Feature extraction/optical flow lab report

Ambroise

October 7, 2023

1 extract_harris

1. Compute image gradients in x and y direction : The formula for the gradient in the x direction is Ix=1/2*(I[x+1,y] - I[x-1,y]). We use convolution with kernels for the x and y directions and maintain the same image size. The x axis is considered horizontal so the kernel for the x-direction is $\begin{bmatrix}0.5 & 0 & -0.5\end{bmatrix}$, while the kernel for the y-direction, which is considered vertical, is $\begin{bmatrix}0.5\\0\\-0.5\end{bmatrix}$. We apply these kernels using `signal.convolve2d`. The image is padded with 0, which is an arbitrary value (it does not have a big influence on since border corners are later filtered when computing descriptors).

2. Blur the computed gradients : We use Gaussian blur with `cv2.GaussianBlur` to smooth the gradient images (equivalent to applying the window function in the course and summing over the window) This smoothing helps reduce noise and prepares the gradients for further processing. The `borderType=cv2.BORDER_REPLICATE` parameter ensures that the image borders are handled correctly.

3. Compute elements of the local auto-correlation matrix "M" : We compute M_{11}, M_{22}, and M_{12} using the smoothed gradient images. Note that M_{21} is equal to M_{12} since the matrix is symmetric.

4. Compute Harris response function C : We use the elements of the auto-correlation matrix to calculate the Harris response. The response is defined as $C = \det(M) - k \cdot \text{trace}(M)^2$.

5. Detection with threshold and non-maximum suppression : We apply a threshold to the Harris response to identify potential corner locations. This creates a first mask with True values for potential corners. For non-maximum suppression, we use to `scipy.ndimage.maximum_filter` to check a 3x3 neighborhood, then use `np.where` to only keep the potential corners whose value hasn't changed after maximum_filter (it is therefore the local maximums). This produces a second mask. We combine both masks and extract coordinates of the filtered corner, considering the top-left corner as the origin, with the x-axis being horizontal and the y-axis being vertical. Finally, we stack the coordinates into the correct output format and return the detected corners as well as the computed Harris responses for the entire image.

The images below show that this method is quite effective as corners are globally well spotted. Although some are sometimes missing and some detected corners are questionable, the method with the associated parameters seems reliable

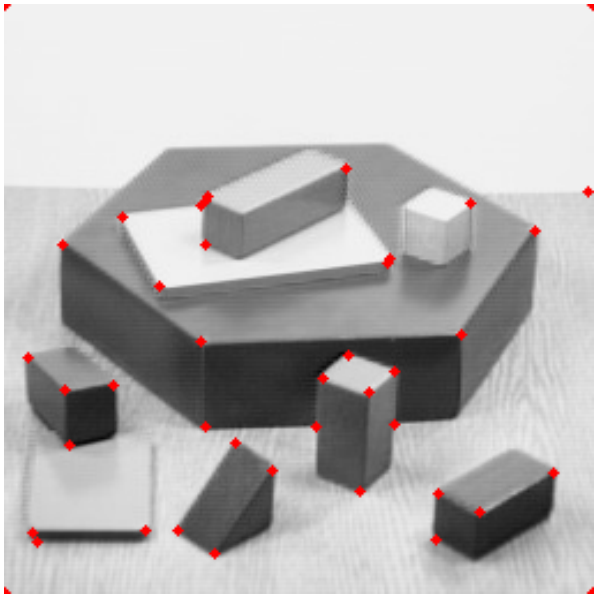

Figure 1: Harris corner detection on blocks (the constants used are the ones on top of main.py)

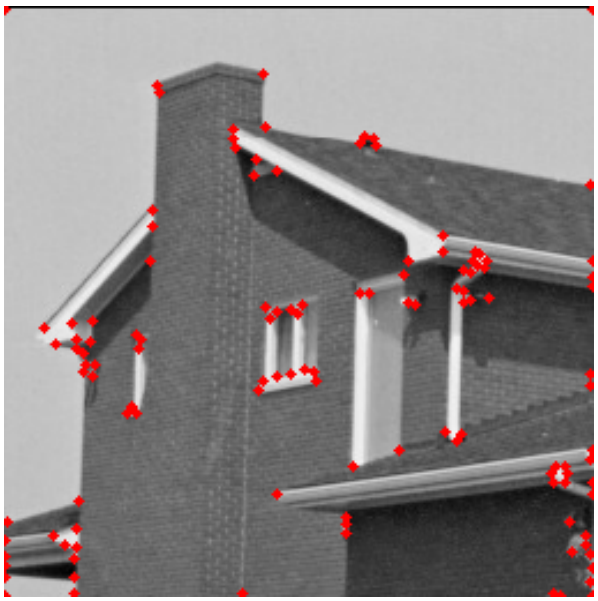

Figure 2: Harris corner detection on house (the constants used are the ones on top of main.py)

Figure 3: Harris corner detection on I1(the constants used are the ones on top of main.py)

Figure 4: Harris corner detection on I2 (the constants used are the ones on top of main.py)

2 filter_keypoints

The goal is to filter out keypoints that are too close to the image edges to ensure that patches centered around these keypoints remain within the image boundaries.

The `patch_size` parameter determines the size of the patches, with each keypoint as its center (default is 9). An offset is calculated based on half the patch size to determine how far keypoints should be from the image edges.

A mask is created to retain only the keypoints that satisfy the following conditions:

- The x-coordinate of the keypoint (`keypoints[:, 0]`) is greater than or equal to the offset.
- The x-coordinate of the keypoint is less than the image width minus the offset.
- The y-coordinate of the keypoint (`keypoints[:, 1]`) is greater than or equal to the offset.
- The y-coordinate of the keypoint is less than the image height minus the offset.

The filtered keypoints are then returned as a numpy array.

3 ssd

The squared differences between the two sets of descriptors are calculated using broadcasting. This allows efficient element-wise subtraction and squaring of the differences between all pairs of descriptors.

Finally, the function sums the squared differences along axis 1 (columns) to obtain the squared distance between each descriptor in `desc1` and each descriptor in `desc2`. The result, `distances`, is returned as a numpy array.

4 match_descriptors

The following steps are common to all three matching methods:
1. Find the index of the minimum distance for each row (finding the nearest neighbor in `desc2` for each descriptor in `desc1`).
2. Create indices associated to each descriptor in `desc1`.
3. Concatenate the indices to obtain the matches. Each row of `matches` contains the index of a descriptor in `desc1` matched with its nearest neighbor in `desc2`.

.

Depending on the chosen matching `method`, additional steps are performed:
- For "one_way" method, no additional action is taken as the matches are already obtained. The quality of matches with this method is not too high as can be seen in the images below (some keypoints are matched while they should not be)
- For "mutual" method, a mask is created to filter matches where the nearest neighbor principle is invertible ; specifically, the pair (i,j) is set to True in the mask if the minimum of row i is j and the minimum of column j is i. Only matches that satisfy this condition are retained. This method improves a little bit the quality of matches as can be seen in the images below.
- For "ratio" method, we compute the second smallest value for each row in the squared distances matrix using 'np.partition'. This value represents the distance to the second nearest neighbor. We do the same for the smallest value in each row, then compute the ration (smallest value)/(second smallest value) and keep only the matches for which this ratio is above a threshold, using a mask. The "ratio" method is particularly effective in improving the quality of matches by discarding matches with ambiguous distance ratios, resulting in a more reliable set of keypoints (see images below)

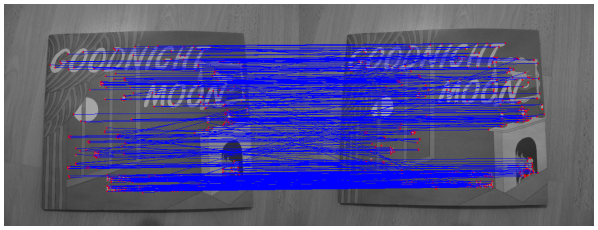

Figure 5: match "one way" method (not too reliable), using default parameters

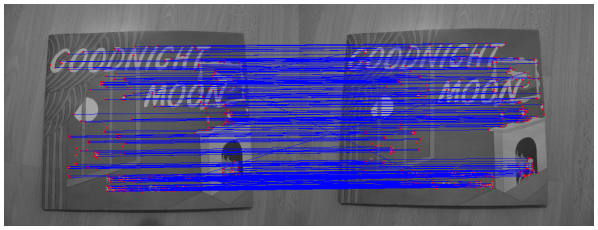

Figure 6: match "mutual" method (better results), using default parameters

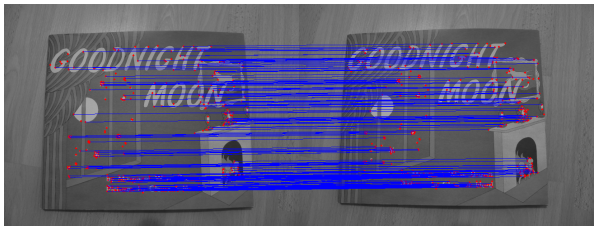

Figure 7: match "ratio" method (very reliable), using default parameters