Assignment 3 – Computer Vision 1 – WS19/20 25/11/2019

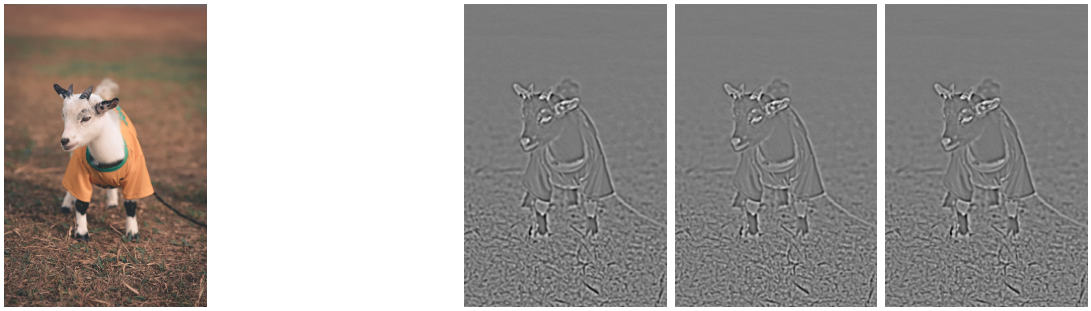
Prof. Stefan Roth
Xiang Chen
Faraz Saeedan
Nikita Araslanov

This assignment is due on December 13th, 2019 at 00:00.

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

Problem 1 - Laplacian of Gaussian (LoG) Detector (20 points)

Scale-invariant interest point detectors are robust to changes in scale of input images, which is necessary when matching between pairs of images with different zoom levels. *Laplacian of Gaussian (LoG)* is an example of such detectors whose popularity is in part due to its simplicity.



(a) (b)

Figure 1: **(a)** Input image. **(b)** Filter responses (normalised) from the three methods below with $\sigma = 1$

In this problem we will implement various practical versions of this interest point detector and apply it to the image depicted in Fig. 1.

Tasks:

Laplacian of a smoothed image. Recall from the lecture that the finite difference Laplacian operator is formulated as:

$$\nabla^2 I = I_{xx} + I_{yy}, \tag{1}$$

where I is the input image. The Laplacian operator in Eq. 1 is sensitive to image noise. To reduce the effect of the noise, a Gaussian smoothing is applied to the image (or the response of the Laplacian operator). The input image is `goat.jpeg` provided in `data/`. The scales denoted by σ's, is a list that range from 0.25 to 4.0 in increments of 0.25.

- Implement `load_image` to load an image, convert it to greyscale for simplicity. Normalise the values to range $[-1, 1]$ and return it as a numpy array.

  [1 point]

- Discretize Eq. 1 and set `laplacian_kernel` to return a 3×3 kernel approximating the Laplacian operator. Make sure that the resulting kernel accounts only for the 4-neighborhood, *i.e.* the top, bottom, left and right neighbors.

  [2 points]

- Implement function `smoothed_laplacian` which uses the input image, a list of σ's, applies Eq. 1 on the image and smoothes the result with a 7×7 Gaussian filter with the corresponding σ.

  [3 points]

- Implement function `blob_detector` that finds *unique* local extrema (maxima or minima) in a 9×9 spatial neighborhood and across the whole scale dimension (*i.e.* $15 \times 9 \times 9$ local window). Collect only 0.1% percentile of the points w.r.t. the global minimum and maximum of the values in the response. *Hint:* You may find function `np.percentile` useful.

  [3 points]

LoG as a single operator. As an alternative approach, one can calculate a single kernel for LoG operator by using the second derivative of a Gaussian and discretizing the result. This new kernel can then be convolved by the input image to yield the output in one stage.

- Analytically calculate the second derivative of a Gaussian filter and then complete `LoG_kernel` function to return an arbitrary-sized kernel representing it.

  [3 points]

- Implement function `laplacian_of_gaussian` which uses the image and a list of σ's, and applies the 9×9 discretized version of LoG to the input image. The scales are the same as in the previous task.

  [3 points]

- *Multiple Choice Question.* Apply function `blob_detector` from the first task to this output. The results should be similar to the first method in terms of areas with a high density of interest points, although the number of interest points itself may differ substantially. What are the advantages of using this new method? Please, complete method `answer` in class `Method`.

  [1 point]

Difference of Guassians. LoG is sometimes approximated by the difference of two Gaussians (DoG) at different scales. A good pair of candidates are $\sigma_1 = \sqrt{2}\sigma$ and $\sigma_2 = \frac{\sigma}{\sqrt{2}}$.

- Implement function `DoG` which uses the image and a list of σ's, and applies difference of Gaussians to the input image. Use 9×9 kernels and the same scales from the previous sections.

  [3 points]

- Apply function `blob_detector` from the first task to this output. The results should be reasonably close to the previous approach (also in terms of quantity).

  [1 point]

Hint: Ad-hoc strategies to padding the image before convolution (*i.e.* constant) can result in a significant number false positives. If you use `convolve2d` from `scipy.signal`, please, use `mode="same"` and `boundary="symm"` in your submission.

Submission: Please only include `problem1.py` in your submission.

Problem 2 - Image Alignment (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 align a rotated image, Image 1, with a reference image, Image 2. To achieve that, we will estimate the homography matrix using SIFT-based point correspondences in both images. We provide the function definitions you will need to implement in `problem2.py`. Please, complete this task using the course material and following the task description below.

Figure 2: **Task overview.** Align the rotated Image 1 w.r.t. Image 2 via homography estimation.

Tasks:

1. Implement function `load_pts_features` that loads a `numpy` array saved in the provided `pts_feats.npz`. The file contains coordinate points and SIFT feature descriptors accessible via keys `pts` and `feats`, respectively. The first element of the key value is the data array for Image 1, and the second is the data for Image 2.
[1 point]

2. What is the minimum number of point correspondences N_{min} we need to estimate the homography? Implement function `min_num_pairs` which returns N_{min}.
[1 point]

3. Implement function `pickup_samples` which selects N_{min} unique correspondences at random. The input to the function are the point coordinates from the two images. Here, we assume the points have been already aligned, *i.e.* the nth point from the first array corresponds to the nth point in the second array.
[1 point]

4. Implement function `compute_homography` that estimates the homography matrix from two point correspondences as a solution to a homogeneous linear equation system.
[4 points]

Recall from the lecture, that we use the number of inliers to judge about the quality of homography estimation. In more detail, we transform the points from the first image into the reference frame of the second image via homography and then count the number of "overlapping" interest points in both images.

5. To implement the first step of this process, the transformation, complete function `transform_pts`.
[3 points]

6. Next, implement function `count_inliers` that computes the number of inliers corresponding to the provided homography matrix. The function first transforms one point set to align with the second. It then counts one correspondence as an inlier, if the L2-distance between the corresponding points is less than `threshold`.
[3 points]

We have now implemented one round of homography estimation: given a randomly selected subset of point correspondences, we can estimate the homography matrix and evaluate it by counting the number of inliers. Eventually, we would like to find the homography matrix with the *maximum* number of inliers. To achieve that, the method should also be robust, since point correspondences may contain outliers. In the lecture, we discussed one approach – the RANSAC algorithm that implements this objective in a robust way.

8. First, implement function `ransac_iters` that returns an estimate of the required number of iterations k for RANSAC. The estimate accounts for three variables: the probability of a point correspondence being an inlier, w, the number of correspondences we pick each time N_{min}, and the probability z of having an outlier in k samples.

[2 points]

9. Next, complete one RANSAC iteration in function `ransac`. The function estimates the homography in k iterations, each with a random sample of N_{min} point correspondences. It keeps track of and returns the best homography matrix, in terms of the inlier number provided by `count_inliers`.

[4 points]

So far, we have assumed the point correspondences given, *i.e.* the point sets provided as the arguments to the functions above are "aligned". Obviously, in practice we first need to find these correspondences. We will do this now using the SIFT features.

10. Implement function `find_matches`, that returns *indices* of the matching points by measuring the L2-distance between the corresponding SIFT-descriptors. As a criterion for a valid match, we will use $d^* = d_{\text{SIFT}}/d'_{\text{SIFT}}$, where d_{SIFT} is the minimum distance for a point in the first set to the one in the second, and d'_{SIFT} is the *second* minimum distance found for the first point. We count the correspondence as valid, if d^* is less than the given threshold.

[3 points]

11. Re-estimate final homography in function `final_homography` using the correspondences you find with your implementation of `find_matches`.

[3 points]

Submission: Please include only `problem2.py` in your submission.