**Computer Vision**
ETH Zurich, Fall 2023
Hardik Shah

# Assignment 1: Feature extraction and Optical flow

**Hardik Shah**

## 1 Overview

The assignment is about implementing the Harris Feature extractor. Typically in computer vision we are interested in locating interest points on an image. Corners in an image are areas of large intensity changes and hence, we want to extract corner points from an image as "feature points". The assignment consists of two major objectives, that are elaborated in the sections below:

1. **Corner Detection**: Using the Harris Corner detection algorithm to detect corners given an input image.
2. **Feature Matching**: Given two images $I_1$ and $I_2$, use the Harris Corner Detection algorithm to detect corners, and then match the detected between the two given images.

## 2 Corner Detection

The Harris corner detection algorithm works on the underlying principle that if a particular point is a corner, then moving that point along with some neighborhood should lead to large intensity changes in all directions i.e. measuring the intensity differences between a patch and windows shifted in several directions. The algorithm takes an image $I$, parameters $\sigma_i$, $\sigma_d$, $K$, $threshold$, and returns the corners detected in the image as $C = (u_1, v_1, (u_2, v_2)...(u_n, v_n)$ where $(u, v)$ denotes pixel coordinates of a detected corner point.
The error function given by

$$E(u,v) = \sum_{(x,y)\in W} [I(x+u, y+v) - I(x,y)]^2 \tag{1}$$

captures the sum of squared differences($SSD$) in image intensities by moving the patch $W$ by $(u, v)$ where $u$ and $v$ represent translation of the patch in $x$ and $y$ directions respectively. The taylor expansion on $I$ and using a gaussian kernel(with $var = \sigma_i^2$) as the window function($W$) gives us the following equation:

$$\begin{aligned} E(\Delta x, \Delta y) &= \begin{bmatrix}\Delta x & \Delta y\end{bmatrix} \mathbf{M} \begin{bmatrix}\Delta x \\ \Delta y\end{bmatrix} \\ \text{where, } M &= \sum_{(x,y)\in P} g(\sigma_i, x, y) * \begin{bmatrix} I_x(\sigma_d)I_x(\sigma_d) & I_x(\sigma_d)I_y(\sigma_d) \\ I_x(\sigma_d)I_y(\sigma_d) & I_y(\sigma_d)I_y(\sigma_d)\end{bmatrix} \end{aligned} \tag{2}$$

In (2), $\sigma_d$ denotes the std. of gaussian smoothing applied to the image derivatives $I_x$ and $I_y$, $P$ denotes the coordinates where the gaussian kernel(as window function) is applied to which depends upon the kernel size, and $g(\sigma_i, x, y)$ denotes the weight of the gaussian kernel indexed at $(x, y)$. Essentially, for each of the element in $M$, we are taking a weighted average of the neighbourhood pixel values in the patch $P$.

## 2.1 Computing autocorrelation matrix(M)

The image derivatives $I_x$ and $I_y$ are first computed by convolving the derivative filters $dx = \begin{bmatrix}0.5 & 0 & -0.5\end{bmatrix}$ for $I_x$ and $dy = \begin{bmatrix}0.5 & 0 & -0.5\end{bmatrix}^T$ for $I_y$ using `scipy.convolve2d`. Notice that the filters are flipped than the intended order of multiplication, this is because in a convolution operation the kernels are flipped by 180° before convolving the filter. We use `mode="same"` for `scipy.convolve2d` so that size of the image $I$ and the image derivatives $I_x$ and $I_y$ match. For this assignment, we ignore gaussian smoothing of the gradients with $g(\sigma_d)$, but in general it is a parameter that can be added to the Harris detector. Next, we compute:

$I_x^2 = g(\sigma_i) * I_x^2$
$I_y^2 = g(\sigma_i) * I_y^2$
$I_x I_y = g(\sigma_i) * I_x I_y^2$

where $*$ denotes the convoution operation and $g(\sigma_i)$ denotes a gaussian kernel. We use `cv2.GaussianBlur` and specify the parameter $\sigma_i$ for convolving the input matrix with a gaussian kernel. Note that we use `kernel_size=(0,0)` which implies `cv2` automatically chooses a suitable value for the `kernel_size` using the parameter $\sigma_i$.

## 2.2 Computing Harris Response

The error function given in (2) must be used to classify each pixel in an image as a corner/non-corner point. We can derive that $E(\Delta x, \Delta y)$ in (2) can be expressed as

$$E(\Delta x, \Delta y) = \lambda_1 ||Q^T \Delta x||^2 + \lambda_2 ||Q^T \Delta y||^2 \tag{3}$$

where $\lambda_1$ and $\lambda_2$ are eigenvalues of $M$, and $Q$ is the orthonormal basis of eigenvectors of $M$(eigenvectors of $M$ as colums of $Q$). This implies that $E(\Delta x, \Delta y)$ is a paraboloid(when you cut a slice, you get an ellipse) with the axes of the ellipse aligned with the eigenvectors. Thus, change in direction of the eigenvector corresponding to the larger eigenvalue will lead to a sharper change in $E$. For a corner point, we want both $\lambda_1$ and $\lambda_2$ to be large. This leads to modeling the response function $R$ as

$$\begin{aligned} & R = det(M) - K * trace(M)^2 \\ \implies & R = (\lambda_1 . \lambda_2) - K * (\lambda_1 + \lambda_2)^2 \\ \implies & R = (I_x^2 I_y^2 - (I_x I_y)^2) - K * (I_x^2 + I_y^2)^2 \end{aligned} \tag{4}$$

The parameter *threshold*, which is essentially the value to threshold the response function $R$ to classify a pixel as corner/non-corner i.e. only if $R > threshold$, pixel is a detected corner.

## 2.3 Non-Max Suppression

After thresholding $R$, we may obtain many corner points clustered together, essentially pointing to a single corner point in the image. To prevent this, we must perform non-max

suppression which implies suppressing values of $R$ that are not maximum in a specified neighbourhood. We use `scipy.ndimage.maximum_filter` that acts like a max pooling operator with size `(3,3)` on $R$. This returns an array of the same size as $I$ and each pixel replaced by the maximum of value of $R$ in a `3x3` patch centered at that pixel. We finally extract the pixels that satisfy `R>threshold & R==NonMaxSuppressed(R)` as corner points from the image.
Figure 1 shows the output of Harris corner detection using the default parameter values for $\sigma_i$, $K$, *threshold*. The corners are marked as red dots on the original image.

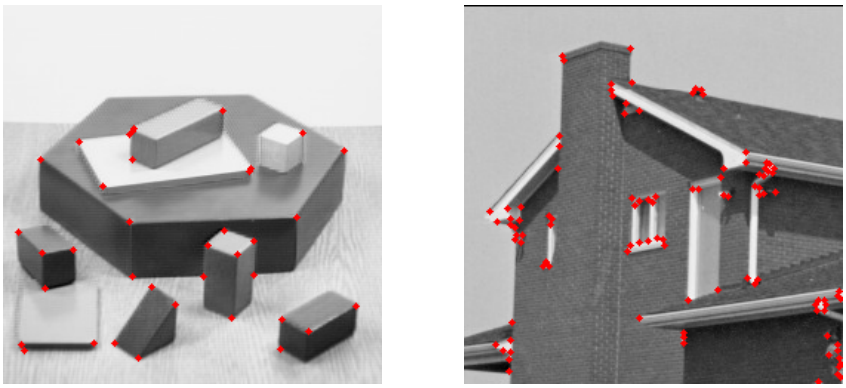

Figure 1: Harris corner detection on the given sample images "House" and "Blocks"

## 2.4 Experimenting with values of $\sigma_i$, $K$, *threshold*

The values of $\sigma_i$, $K$, *threshold* need to be tuned after analysing the results so that optimal number of corners are detected.

1. **K** appears in (4) and clearly, increasing $K$ will reduce the response $R$ which in turn will reduce the number of corners detected.
2. **threshold** directly controls the number of corners detected, a higher threshold implies lesser corners.
3. $\sigma_\mathbf{i}$ determines the weight of the gaussian kernel used for convolving with the elements of $M$ and essentially also determines the kernel size (`cv2.GaussianBlur` automatically chooses the kernel size given value of sigma so that $\pm 3\sigma$ are accomodated in the kernel).

We experiment with the values of $K$ as `K ∈ [0.04, 0.05, 0.06]`,
*threshold* as `threshold ∈ [1e-4, 1e-5, 1e-6]`, and
$\sigma_i$ as `sigma` *in* `[0.5, 1.0, 1.5]`.
The results for these experiments are displayed for the "Blocks" picture in Figure 2 and "House" picture in Figure 3.

Figure 2: Experimenting with values of $\sigma_i$, $K$, *threshold* on "Blocks" image

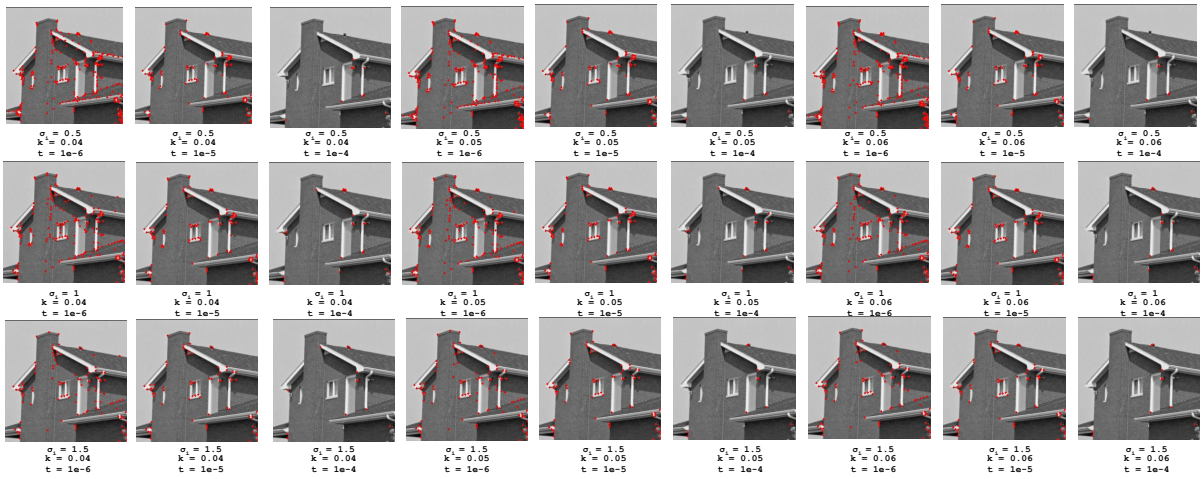

Figure 3: Experimenting with values of $\sigma_i$, $K$, *threshold* on "House" image

Observing the results of using different values, we choose $\sigma_\mathbf{i} = \mathbf{1.0}$, $\mathbf{K = 0.05}$, $\mathbf{threshold = 1e-05}$ as they give decent results without many outliers.

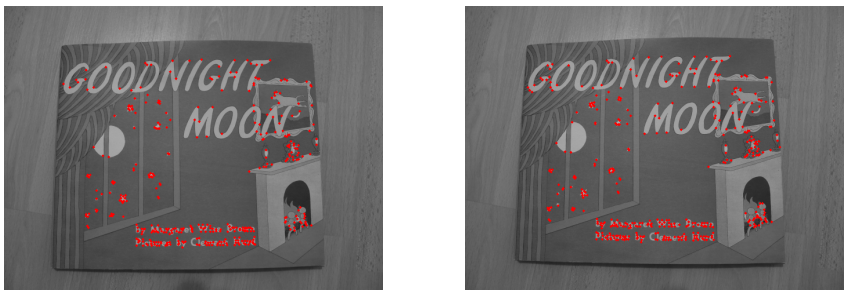


Figure 4: Harris corner detection on the images to match $I_1$ and $I_2$

# 3 Feature Matching

Feature matching entails comparing the detected corners in two different images of the same scene and then matching two corners as corresponding to the same world co-ordinate. Given corners $C_1 = (u_1, v_1), (u_2, v_2)...(u_n, v_n)$ for $I_1$, and $C_2 = (r_1, p_1), (r_2, p_2)...(r_m, p_m)$ for $I_2$, feature matching establishes an equivalence between $(u_i, v_i) \in C_1$ and $(r_j, p_j) \in C_2$ such that the same "world-frame" object occupies both $(u_i, v_i)$ and $(r_j, p_j)$
i.e. $W_1((u_i, v_i)) \approx W_2((r_j, p_j))$ where $W_1$ and $W_2$ denote the transformation matrices that project points in the camera frame to the world frame for $I_1$ and $I_2$ respectively.

To compare two corner points, we first extract a descriptor for each corner point. For extracting a descriptor we can use sophisticated methods like deep learning that projects a neighbourhood around the corner into a lower dimensional latent space, however for this assignment we simply collect pixel intensities in a `(9x9)` patch around the corner point and flatten them into an 81 dimensional column vector. Corner points for which the (9x9) patch overshoots the boundary of the image are removed from our matching. We perform this check using the `filter_keypoints` function.

The metric to match two corner points is the distance between the descriptors of the two points. We can define this distance as Sum of squared differences($SSD$) between the descriptors.

$$SSD(p, q) = sum_i(p_i - q_i)^2 \tag{5}$$

A vectorized version of SSD computation has been implemented in the `ssd` function using `numpy.meshgrid` and `numpy` indexing.

We explore three methods to establish such equivalence between corners computed using the Harris detector algorithm.

## 3.1 One-Way Nearest Neighbours Matching

For each descriptor $p \in C_1$, we look for the descriptor $q \in C_2$ that minimizes $SSD(p, q)$. For implementing this we simply take an `argmin` over `axis=1` of the computed `distances` between descriptors in $C_1$ and $C_2$. So, if $|C_1| = n$ and $|C_2| = m$, we always get $n$ matches via this approach. This being a very rudimentary and basic matching technique, there are mismatches to some extent as seen in Figure 5, characterized by the slanting/non-parallel lines to the horizontal image edges.

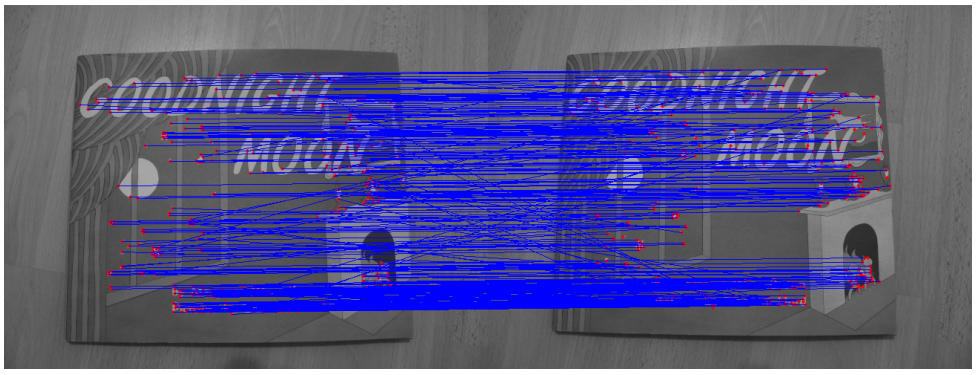


Figure 5: One-way Nearest Neighbour matching between corner points of $I_1$ and $I_2$

## 3.2 Mututal Nearest Neighbours Matching

Descriptors $p \in C_1$ and $q \in C_2$ are matched if

$$\underset{x \in C_2}{\text{argmin}}\ SSD(p, x) = q \textbf{ and } \underset{x \in C_1}{\text{argmin}}\ SSD(x, q) = p \tag{6}$$

Basically if $q \in C_2$ is one-way nearest neighbour match for $p \in C_1$, then $p$ must also be a one-way match for $q$ for $p$ and $q$ to be mutually matched. It is evident from the formulation of this matching that the number of matches will be less than one-way nearest neighbour matching. For implementing this, we compute one-way nearest neighbour matches for each $p \in C_1$, and also for each $q \in C_2$ and take the intersection of these two sets i.e. indices where `numpy.logical_and(`$C_1$`, `$C_2$`)` is `True` and we define $C[i, j]$`=True` if there is a one-way match between descriptors `i` and `j`.
Mismatches are considerably lesser than in one-way nearest neighbour matches as seen in Figure 6.

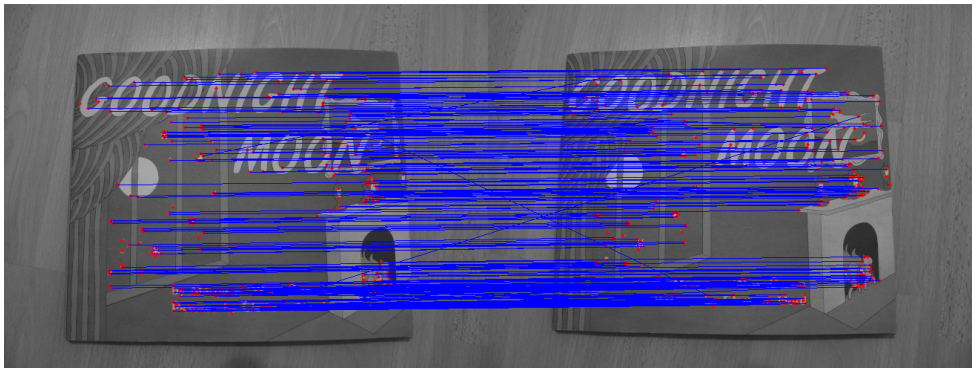


Figure 6: Mutual Nearest Neighbour matching between corner points of $I_1$ and $I_2$

## 3.3 Ratio Test Matching

For a one-way nearest neighbour match between $p \in C_1$ and $q \in C_2$, and $t \in C_2$ is the second nearest one-way match for $p$, then $p$ and $q$ are matched only if $\frac{q}{t} < r$ for a pre-decided ratio threshold $r$. For the assignment we use `r=0.5`. Again, clearly the number of matched points will be lesser than in one-way nearest neighbour matching since we are enforcing a stronger condition here. We implement this using `numpy.partition` function that gives us the second nearest neighbour for each descriptor in $C_1$. We precompute the one-way nearest neighbours for $C_1$, then assuming $|C_1| = n$, we select $m < n$ matches where the match ratio does not exceed $r$. This appears to be the most robust of all feature matching approaches implemented in this assignment, as seen in Figure 7.

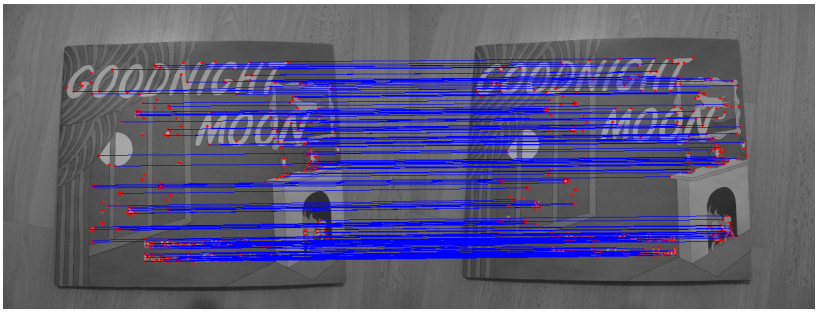


Figure 7: Ratio Test matching between corner points of $I_1$ and $I_2$