Exercise 2 - Panorama Registration & Stitching

Due date: 10.02.2026

1 Overview

In this exercise, you will build a panorama image from a sequence of images. The image sequence comes from a video scanning a scene from left to right (we assume a rigid transformation (rotation and/or translation) between images), with significant overlap in the field of view of consecutive frames.

This exercise consists of the following steps:

- Registration: find the geometric transformation between each consecutive image pair by:
  1. Detecting **Harris feature points**
  2. Extracting their **descriptors**
  3. Matching descriptors between images to find corresponding pairs
  4. Fitting a rigid transformation that agrees with a large set of inlier matches using the **RANSAC** algorithm.
- Stitching:
  1. Warping each frame by a cumulative transformation so it is aligned with the middle frame.
  2. combining strips from aligned images into a panorama.

2 Utils

"*ex2_empty.py*": the code outline to be filled. You are required to submit this file, after filling it and changing its name to: "$ex2\_ < student\_id > .py$".

Additionally, you are supplied with a "*utils.py*" file and some helpful functions in the outline file, containing the following useful functions with their documentation.

1. *read_image(filename, representation)*
2. *build_gaussian_pyramid(im, max_levels, filter_size)*

3. *blur_spatial(im, kernel_size)*
4. *non_maximum_suppression(image)*
5. *spread_out_corners(im, m, n, radius, harris_corner_detector)*
6. *visualize_points(im, points)*
7. *estimate_rigid_transform(points1, points2)*
8. *filter_homographies_with_translation(homographies, minimum_right_translation)*

Make sure the required packages for this exercise (numpy, scipy, skimage, matplotlib) are installed in your environment. Do not import additional packages except for the ones already imported in the supplied files, and do not use existing implementations (e.g. cv2 implementations) of algorithms you are required to implement.

You are also given some input examples that you can use for debugging using the code under main, and you are encouraged to try different inputs and parameters to further test your code.

3 Image Pair Registration

In this part, we will perform image registration between a pair of overlapping images as learned in class. For each image, we will first identify feature points and extract descriptors. Next, we will find matching points and, based on those, calculate the homography for aligning one image to the other. Finally, we will apply the transformation over the first image to get two aligned images.

3.1 Feature Points Detection

As discussed in class, good feature points are points that are distinctive, well-localized, and stable to noise, viewpoint, and illumination changes. A good option for such features we saw in class is corners. Therefore, you will implement the Harris corner detector in $harris\_corner\_detector(im)$ to identify feature points in each frame. This function gets a grayscale image and returns (x, y) locations that represent corners in the image.

You should implement the following algorithm:

1. Get the I_x and I_y image derivatives using the filters $[1, 0, -1], [1, 0, -1]^T$, respectively.
2. Blur the images: I_x^2, I_y^2, $I_x I_y$. You may use the supplied *blur_spatial* function with kernel_size = 3.
3. This will yield the matrix:

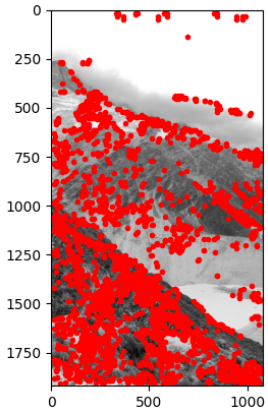
$$M = \begin{bmatrix} I_x^2 & I_x I_y \\ I_y I_x & I_y^2 \end{bmatrix}$$

As learned in class, looking at the eigenvalues of this matrix will tell us the intensity changes of a window neighborhood around each pixel. A small change means this is a smooth area. A large change in one direction (one large eigenvalue) means this window contains an edge, and A large change in two directions (two large eigenvalues) means that this window contains a corner. To measure how big the two eigenvalues are, we will use the response: $R = det(M) - \alpha(trace(M))^2$ with $alpha = 0.04$.

4. The corners are the local maximum points of the response image R. To find these points you can use the supplied *non_maximum_suppression* function, which gets a response image as an input, thresholds out areas with low response, and returns a binary image containing the local maximum points.

5. Return the (x, y) coordinates of the corners.

To visualize the detected corner points, use the supplied *visualize_points* function. You should get something similar to the below.



3.2 Feature Points Description

After detecting feature points in two images, we want to match them. To do this, you will implement a simplified MOPS-like descriptor in

$feature\_descriptor(im, points, desc\_rad = 3)$.

The goal of the feature descriptor is to represent the local appearance around each feature point in a way that is robust to small changes in illumination and noise. Note that this simplified descriptor does not include rotation invariance (unlike the full MOPS descriptor) for better performance, but it is sufficient for our panorama stitching case where consecutive frames have limited rotation between them, and RANSAC handles finding the inlier matches.

For each detected feature point, you should perform the following steps:

1. Extract a square image patch of size $(2desc\_rad + 1) \times (2desc\_rad + 1)$ pixels centered at the feature point. With $desc\_rad = 3$, this yields a 7×7 patch.

2. Sample the patch using bilinear interpolation via *scipy.ndimage.map_coordinates*.

3. Normalize the resulting patch by subtracting its mean and dividing by its l2 norm (you can use *np.linalg.norm* to calculate the norm). Handle the case where the norm is zero to prevent division by zero.

The final descriptor for each feature point is therefore a normalized 7×7 patch (a 49-dimensional vector when flattened).

3.3 Find Features

For some scale robustness, we will extract the features and their descriptors from the third level of a gaussian pyramid. Additionally, to spread corners around the whole image and avoid focusing on one area with a lot of texture, you are provided with a *spread_out_corners* function you should use instead of calling *harris_corner_detector* directly. This function splits the input image into $n \times m$ approximately equal sub-images and runs a *harris_corner_detector* on each. You are encouraged to experiment with the values of the parameters to the function, but a good starting point would be $n = 7, m = 7, radius = 12$.

Together, the function $find\_features$, should:

1. generate a gaussian pyramid with 3 levels.

2. call $spread\_out\_corners(im, m = 7, n = 7, radius = 12)$. This will return spread out corners across the image.

3. convert the location of points to their location on the third level, by $p_3 = 2^{13}p_1 = 2^{-2}(x_i, y_i) = \frac{(x_i, y_i)}{4}$.

4. extract descriptors from the convert points using the third pyramid level image: $feature\_descriptor(pyr[2], points\_level3, desc\_rad = 3)$

5. return the feature points and their descriptors

3.4 Feature Points Matching

Now that each feature point has a descriptor, they can be used to match feature points between images, using a distance-based similarity measure. You will implement the function $match\_features(desc1, desc2, min\_score)$ that compares feature descriptors from two images and returns the ones that have higher similarity score than *min_score*. The match score between two feature descriptors will be their dot product, after flattening each to 1D arrays. We will say the two feature points match, if their dot product satisfies the following properties:

1. the dot product S_j, k between the jth descriptor in frame i and the kth descriptor in frame $i+1$ is in the best 2 features that match feature j in image i, from all the features in image $i+1$.

2. S_j, k is in the best 2 features that match feature k in image $i+1$, from all the features in image i.

3. S_j, k is greater than the min_score.

Note that the dot products will be in the range [-1,1]. you should play with the min_score until you get good matches. A good place to start from is $min\_score = 0.5$.

3.5 Apply Homography

In the next section you will implement RANSAC to find a homography between frame i and frame $i+1$. To use RANSAC we first need to implement a function for applying a homography over a set of points *apply_homography*. *apply_homography* takes an array of 2D points and a 3×3 homography matrix and applies the homography onto the given points. Note that you to convert the 2D points to homogenous coordinates before applying the homography to them, and then project the result back to 2D.

3.6 RANSAC

After implementing *apply_homography*, you can implement the *ransac_homography* function that uses it to find the homography between two images given their matches. The matches might contain outliers, therefore we will apply RANSAC to find the homography that the most points agree with. In this exercise, we will assume the transformation between frames is rigid (i.e. only translation and rotation), so only 2 point correspondences are needed. *ransac_homography* gets 2 sets of points, where each point $p_{i,j}$ is the point coordinates for match j in $image\_i$. The algorithm you will implement is:

For iter in num_iters do:

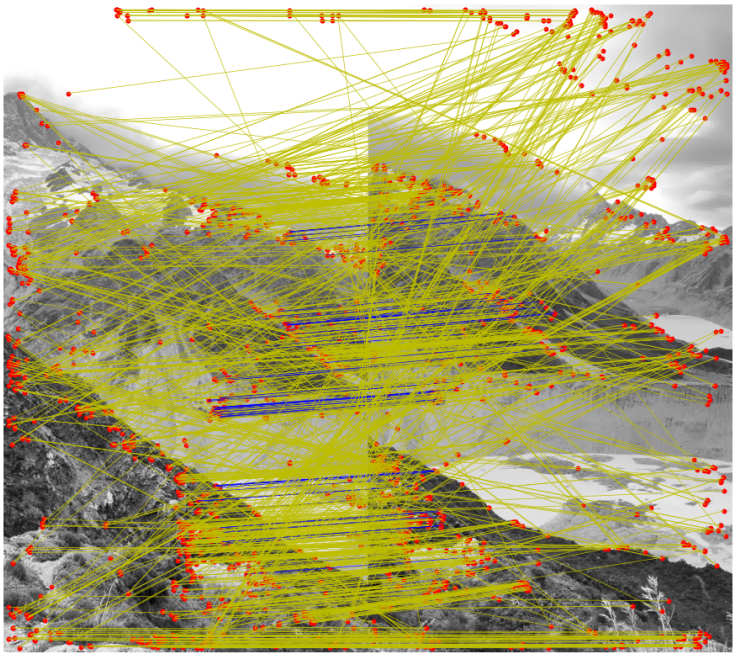
1. Randomly sample 2 point correspondences from the supplies N matches: $P_{1,i}, P_{2,i}, P_{1,j}, P_{2,j}$.

2. Compute the homography $H_{1,2}$ that transforms $P_{1,i}, P_{1,j}$ into $P_{2,i}, P_{2,j}$ using the supplied $estimate\_rigid\_transform$ function that returns a 3×3 matrix that perform the transformation.

3. Apply $H_{1,2}$ over the P_1 points using your implemented *apply_homography*, and compute the squared Euclidean distance: $||H_{1,2} \begin{bmatrix} P_{1,i} \\ P_{1,j} \end{bmatrix} - \begin{bmatrix} P_{2,i} \\ P_{2,j} \end{bmatrix} ||^2$. Mark all points with distance $<$ *inlier_tol* threshold as inlier matches, keeping a record of the largest inliers set.

Once *num_iters* is done and an inliers set has been found, recompute the homography with all inliers. Finally, the *ransac_homography* function will return the homography and an array with the indices of the inlier matches.

To visualize the final matches, implement the *display_matches* function that gets 2 images, 2 point correspondences array, and the inliers set from the *ransac_homography* function, and displays the images horizontally concatenated, with all matching points overlayed on top of the images as red dots. Then, between each outlier match you should draw a yellow line, and between each inlier match – a blue line. You should get a result similar to the below (it's ok if it doesn't look exactly the same, due to the randomness of RANSAC).



4 Building the Panorama

4.1 Accumulate homographies

The *ransac_homography* function you implemented finds homographies between each 2 frames. However, to build one panorama, we want to choose a common coordinate system and align all images to this coordinate system. Therefor, we will choose the middle frame in our frames sequence to be our reference image and use its coordinate system. Next, we will need to generate homographies from each frame into the reference one. The best way to do so, is by accumulating homographies. Meaning, the homography from frame i to the reference frame m will be the dot product of all homographies until m, where:

- if $i < m$: $H_{i,m} = \Pi_{j=m-1}^{i} H_{j,j+1}$.
- if $i > m$: $H_{i,m} = \Pi_{j=m}^{i-1} H_{j,j+1}^{-1}$.
- if $i = m$: $H_{i,m} = I(3,3)$ (a $x \times 3$ identity matrix)

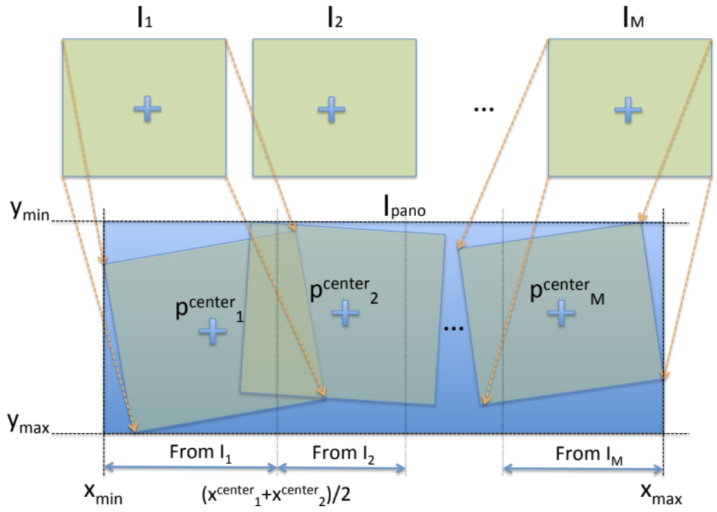
You will implement this in the function *accumulate_homographies*.

In this exercise, we want to always maintain the prpperty that $H[2,2] == 1$. This should be done by normalizing the homographies using $H = H/H[2,2]$ after calculating them.

4.2 Warp images

In this exercise we will use backward warping. To do so, we first need to define the region where we want the panorama to be rendered. We want this region to be large enough to include all frames. To do so, you will implement the function *compute_bounding_box* that for each frame, computes where its 4 corners land after applying the $H_{i,m}$ homography to them. Then, it returns the bounding box containing all corners by returning the $[x_{min}, y_{min}; x_{max}, y_{max}]$.

Next, we need to choose which parts of the panorama will come from each frame. To do so, you will divide the panorama into M vertical strips, each covering a portion of $[x_{min}; x_{max}]$. The boundaries between strips $i, i+1$ will be $x_i^{center} + x_{i+1}^{center}/2$, where x_i^{center} is the x coordinate of the center of ith frame strip P_i^{center} in the panorama coordinates system (after applying the homography), as shown below.



This will be done by the supplied *generate_panoramic_images* function.

Finally, you will implement back-warping for each strip in the functions *warp_channel*, *warp_image*. *warp_channel* takes a single-channel image and

an homography, and back-warps the image by the homography. This function should first compute the bounding box using your implemented *compute_bounding_box*. Then, prepare coordinate strips using *np.meshgrid*. Now, the coordinate mesh created for this image should be transformed by the inverse homography $H_{i,m}^{-1}$, using *apply_homography* to transform the grid back to the coordinate system of frame i. These back-warped coordinates can now be used to interpolate the image with *scipy.ndimage.map_coordinates*.

warp_image calls *warp_channel* for every channel in (R, G, B) and stacks the results together to generate RGB panoramas.

4.3 Stitching it all together

The supplied function *generate_panoramic_images* called from the main function will generate *num_panoramas* panoramas using your implemented functions and save them to *out_dir*. An example panorama using the supplied video "mt_cook.mov":

You are encouraged to also try it over your own videos.
Good luck and enjoy!