Assignment 4: Feature points, matching, homography

General instructions: The assignment is composed of the compulsory tasks and optional tasks (denoted by ★). To approach the assignment defense, you are required to complete at least the compulsory parts. Successfully defending the compulsory part will give you a maximum 75 points out of 100. At the defense, the obligatory tasks **must** be implemented correctly and completely. If your implementation is incomplete or has major errors, you will not be able to successfully defend the assignment. You can choose arbitrarily among the optional tasks to reach the 100 points. The number of points for an optional task is given in the task description. There might be more than 25 points available for an assignment. However, the maximum possible number of points for a single assignment is 100. The maximum overall number of points obtained from the 6 assignments is 600.

Required formating and submission: Create a folder `assignment4` that you will use during this assignment. Unpack the content of the `assignment4.zip` that you can download from the course webpage to the folder. Save the solutions for the assignments as Python scripts to `assignment4` folder. In order to complete the assignment you have to present your solutions to the teaching assistant. Some assignments contain questions that require sketching, writing or manual calculation. Write these answers down and bring them to the assignment defense as well. The code must be submitted on the e-classroom **before** the defense. Submit the code as .py source files (not Jupyter notebooks). If you have more that one source file, submit a zip of .py files.

COMMON ERRORS AND DEBUG IDEAS

- check your x and y coordinates
- check your data type: float, uint8
- check your data range: [0,255], [0,1]
- perform simple checks (synthetic data examples)

Introduction

This assignment will deal with automatic searching for correspondence points between two images. Correspondences are a key step when dealing with the task of aligning two or more images, for example, building a panorama from multiple images. Methods that find correspondences between images usually start by finding feature points in them. Feature points denote regions in the image that have a high chance of re-detection in an image where the same scene is captured from a slightly different angle or viewing conditions.

Exercise 1: Feature points detectors

In this exercise, you will implement two frequently used feature point detectors: the Hessian algorithm [1](p. 44) and the Harris algorithm.

(a) The Hessian detector is based on the matrix of second derivatives $\mathbf{H}(x, y)$ (also named Hessian matrix, hence the name of the algorithm) at point (x, y) in the image:

$$\mathbf{H}(x, y) = \begin{bmatrix} I_{xx}(x, y; \sigma) & I_{xy}(x, y; \sigma) \\ I_{xy}(x, y; \sigma) & I_{yy}(x, y; \sigma) \end{bmatrix}, \tag{1}$$

where σ explicitly states that the second derivative is computed on a *smoothed image*. Hessian detector selects point (x, y) as a feature point if the determinant of the Hessian matrix exceeds a given threshold value t:

$$det(\mathbf{H}(x, y)) = I_{xx}(x, y; \sigma) I_{yy}(x, y; \sigma) - I_{xy}(x, y; \sigma)^2 > t. \tag{2}$$

Implement a function `hessian_points()`, that computes a Hessian determinant using the equation (2) for each pixel of the input image. As this computation can be very slow if done pixel by pixel, you have to implement it using vector operations (without explicit `for` loops). Test the function using image from `graf_a.jpg` as your input (do not forget to convert it to grayscale) and visualize the result.

Extend the function `hessian_points()` by implementing a non-maximum suppression post-processing step that only retains responses that are higher than all the neighborhood responses and whose value are higher than a given threshold value `thresh`. Try different box sizes for the neighborhood.

Create a function that you will use to plot the detected points (use `plt.plot()`) over the input image. Load the image `graf_a.jpg`, process it and visualize the result. Since the range of values of the determinant varies with different values of sigma, normalize the determinant values before applying the threshold. Experiment with different values of sigma and threshold (you are invited to implement a slider for each parameter for smoother visualization).

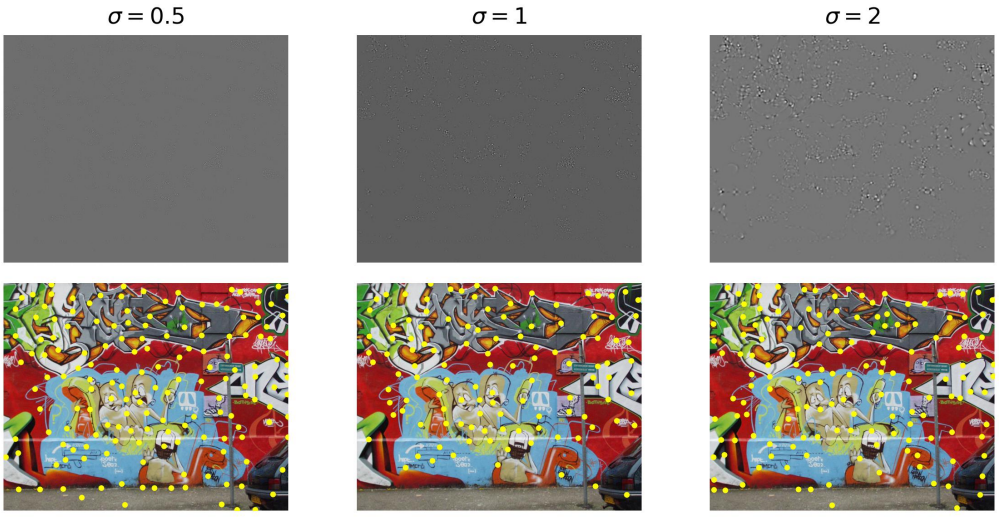

Figure 1: Raw Hessian determinant values for different values of σ.

Question: What kind of structures in the image are detected by the algorithm? How does the parameter σ affect the result?

(b) Implement the Harris feature point detector. This detector is based on the auto-correlation matrix $\mathbf{C}$ that measures the level of self-similarity for a pixel neighborhood for small deformations. At the lectures, you have been told that the Harris detector chooses a point (x, y) for a feature point if both eigenvalues of the auto-correlation matrix for that point are large. This means that the neighborhood of (x, y) contains two well-defined rectangular structures – i.e. a corner. Auto-correlation matrix can be computed using the first partial derivatives at (x, y) that are subsequently smoothed using a Gaussian filter:

$$\mathbf{C}(x, y; \sigma, \tilde{\sigma}) = \begin{bmatrix} (G(\tilde{\sigma}) * I_x^2(\sigma))[x, y] & (G(\tilde{\sigma}) * (I_x(\sigma) I_y(\sigma)))[x, y] \\ (G(\tilde{\sigma}) * (I_x(\sigma) I_y(\sigma)))[x, y] & (G(\tilde{\sigma}) * I_y^2(\sigma))[x, y] \end{bmatrix}, \quad (3)$$

where $*$ stands for convolution, σ is used for computing the image derivative as in the previous assignments, and $\tilde{\sigma}$ is used for subsequent smoothing of the derivative.

Computing the eigenvalues λ_1 and λ_2 of matrix $\mathbf{C}(x, y; \sigma, \tilde{\sigma})$ is expensive, therefore we will use the following relations[1]

$$\det(\mathbf{C}) = \lambda_1 \lambda_2 \quad (4)$$

$$\mathrm{trace}(\mathbf{C}) = \lambda_1 + \lambda_2 \quad (5)$$

to compute the ratio $r = \lambda_1 / \lambda_2$. If we assume that

$$\frac{\mathrm{trace}^2(\mathbf{C})}{\det \mathbf{C}} = \frac{(\lambda_1 + \lambda_2)^2}{\lambda_1 \lambda_2} = \frac{(r\lambda_2 + \lambda_2)^2}{r\lambda_2 \lambda_2} = \frac{(r+1)^2}{r}, \quad (6)$$

we can express the feature point condition for (x, y) as:

$$\det(\mathbf{C}) - \alpha \mathrm{trace}^2(\mathbf{C}) > t. \quad (7)$$

In practice we use $\tilde{\sigma} = 1.6\sigma$, $\alpha = 0.06$ for parameter values. Implement the condition (7) for all pixels without `for` loops, as you did for the core part of the Hessian detector. Perform non-maximum suppression post-processing step as well as thresholding using threshold `thresh`.

Load the image `graf_a.jpg` and compute the Harris feature points. Compare the result with the feature points detected by the Hessian detector. Experiment with different parameter values. Do the feature points of both detectors appear on the same structures in the image?

[1] In the following text we will omit the parameters of the auto-correlation matrix $\mathbf{C}(x, y; \sigma, \tilde{\sigma})$ for clarity and write $\mathbf{C}$ instead.

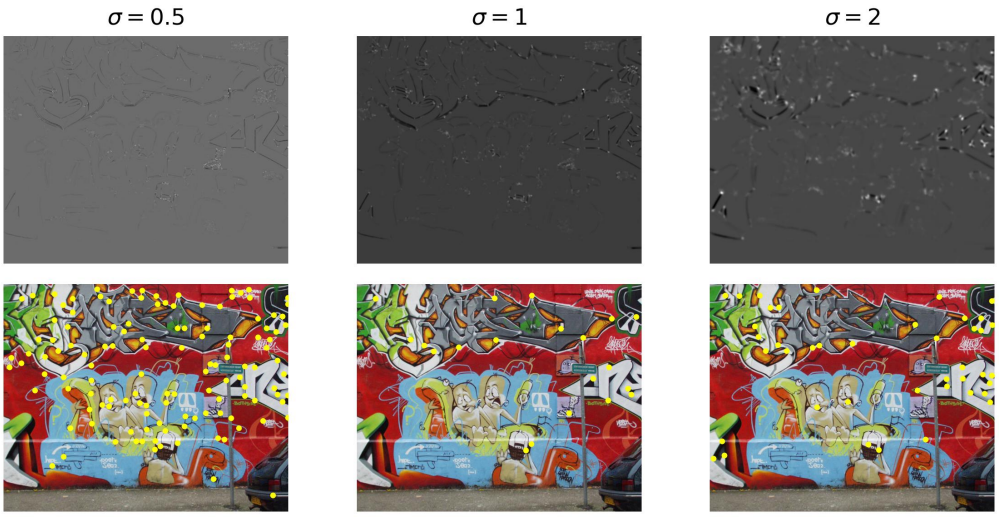

Figure 2: Raw Harris values for different values of σ.

Exercise 2: Matching local regions

One of the uses of feature points is searching for *similar structures* in different images. To do this, we will need descriptors of the regions around these points. In this assignment you will implement some simple descriptors as well as their matching.

(a) Use the function `simple_descriptors()` from `a4_utils.py` to calculate descriptors for a list of feature points. Then, write a function `find_correspondences()` which calculates similarities between all descriptors in two given lists. Use Hellinger distance (see Assignment 2). Finally, for each descriptor from the first list, find the most similar descriptor from the second list. Return a list of $[a, b]$ pairs, where a is the index from the first list, and b is the index from the second list.

Write a script that loads images `graf/graf_a_small.jpg` and `graf/graf_b_small.jpg`, runs the function `find_correspondences()` and visualizes the result. Use the function `display_matches()` from the supplementary material for visualization. Experiment with different parameters for descriptor calculation and report on the changes that occur.

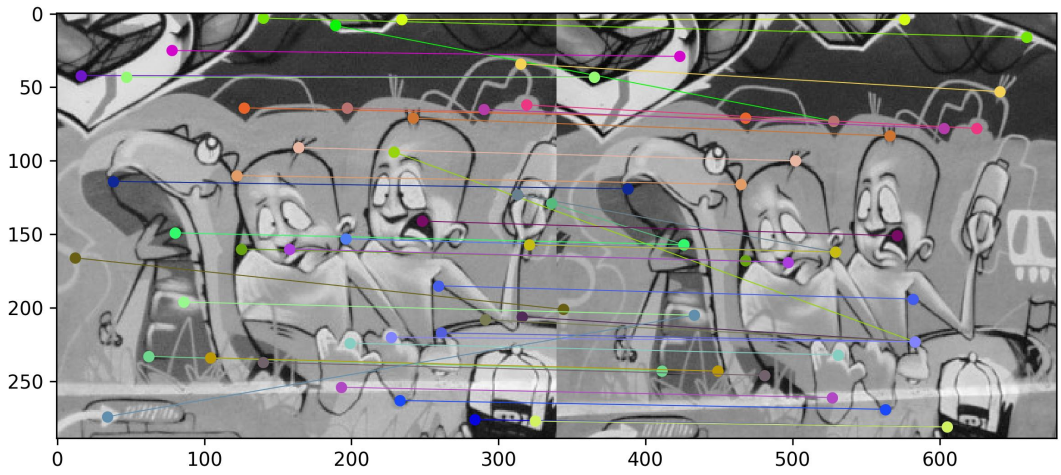

Figure 3: Matching points between images `graf_a_small.jpg` and `graf_b_small.jpg`. The parameters used for `simple_descriptor()` were `n_bins`=16 and `window_size`=20.

(b) Implement a simple feature point matching algorithm. Write a function `find_matches()` that is given two images as an input and returns a list of matched feature points from image 1 to image 2. The function should return a list of index pairs, where the first element is the index for feature points from the first image and the second element is the index for feature points from the second image.

Follow the algorithm below:

- Execute a feature point detector to get stable points for both images (you can experiment with both presented detectors),
- Compute simple descriptors for all detected feature points
- Find best matches between descriptors in left and right images using the Hellinger distance, i.e. compute the best matches from the left to right image and then the other way around. In a post-processing step, only select *symmetric* matches. A symmetric match is a match where a feature point P_1^i in the left image is matched to point P_2^j in the right image and at the same time point P_2^j in the right image is matched to the point P_1^i in the left image. This way we get a set of point pairs where each point from the left image is matched to exactly one point in the right image as well as the other way around.

Use the function `display_matches()` from the supplementary material to display all the symmetric matches. Write a script that loads images `graf/graf_a_small.jpg` and `graf/graf_b_small.jpg`, runs the function `find_matches()` and visualizes the result.

Question: What do you notice when visualizing the correspondences? How accurate are the matches?

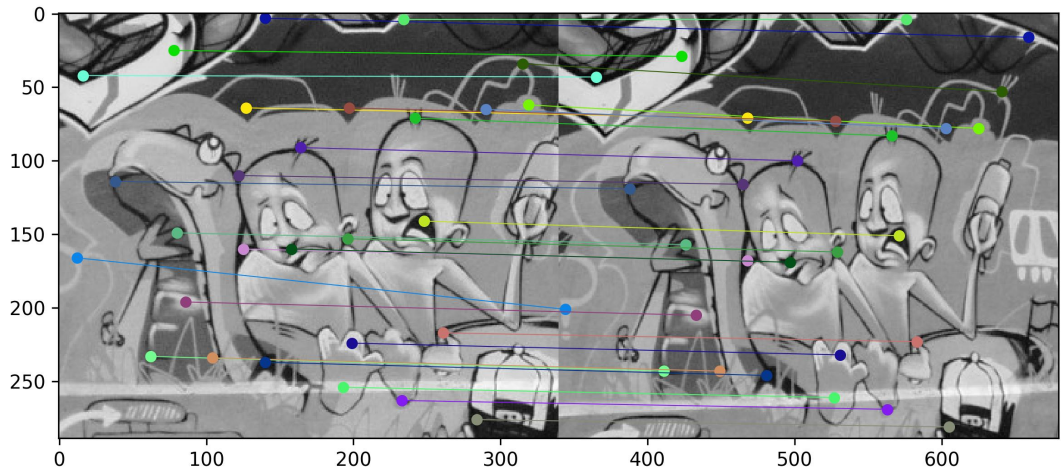

Figure 4: Matching points between images `graf_a_small.jpg` and `graf_b_small.jpg` when using symmetric matching.

(c) ★ (25 points) Implement a local feature descriptor of your choice (e.g. SIFT, SURF, BRIEF, HOG). Test it on other assignments and report on the changes (increased robustness etc.).

Note: If choosing to implement SIFT, implement only the descriptor part, not the DoG keypoint localization.

(d) ★ (15 points) Implement image stabilization using keypoints. First, you will need to record your own video to use as input. The video should include some camera translation, some rotation along the camera's optical axis (roll) and some forward/backward movement relative to the scene (which should be static). Then, you will detect keypoints in each frame of the video and match them to the keypoints detected in the first frame of the video. Finally, you will calculate the affine transformation matrix for each frame (using `cv2.estimateAffine2D` or `cv2.estimateAffinePartial2D`) and apply it to the corresponding image (using `cv2.warpPerspective`). The result should be a stabilized video. You can use the keypoint detection and matching you developed in this task, or use something like `cv2.SIFT` and `cv2.BFMatcher` from OpenCV.

Note: Please do not develop your solution on full-resolution videos, downsample them to a reasonable size.

Exercise 3: Homography estimation

In this assignment we are dealing with planar images, therefore we can try and estimate a homography matrix $\mathbf{H}$ that maps one image to another using planar correspondences. You will implement an algorithm that computes such a transformation using the minimization of the mean square error. For additional information about the method, consult the lecture notes as well as the course literature [1] (in the literature, the term *direct linear transform (DLT)* is frequently used to describe the idea).

We will start with a short overview of the minimization of mean square error on a simpler case of *similarity transform* estimation. The similarity transform is a linear transform that accounts for translation, rotation, and scale. The transformation of a point $\mathbf{x}_r$ can be written as $\mathbf{x}_t = f(\mathbf{x}_r, \mathbf{p})$, where $\mathbf{p} = [p_1, p_2, p_3, p_4]$ is a vector of four parameters that define the transform such that

$$\mathbf{x}_t = \begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} x_r p_1 - y_r p_2 + p_3 \\ x_r p_2 + y_r p_1 + p_4 \end{bmatrix}.$$

Question: Looking at the equation above, which parameters account for translation and which for rotation and scale?

Question: Write down a sketch of an algorithm to determine similarity transform from a set of point correspondences $P = [(\mathbf{x}_{r1}, \mathbf{x}_{t1}), (\mathbf{x}_{r2}, \mathbf{x}_{t2}), \dots (\mathbf{x}_{rn}, \mathbf{x}_{tn})]$. For more details, consult the lecture notes.

You will now implement a similar, but more complex algorithm for homography estimation. For a reference point $\mathbf{x}_r$ in the first image we compute a corresponding point $\mathbf{x}_t$ in the second image as:

$$\mathbf{H}\mathbf{x}_r = \mathbf{x}'_t \tag{8}$$

$$\begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & 1 \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ 1 \end{bmatrix} = \begin{bmatrix} x'_t \\ y'_t \\ z'_t \end{bmatrix}, \text{ and } \mathbf{x}_t = \begin{bmatrix} x_t \\ y_t \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x'_t}{z'_t} \\ \frac{y'_t}{z'_t} \\ 1 \end{bmatrix},$$

where points $\mathbf{x}_r$ and $\mathbf{x}_t$ are written in *homogeneous coordinates*[2], x_r and y_r correspond to location in the first image, and x_t and y_t to the second image. Using the equations (8) we get a system of linear equations with eight unknowns $h_{1,1}, h_{1,2}, h_{1,3}, h_{2,1}, h_{2,2}, h_{2,3}, h_{3,1}, h_{3,2}$:

$$\frac{h_{1,1}x_r + h_{1,2}y_r + h_{1,3}}{h_{3,1}x_r + h_{3,2}y_r + 1} = x_t \tag{9}$$

$$\frac{h_{2,1}x_r + h_{2,2}y_r + h_{2,3}}{h_{3,1}x_r + h_{3,2}y_r + 1} = y_t, \tag{10}$$

that can be transformed to

$$h_{1,1}x_r + h_{1,2}y_r + h_{1,3} - x_t h_{3,1}x_r - x_t h_{3,2}y_r - x_t = 0 \tag{11}$$

$$h_{2,1}x_r + h_{2,2}y_r + h_{2,3} - y_t h_{3,1}x_r - y_t h_{3,2}y_r - y_t = 0. \tag{12}$$

If we want to estimate the eight parameters that determine a homography, we need at least four pairs of matched feature points. As some matches can be imprecise, we can increase the accuracy of our estimate by using a larger number of matches $(\mathbf{x}_{r1}, \mathbf{x}_{t1}), \dots, (\mathbf{x}_{rn}, \mathbf{x}_{tn})$. This way we get an *overdetermined system* of equations:

[2]Homogeneous coordinates for a point in 2D space are obtained by adding a third coordinate and setting it to 1.

$$\mathbf{A}\mathbf{h} = \mathbf{0} \tag{13}$$

$$\begin{bmatrix} x_{r1} & y_{r1} & 1 & 0 & 0 & 0 & -x_{t1}x_{r1} & -x_{t1}y_{r1} & -x_{t1} \\ 0 & 0 & 0 & x_{r1} & y_{r1} & 1 & -y_{t1}x_{r1} & -y_{t1}y_{r1} & -y_{t1} \\ x_{r2} & y_{r2} & 1 & 0 & 0 & 0 & -x_{t2}x_{r2} & -x_{t2}y_{r2} & -x_{t2} \\ 0 & 0 & 0 & x_{r2} & y_{r2} & 1 & -y_{t2}x_{r2} & -y_{t2}y_{r2} & -y_{t2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{rn} & y_{rn} & 1 & 0 & 0 & 0 & -x_{tn}x_{rn} & -x_{tn}y_{rn} & -x_{tn} \\ 0 & 0 & 0 & x_{rn} & y_{rn} & 1 & -y_{tn}x_{rn} & -y_{tn}y_{rn} & -y_{tn} \end{bmatrix} \begin{bmatrix} h_{1,1} \\ h_{1,2} \\ h_{1,3} \\ h_{2,1} \\ h_{2,2} \\ h_{2,3} \\ h_{3,1} \\ h_{3,2} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \tag{14}$$

that can be (similarly to estimation of similarity transform) solved as a minimization of mean square error. If the matrix $\mathbf{A}$ is a square matrix, then we get an exact solution of the system. In case of an over-determined system (e.g., $n > 4$) the matrix $\mathbf{A}$ is not square. This problem is usually [1] solved using a matrix pseudo-inverse $\mathbf{A}^T\mathbf{A}$, that is square and can be therefore be split to eigenvectors and eigenvalues. A solution of such a system is the unit eigenvector of $\mathbf{A}^T\mathbf{A}$ that corresponds to the lowest eigenvalue. The same solution can be obtained more efficiently using the singular-value decomposition (SVD) as an eigenvector that corresponds to the lowest eigenvalue of $\mathbf{A}$ (using function `np.linalg.svd()`).

$$\mathbf{A} \overset{svd}{=} \mathbf{U}\mathbf{D}\mathbf{V}^T = \mathbf{U} \begin{bmatrix} \lambda_{1,1} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_{9,9} \end{bmatrix} \begin{bmatrix} v_{1,1} & \dots & v_{1,9} \\ \vdots & \ddots & \vdots \\ v_{9,1} & \cdots & v_{9,9} \end{bmatrix}^T \tag{15}$$

A vector $\mathbf{h}$ that contains the parameters of the homography matrix is obtained from the last column of matrix $\mathbf{V}$ and by normalizing the vector with the value of $v_{9,9}$ to set the last element in $\mathbf{h}$ to 1:

$$\mathbf{h} = \frac{[v_{1,9}, \dots, v_{9,9}]^T}{v_{9,9}}. \tag{16}$$

(a) Write a function `estimate_homography()`, that approximates a homography between two images using a given set of matched feature points, following the algorithm below.

- Construct matrix $\mathbf{A}$ using the equation (14).
- Perform a matrix decomposition using the SVD algorithm: `U, S, VT = np.linalg.svd(A)`.
- Compute vector $\mathbf{h}$ using equation (16).
- Reorder the elements of $\mathbf{h}$ to a 3×3 matrix $\mathbf{H}$ (e.g. using the function `np.reshape()`).

Load the two New York cityscape images (inside `newyork` folder), and four hand-annotated correspondence pairs in the file `newyork.txt`. You can load the data with `np.loadtxt()`. The columns are formatted as follows: x_1, y_1, x_2, y_2, i.e. the first column contains the x-coordinates of the points for the first image etc. Use the function `display_matches()` to display the pairs of points. Using these pairs,

estimate the homography matrix **H** from the first image to the second image and use function `cv2.warpPerspective()` to transform the first image to the plane of the second image using the given homography, then display the result. Also test your algorithm with images `graf/graf_a.jpg`, `graf/graf_b.jpg` and points from `graf.txt`.

Note: You can check the correctness of your implementation by comparing your result with the data in file `newyork/H.txt`. The numbers might not be absolutely identical, but should not differ from the reference before the third decimal place.

Note: For easier visualization, you can overlay images using `plt.imshow()` argument `alpha`. You can first plot an image normally using `plt.imshow(first_image)`, then overlay a semi-transparent image like this: `plt.imshow(second_image, alpha=0.5)`.

RANSAC algorithm

In practice, automatic correspondence detection algorithms very rarely find perfect matches. In most cases the locations of these points contain some noise, or in some cases several correspondences might also be completely wrong. These correspondences are called *outliers*[3]. A good meta-algorithm for such cases is RANdom SAmple Consensus (RANSAC). The algorithm can be applied to a wide variety of problems.

A RANSAC variant that robustly estimates a homography can be structured as follows:

- Randomly select a minimal set of matches that are required to estimate a model (that means 4 matches for homography).
- Estimate the homography matrix.
- Determine the *inliers* for the estimated homography (i.e. matched pairs with the reprojection error below a given threshold).
- If the percentage of inliers if large enough, use the entire inlier subset to estimate a new homography matrix.
- If the error of the newly computed homography is lower than any before, save the inlier set and the corresponding homography matrix.
- Iterate for k iterations.

The value of parameter k is defined by the properties of our data that are set by knowing the estimation problem that we are trying to solve. E.g., suppose that we consistently encounter at least w percent of inliers (correctly matched point pairs). The number of pairs required to estimate a homography matrix **H** is at least $n = 4$. We can now compute the probability of successful estimation of **H**, i.e. the probability that all n selected points will be inliers as w^n. The probability that this will not be true is $1 - w^n$. The probability that we do not manage to select a clean set of inliers in k repetitions is $p_{\text{fail}} = (1 - w^n)^k$. In practice, we therefore select k high enough to reduce the probability of failure p_{fail} to an acceptable level.

[3] You have probably noticed some outliers in the previous exercise when computing a homography.

(b) ★ (10 points) Implement 2d line fitting with RANSAC. To ease you into the idea of fitting a model to noisy data, you can look at the function `line_fitting()` in the supplementary material. It displays a noisy set of 2d points lying roughly on a line. Pairs of points are then selected in a loop, and the line determined by the selected points is drawn. This simulates the process of randomly selecting the minimal number of samples required for model fitting. This example can be easily modified into a proper RANSAC line fitting implementation just by determining the inliers for each selected pair of points. This can be done by calculating the distance of each point to the currently selected line. Display every step of the algorithm visually: show the currently selected line, the inliers, and outliers with different colors, and the best solution found thus far.

(c) Using the `find_matches` function that you have implemented in the previous exercise, find a set of matched points from either the `graf` or `newyork` image pairs. Then, implement the RANSAC algorithm for robust estimation of the homography matrix. For that, you will need the reprojection error for each of the proposed solutions. The reprojection error for each point can be calculated by multiplying the point's coordinates with the homography matrix and comparing the result to the reference point from the other image. You can use Euclidean distance for that. The final reprojection error for a solution should then be the average of reprojection errors for all included points.

Find a subset of points that produce a high quality homography estimation and use that matrix H to transform one image to the other (you can again use `cv2.warpPerspective()`).

Question: How many iterations on average did you need to find a good solution? How does the parameter choice for both the keypoint detector and RANSAC itself influence the performance (both quality and speed)?

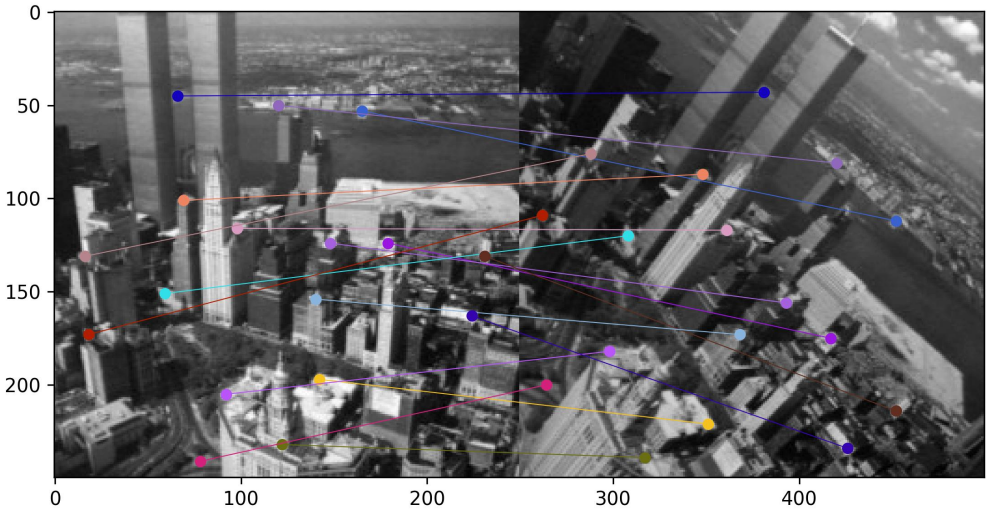

Figure 5: RANSAC-filtered matches for `newyork` image pair.

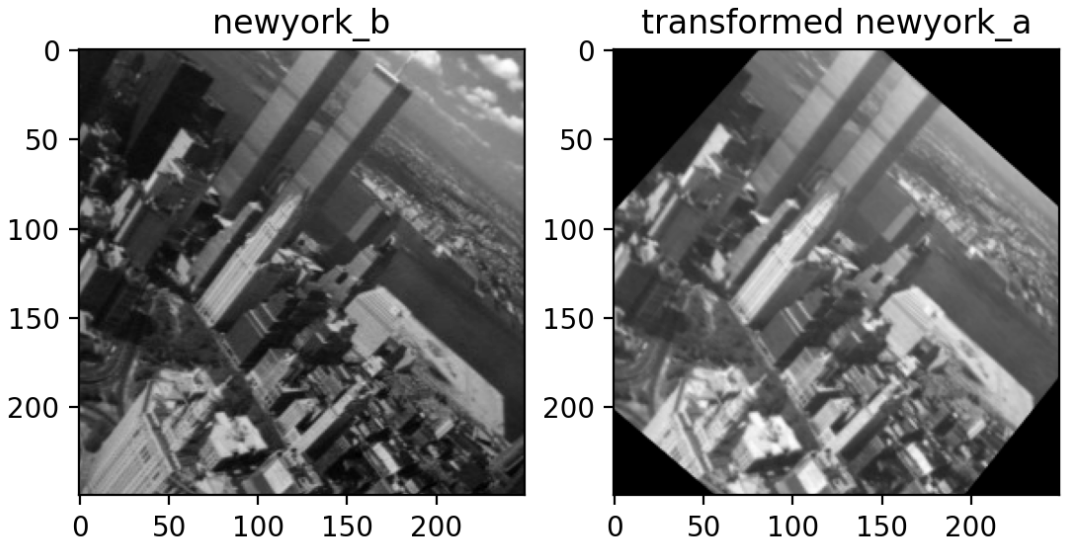

Figure 6: Image `newyork_a` transformed with the RANSAC-estimated homography matrix.

(d) ★ (5 points) Calculate the number of expected iterations for RANSAC using the formula mentioned in the instructions. Estimate the missing values (i.e. the inlier probability) using the example images used in this assignment. Propose and implement a method that will try to stop the algorithm as soon as a good enough solution is found.

(e) ★ (5 points) Write your own function for mapping points using the homography matrix. It should work like OpenCV's `warpPerspective()`. It should accept an image and a homography matrix and return the input image as remapped by the homography matrix. The size of the output should match the size of the input image.

Hint: You will need to use homogeneous coordinates.

Note: The mapping should be without holes or artifacts. Additional post-processing (or interpolation) is not allowed. The correct solution is achievable using only homography mapping.

References

[1] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach.* Prentice Hall, 2002.