Computer Vision I
Assignment 3

Prof. Stefan Roth
Krishnakant Singh
Shweta Mahajan

TECHNISCHE UNIVERSITÄT DARMSTADT

06/12/2021

This assignment is due on December 20th, 2021 at 23:59.

Please refer to the previous assignments for general instructions and follow the handin process described there.

Problem 1: Hessian Detector (10 Points)

In this problem we will take a look at interest point detection, e.g. as shown below:

Figure 1: Interest points.

One of the earliest interest point detectors was the Hessian detector which identifies corner-like structures by searching for points $\boldsymbol{p} = (x, y)^T$ with a strong Hessian determinant $\det(\boldsymbol{H})$. We use the Hessian matrix $\boldsymbol{H}$ that is calculated on the image smoothed by a Gaussian filter with kernel width σ, i.e.

$$\boldsymbol{H}(\sigma) = \begin{bmatrix} I_{xx}(\sigma) & I_{xy}(\sigma) \\ I_{xy}(\sigma) & I_{yy}(\sigma) \end{bmatrix} \tag{1}$$

with $I_{xx}(\sigma)$, $I_{xy}(\sigma)$ and $I_{yy}(\sigma)$ denoting the partial (horizontal and vertical) second derivatives of the smoothed image I. The interest points are defined as those points whose Hessian determinant is larger than a certain threshold t, i.e.

$$\sigma^4 \cdot \det\left(\boldsymbol{H}\right) > t. \tag{2}$$

Note that we include an additional scale normalization factor σ^4 so that we can use the same threshold t independently of the value of σ. For the following tasks please use the functions `gauss2d` and `derivative_filters` given in `problem1.py` to generate a Gaussian smoothing filter with size 25×25 and $\sigma = 5$ and use the central difference filters to compute the derivatives. For color images you only need to detect the interest points in the gray-scale space (`load_img` returns color and gray-scale images).

The code outline is given in `problem1.py` which should be completed with the necessary functions:

- Function `compute_hessian` to obtain the required components of the Hessian $\boldsymbol{H}$ with the given Gaussian and derivative filters. Use *mirror* boundary conditions for any filtering involved.

  (3 points)

- Function `compute_criterion` that computes the scaled Hessian determinant given by the left-hand side of (2).

  (3 points)

- Function `nonmaxsuppression` that applies non-maximum suppression to the computed criterion in order to extract local maxima, *i.e.*, points for which function values are the largest within their surrounding 5×5 windows, respectively. Allow multiple equal maxima in one window and throw away all interest points in a 5 pixel boundary at the image edges. After that find all local maxima with a function value that is larger than the threshold $t = 1.5 \cdot 10^{-3}$. Note: To implement `nonmaxsuppression` the function `maximum_filter` from `scipy.ndimage` might be handy. Figure 1 shows an example of how an interest point visualization can look like.

  (4 points)

Problem 2: Image Stitching (25 Points)

Image alignment has a wide range of applications in visual tasks, such as panorama stitching and scene reconstruction. As shown in Fig. 2, in this task we will stitch two images. To achieve that, we will use RANSAC to estimate the homography matrix based on SIFT point correspondences in both images. We provide the function definitions you will need to implement in `problem2.py`.

Figure 2: Stitched images

Tasks:

We have already precomputed a set of interest points using the Harris interest point detector as well as SIFT features for all interest points, which will be loaded in the beginning of the script.

- For finding putative matches between the keypoints in both images, we have to define a distance function for corresponding feature vectors. A simple distance measure is given by the (squared) Euclidean distance which is defined as

$$d^2(p, q) = \sum_i (q_i - p_i)^2 \quad i = 1 \dots D,$$

where p, q are given feature vectors with dimension D. Implement `euclidean_square_dist` which returns the pairwise squared Euclidean distance for two sets of keypoints.

(2 points)

- Next, implement the function `find_matches` that takes two sets of keypoints as well as the pairwise distance matrix and computes for each point in the smaller set the nearest neighbor in the larger set.

(2 points)

You will now implement the RANSAC algorithm:

- First, implement the function `ransac_iters` that returns an estimate of the required number of iterations n for RANSAC. The estimate accounts for three variables: the probability of a point correspondence being an inlier, p, the number of correspondences we pick each time k, and the probability z of having at least one sample with no outlier after n iterations.

(1 point)

- Next, implement the function `pick_samples` that randomly picks k points from given keypoints. For our problem we draw $k = 4$ correspondences per sample.

(2 points)

- In order to improve numeric stability, the function `condition_points` receives a set of cartesian point coordinates and returns the conditioned points in homogeneous coordinates as well as the conditioning matrix T.

(3 points)

- Implement the function `compute_homography` that estimates the homography from the given conditioned coordinates of correspondence points. You may use `np.linalg.svd` for this function. Return the homography matrix with respect to the unconditioned coordinates as well as the homography matrix regarding the conditioned coordinates. Normalize these matrices so that the bottom right value equals 1.

(4 points)

- Next, implement the function `transform_pts` which computes the transformation by the homography matrix for an array of points.

(2 points)

- To find inliers, first implement the function `compute_homography_distance` that evaluates a homography w.r.t. the putative correspondences. It computes for each point the symmetric (squared) distance to the corresponding point transformed by the homography, i.e.

$$d^2(H, x_1, x_2) = \|Hx_1 - x_2\|^2 + \|x_1 - H^{-1}x_2\|^2.$$

  Based on these distances and a threshold `find_inliers` should return the number of inliers as well as the inliers themselves.

(2 + 1 points)

- Using these functions, now implement the RANSAC algorithm in `ransac`. First, randomly draw a sample of k corresponding point pairs and estimate the corresponding homography using your homography function. Then, evaluate the homography by means of the homography distance specified above. For each iteration you have to determine the number of inliers for the estimated homography. The `ransac` function should return the best homography based on the number of inliers, the maximum number of inliers found and the inliers themselves.

(4 points)

- As a final step, reestimate the homography matrix based on the inliers with the function `recompute_homography`.

(2 points)

  Based on your estimate of the homography matrix the function `stitch_images` aligns the second image with the first image and stitches them together, similar to Fig. 2.