Assignment 3: Corners Detection - Image Stitching

Due Date: 08 November 2024, 23:59

Introduction

This assignment consists of two main tasks. In the first task, you will implement the Harris Corner Detector and test your implementation using a set of sample images. In the second task, you will create a panorama by combining five images of the same 3D scene, taken from slightly different viewpoints. To do this, you will implement the RANSAC algorithm to estimate the homographies between the images. Finally, you will use these homographies to seamlessly stitch the images together into a single panoramic image.

Preliminary step

For all checks required in the following questions, you should use the `assert()` function. Ensure that all intermediate and final results are plotted using the `matplotlib` package.

1 Harris Corner Detector (50 points)

In this question, you will implement the Harris Corner Detector algorithm and validate your results using several example images. First, load all six images from the `data/corners` directory and plot them in a 2×3 grid. Next, create a function with the following 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 perform the following checks:

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

After these checks, compute the first-order partial derivatives of the `input_image` using the `cv2.Sobel()` function. Then, calculate the structure (covariance) matrix for each pixel with respect to the `block_size`. The window function weights should be 1 (i.e., not a Gaussian window). Based on this, compute the R-score for each pixel, and identify candidate corner pixels based on the `quality_level`, which is a ratio of the maximum R-score.

Next, ensure that the minimum Euclidean distance between the detected corners is equal to `min_distance`. To achieve this, first, sort the candidate corners in descending order based on their R-score. Then, use the k-d tree data structure to suppress corners within a `radius = min_distance`. Finally, return the remaining corners (sorted by R-score in descending order) up to `max_corners`.

Validate your results using OpenCV's built-in function as follows:

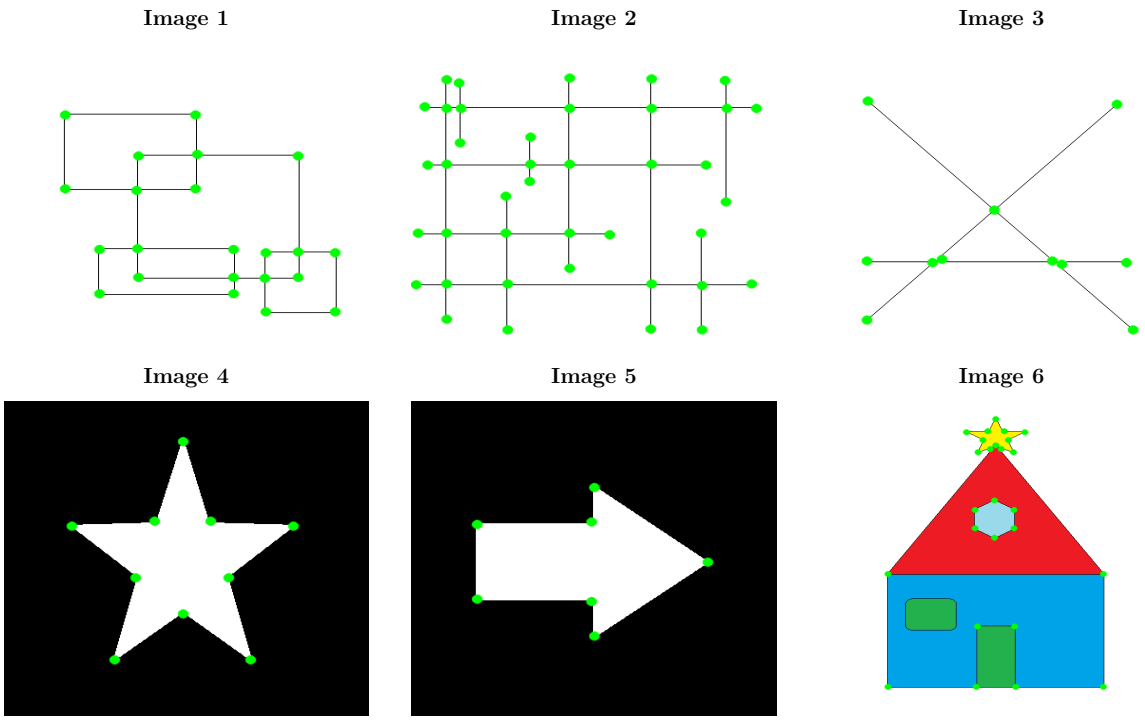

Figure 1: Corner detection results

```
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)
```

Use the same input parameters for your implementation. For each image, mark the detected corners using `cv2.circle()`. Finally, display the results from both OpenCV's function and your implementation in two separate 2×3 grids. Your results should be similar to those in Figure 1.

2 Image Stitching (50 points)

In this question, you will stitch together the provided images to create a panoramic image. First, load all five images from the `data/panoramas` directory and display them in a 1×5 grid. Next, convert the images to grayscale and compute SIFT features using OpenCV's implementation. For each pair of images, find the two nearest neighbors based on their SIFT features (note that not all image pairs will have overlapping areas). To identify *good* feature correspondences between images, apply the ratio test with `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 image pairs. To do this, you will need to implement a function with the following signature:

```
def ransac(src_points, dst_points, ransac_reproj_threshold=2, max_iters=500, inlier_ratio=0.8):
    """
        Estimate the homography transformation using the RANSAC algorithm, while
    identifying inlier correspondences.
    :param src_points: numpy.array(float), coordinates of points in the source image
    :param dst_points: numpy.array(float), coordinates of points in the destination
    image
    :param ransac_reproj_threshold: float, maximum reprojection error allowed to
    classify a point pair as an inlier
    :param max_iters: int, maximum number of RANSAC iterations
    :param inlier_ratio: float, the desired ratio of inliers to total correspondences
    return:
        H: numpy.array(float), the estimated homography matrix using linear
        least-squares
        mask: numpy.array(uint8), mask indicating the inlier correspondences
    """
```

Your implementation should perform the following checks:

Figure 2: Panorama

Figure 3: Panorama under cylindrical warping

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

This function takes the *good* corresponding points as input and applies the RANSAC algorithm to remove outliers. To estimate the homography transformation for the initial set of inliers, you can use `cv2.getPerspectiveTransform()`. After the RANSAC algorithm converges, the final homography should be calculated using the linear least-squares method, focusing on the inliers (you can use `numpy.linalg.pinv()` to compute the pseudo-inverse matrix).

To validate your results, you can 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, use the same input parameters, with `inlier_ratio=0.8`. Plot the inlier correspondences from both OpenCV's method and your implementation in two 2×2 grids.

Finally, stitch the images into two panoramas: one using OpenCV's homographies, and the other using the homographies from your implementation. Ensure that both panoramas are trimmed to remove any black borders from the stitching process. Display both panoramas in a 2×1 grid. Your results should be similar to those in Figure 2.

3 Extra Credits: Cylindrical Warping (10 points)

For this **extra credits** question, after loading the images, you will warp them onto a cylindrical coordinate system. First, find the focal length from the image properties and convert it from millimeters to pixels. Then, set the image center coordinates as $(x_c, y_c) = (\text{width}/2, \text{height}/2)$. Display the warped images in a 1×5 grid. Next, follow the entire process described in Question 2 to create panoramas using the cylindrical-warped images. Your results should be similar to those in Figure 3.

4 Instructions

- The assignment is due by **Friday November 8th, 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_<UC-ID-Number>.zip**".
- For further questions you can create an issue on the course's GitHub repository.
- 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.