two columns of `keypoints` contain x, y; the last two are the angle and (roughly) the scale at which they were found in case those are of interest. Only the coordinates will be needed for this problem. The descriptor has also been post-processed so that the Euclidean distance between two descriptors

Figure 2: The two input images and the final stitched output for the `vgg` example. Notice how the left input image appears undistorted in the output, while the second input image has been warped to match the first one.

is meaningful. In other words, the norm of the vector $\mathbf{d}_i - \mathbf{d}'_j$ is the dissimilarity between two descriptors. If this was 0, the descriptors would be identical and thus a potentially perfect match. Typically, $\mathbf{d}_i$ is from one image and $\mathbf{d}'_j$ is from the other.

The functions related to homographies from Problem 1 should be used directly.

The steps you should implement are as follows:

Step 1: Extract keypoints and descriptors from the images. You can use the provided function `get_AKAZE(img)` for this. You can visualize the locations of the detected keypoints using `plt.imshow()` and `plt.scatter()` (see above) before `plt.show()`. This should generate images like Fig. 3. (There is no reason to worry about the inconsistency in the order of the color channels between OpenCV and `matplotlib`. `img2 = cv2.cvtColor(img1, cv2.COLOR_BGR2RGB)` can be used to flip the red and blue channels, and therefore works for switching the format in both directions, but this is not required.)

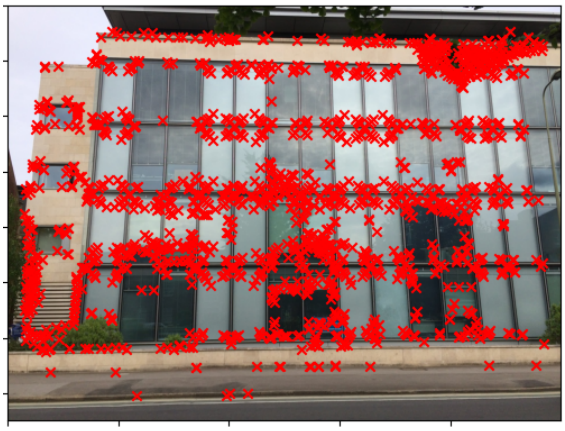

Figure 3: AKAZE keypoint locations in an image from the `vgg` example

Step 2: Detect putative correspondences. The goal here is to determine which keypoint in the second image is the most likely correspondence for a given keypoint in the first image. These correspondences will be the input to RANSAC. (We cannot use all possible pairs of keypoints considering that if 1,000 keypoints are detected in each image, that would yield 1,000,000 putative correspondences, with at least 99.9% of them being outliers. These settings are too hard even for RANSAC.)

To identify correspondences that are likely to be true, you should compute the distances between all descriptors from the first image to all descriptors from the second image. Then, apply the distance ratio test to select unambiguous correspondences. You will have to pick a threshold for the ratio test. A value between 0.7 and 1 is reasonable, but you should experiment with it. Keep in mind that AKAZE behaves differently than SIFT.

Optionally, you can apply the distance ratio test in both directions, ensuring that the correspondences appears to be unambiguous with respect to both images.

This step should be implemented by the `find_matches()` function, which calls the `compute_distance()` function. This is the most computationally expensive part of the assignment. Use the split I suggest in the notebook so that you do not have to repeat it many times.

Then, the `draw_matches()` function should be used to draw lines connecting these correspondences. The lines should be overlayed on an image formed by stacking the two inputs vertically. You should use `cv2.line()` for this purpose. See Fig. 4.

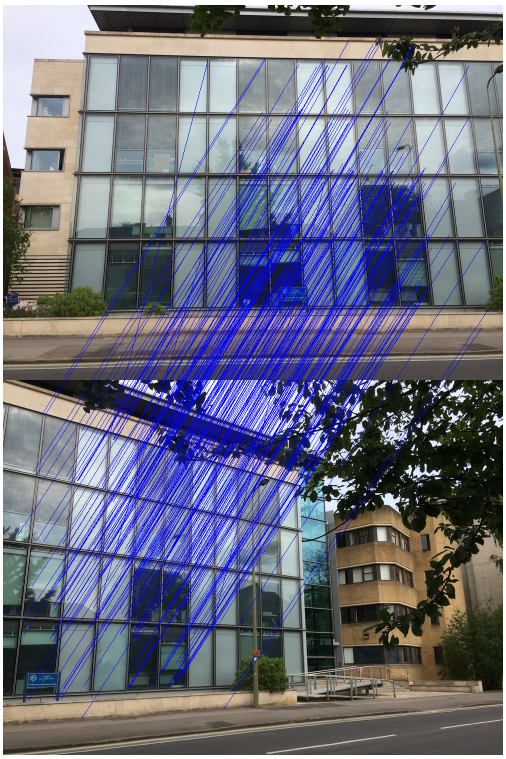

Figure 4: Putative correspondences for the `vgg` example

Step 4: RANSAC for homography fitting. Use `get_match_points()` to generate the list of pixel coordinates of corresponding keypoints and then implement `RANSAC_fit_homography()` to fit the best homography.

You should keep track of the best set of inliers you have seen in the RANSAC loop. Once the loop is complete, **re-fit the model to these inliers**. In other words, for N iterations of RANSAC, you should fit a homography N times on the minimum number of points needed; this should be followed by a single fitting of a homography on many more points (the inliers for the best of the N models). You will need to set epsilon's default value: 0.1 pixels is too small; 100 pixels is too big.

When sampling correspondences, draw without replacement to avoid degenerate solutions.

Step 5: stitching the images. The objective of this step is to stitch the two images by transforming one onto the canvas of the other. See how the left image is just copied, without being warped, in Fig. 2. The second image (in the middle of the figure) is warped onto the canvas of the first one.

You should make the canvas of the new image large enough to fit every pixel of both images. Be aware that the warped image may break any, none or all boundaries of the un-warped one. If the warped image extends to the left and above of the un-warped one, this has to be accounted for.

When both images map to a pixel of the output, picking either one is fine. In other words, there is no need to blend them. If only one of the images map to a pixel of the output, then that image should be selected (obviously). Some pixels of the output will be black if they receive no input pixels.

Implement bilinear interpolation for the warping. Note that this must handle RGB values for this problem. You can use nearest-neighbor interpolation incurring a small penalty.

Hint

For debugging the entire process, you can provide two images that are crops of the same larger image, (e.g., `I[100:400,100:400]` and `I[150:450,75:375]`). This allows you to verify that keypoints are matched correctly by visually inspecting a row of the `XY` array passed to RANSAC. It is also easy to verify the homography, since it is just a translation, i.e. the top-left 2×2 submatrix should be identity and the first two elements of the last row should be 0.

Report:

The report should be based on one image pair of your choice, other than `vgg` used in this document.

Step 1: report the number of keypoints extracted from each of the two images.

Step 2: report the number of putative correspondences.

Step 2: include a picture of the matches between the image pair, similar to Fig. 4.

Step 3: report the number of inliers found by RANSAC.

Step 4: show the final stitched image like Fig. 2 (right).