Assignment 3: Corners Detection - Image Stitching

Due Date: 03 November 2023, 23:59

Introduction

The current assignment is divided into two parts. In the first part, you are asked to implement the Harris Corner Detector and validate your implementation on some toy example images. For the second part, you will compose a panorama from five different images that illustrate the same 3D scene from slightly different viewpoints. In order to achieve this, you will need to implement the RANSAC algorithm. Finally, by making use of the estimated homographies, you need to stitch the images together thus forming a panoramic image.

Preliminary step

For all checks that are needed for the following questions, you should use the `assert()` function. All intermediate and final results should be plotted using the `matplotlib` package.

1 Harris Corner Detector (50 points)

In this question you are going to implement the Harris Corner Detector algorithm and validate your results on a number of example images. First, load all 6 images that are found under `data/corners` and plot them in a 2×3 grid. Then, create a function with the following function signature:

```
def detect_corners(input_image, max_corners=0, quality_level=0.01, min_distance=10, block_size=5, k=0.05):
    """
        Detect corners using Harris Corner Detector
    :param input_image: numpy.array(uint8 or float), input 8-bit or floating-point 32-bit, single-channel
                        image
    :param max_corners: int, maximum number of corners to return, if 0 then return all
    :param quality_level: float, parameter characterizing the minimal accepted quality of image corners
    :param min_distance: float, minimum possible Euclidean distance between the returned corners
    :param block_size: int, size of an average block for computing a derivative covariation matrix
                        over each pixel neighborhood.
    :param k: float, free parameter of the Harris detector
    :return:
        corners: numpy.array(uint8)), corner coordinates for each input image
    """
```

Your implementation should do the followings checks:

- `max_corners`: should be a positive number (zero included)
- `quality_level`: should lie within the range $[0, 1]$
- `min_distance`: should be a positive number (zero included)
- `block_size`: should be a non-zero odd positive number
- `k`: should lie within the range $[0.04, 0.06]$

After the aforementioned checks, first you need to calculate the first-order partial derivatives for the `input_image`. For this step you can use the `cv2.Sobel()` function. Then, calculate the structure matrix (covariation matrix) w.r.t. `block_size` for each pixel. The weights for the window function should be 1 (**not** Gaussian window). Based on this, calculate the R-score for each pixel and mark candidate corner pixels using the `quality_level` (ratio of the maximum R-score value). Then you need to force that the minimum possible Euclidean distance between the returned corners is equal to `min_distance`. In order to do this, first you need to sort, in descending order, the candidate corners w.r.t. their R-score. Next, you need to use the $k-d$ tree space partitioning data structure in order to suppress corners within a `radius=min_distance`. Finally, return corners left after the radius search (in descending order w.r.t. their R-score) w.r.t. `max_corners`.

Your results should be validated using OpenCV's function as follows:

```
quality_level, max_corners, min_distance, block_size, k = 0.01, 0, 10.0, 5, 0.05
cv2.goodFeaturesToTrack(image=<input_image>, maxCorners=max_corners, qualityLevel=quality_level,
                        minDistance=min_distance, blockSize=block_size, useHarrisDetector=1, k=k)
```

For your implementation you should use the same input parameters. For each image mark detected corners using `cv2.circle()`. Finally, plot the results from OpenCV and from your implementation in two 2×3 grids.

2 Image Stitching (50 points)

For this question you are going to stitch together the provided images and form a panoramic image. First, load all 5 images that are found under `data/panoramas` and plot them in a 1×5 grid. Then you need to compute SIFT features using OpenCV's implementation on the grayscale images and find the 2-nearest neighbors w.r.t. SIFT features between pairs of images (not all pairs of images have overlapping areas). In order to find *good* correspondences between images you need to apply the ratio test (`ratio=0.75`). Plot the detected SIFT features in a 1×5 grid and the *good* correspondences in a 2×2 grid.

The next step is to remove outlier correspondences and estimate the homography transformation between pairs of images. In order to do this you need to create a function with the following function signature:

```
def ransac(src_points, dst_points, ransac_reproj_threshold=2, max_iters=500, inlier_ratio=0.8):
    """
        Calculate the set of inlier correspondences w.r.t. homography transformation, using the
        RANSAC method.
    :param src_points: numpy.array(float), coordinates of the points in the source image
    :param dst_points: numpy.array(float), coordinates of the points in the destination image
    :param ransac_reproj_threshold: float, maximum allowed reprojection error to treat a point pair
                                    as an inlier
    :param max_iters: int, the maximum number of RANSAC iterations
    :param inlier_ratio: float, ratio of inliers w.r.t. total number of correspondences
    :return:
        H: numpy.array(float), the estimated homography transformation using linear least-squares
        mask: numpy.array(uint8), mask that denotes the inlier correspondences
    """
```

Your implementation should do the followings checks:

- `src_points` and `dst_points` should have the same dimensions
- `ransac_reproj_threshold`: should be a positive number (zero included)
- `max_iters`: should be a non-zero positive number
- `inlier_ratio`: should lie within the range $[0, 1]$

This function will take as an input the *good* corresponding points and use the RANSAC algorithm to remove outliers (for estimating the homography transformation on the seed group you can use the `cv2.getPerspectiveTransform()`). After the algorithm converges, the final homography transformation should be estimated using the linear least-squares approach w.r.t. inliers (you can use the `numpy.linalg.pinv()` in order to compute the pseudo-inverse matrix). In order to validate your results you need to use OpenCV's function as follows:

```
ransac_reprojection_threshold, max_iters = 1.0, 1000
cv2.findHomography(srcPoints=<src_points>, dstPoints=<dst_points>, method=cv2.RANSAC,
                   ransacReprojThreshold=ransac_reprojection_threshold, maxIters=max_iters)
```

For your implementation you should use the same input parameters (use `inlier_ratio=0.8` for your implementation). Plot the inlier correspondences from OpenCV and from your implementation in two 2×2 grids.

Finally, stitch the images in two panoramas, one for OpenCV's homographies and one for the estimated homographies from your implementation. The panoramas should be trimmed, in order to remove any black borders caused by the stitching process. Plot both panoramas in a 2×1 grid.

3 Extra Credits: Cylindrical Warping (10 points)

For this **extra credits** question, after loading the images you need to warp them within a cylindrical coordinate system. Find the focal length of the image in their properties, and convert from *mm* to pixels. Also, set $(x_c, y_c) = (width/2, height/2)$. Plot the warped images in a 1×5 grid. Then run the whole process as described in Question 3, in order to create panoramas using the cylindrical warped images.

4 Instructions

- The assignment is due by **Friday November 3rd, at 23:59** (see the course's GitHub repository for more details regarding the late assignment policy).
- The assignment should be submitted **only** through Moodle. Compress all the needed **source code** files, e.g. all *.py files into a .zip file and name it as follows "**Lab_Assignment_3_<ID-Number>.zip**".
- For further questions you can either create an issue on the course's GitHub repository or contact me via email mloizo11@ucy.ac.cy.
- All lab assignments should be conducted **independently**.
- The Moss system will be used to detect code similarity.
- Plagiarism and/or cheating will be punished with a grade of zero for the current assignment. A second offense will carry additional penalties.