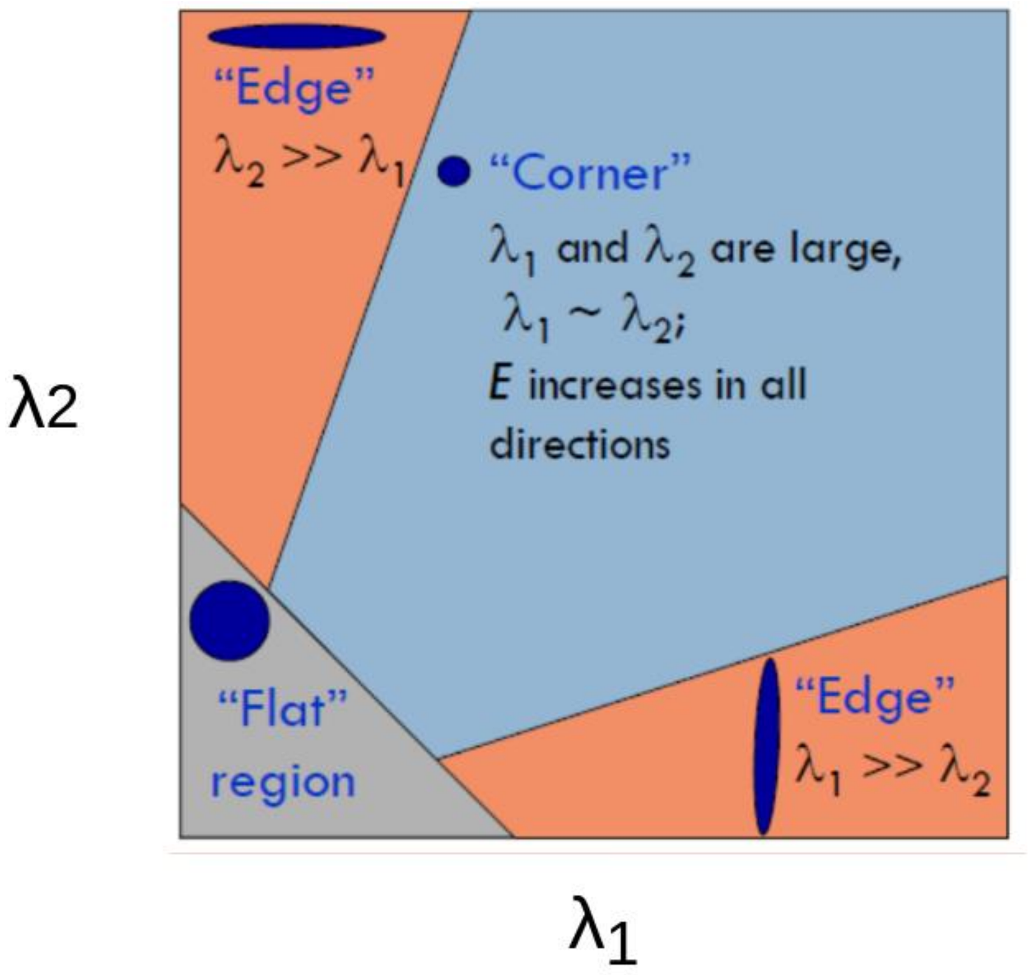
Shreyash Agrawal
MTech CSE
2021201074

Computer Vision Assignment-2 Report

HARRIS CORNER DETECTOR

A corner is a point whose local neighborhood stands in two dominant and different edge directions. In other words, a corner can be interpreted as the junction of two edges, where an edge is a sudden change in image brightness. Corners are the important features in the image, and they are generally termed as interest points which are invariant to translation, rotation, and illumination.

The idea is to consider a small window around each pixel p in an image. We want to identify all such pixel windows that are unique. Uniqueness can be measured by shifting each window by a small amount in a given direction and measuring the amount of change that occurs in the pixel values.


"Edge"
$\lambda_2 >> \lambda_1$
"Corner"
λ_1 and λ_2 are large,
$\lambda_1 \sim \lambda_2$;
E increases in all directions
"Flat" region
"Edge"
$\lambda_1 >> \lambda_2$
λ2
λ_1

Lucas-Kanade Algorithm

1. Get the target features from the corner detector function cv2.goodFeaturesToTrack() (with Harris Corner Algo).

2. Normalize the image

3. Determine the I_x and I_y gradients using Sobel filter.

4. Calculate the I_{xx} as I_x*I_x, I_{yy} as I_y*I_y and I_{xy} as I_x*I_y.

5. Calculate I_t as difference of two images (img2-img1). [Subtracting img1 from img2 since b vector as negative sign which gets considered within]

6. Iterate over every feature: (x,y)

    6.1. Define a window around the feature (x,y). [Window size is user defined]

    6.2. Extract I_x, I_y, I_{xx}, I_{xy}, I_{yy} and It for the same window.

    6.3. Define matrix $(A^T)A$ = [[sum(I_{xx}) sum(I_{xy})], [sum(I_{xy}), sum(I_{yy})]],

    6.4. Calculate $It_x = I_x*It$ and $It_y = I_y*It$ for the same window

    6.5. Calculate vector $(A^T)b$ = [[sum(It_x) sum(It_y)]].T (and take the transpose of it)

    6.6. Find the inverse of $(A^T)A$ and then find the dot product with $(A^T)b$.

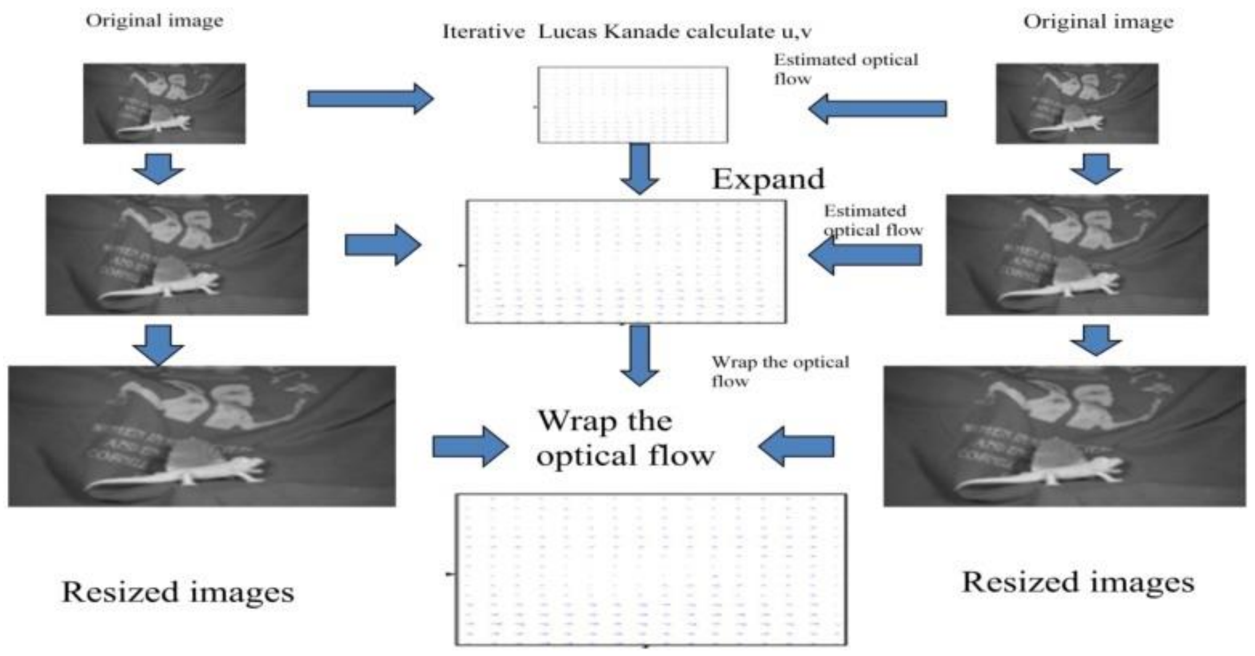
    6.7. Store the resulting (u,v) direction vectors

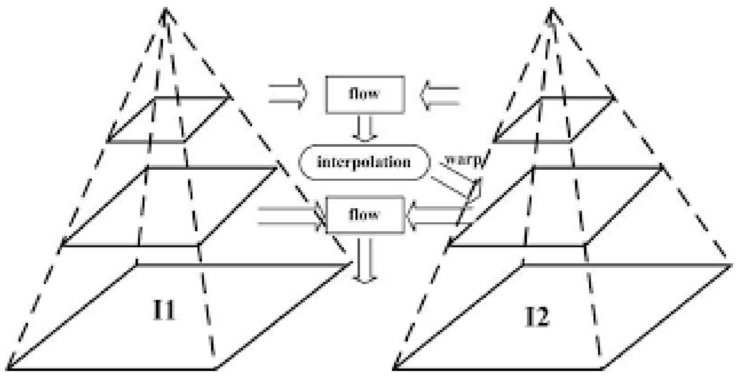
7. Plot all the direction vectors (u,v) for each feature over the image. (plt.quiver())

8 Return the (u,v) director vectors. This represents the optical flow across the points.

Multi-scale coarse to fine optical flow

The basic idea is to initially compute the optical flow at a coarse image resolution. The obtained optical flow can then be up sampled and refined at successively higher resolutions, up to the resolution of the original input images. By computing the optical flow in this manner, we can circumvent the aperture problem to some degree – because the window size remains fixed across all resolutions, the algorithm effectively computes the optical flow using successively smaller apertures. This approach also works well on images with large displacements, something the single-scale algorithm is unable to handle.





OpticalFlowRefine(Img1, Img2, windowSize, u0, v0) This function refines the optical flow according to,

u = u0 + Δu and v = v0 + Δv

where Δu and Δv represent the offset between the initial estimate and the refined estimate.

To compute Δu and Δv, we simply shift the window in Img2 by (u0 ,v0) and compute the optical flow between the shifted windows.

MultiScaleLucasKanade should implement the following pseudo-code,

```
Step 01. Gaussian smooth and scale Img1 and Img2 by a factor of 2ˆ(1 - numLevels).
Step 02. Compute the optical flow at this resolution.
Step 03. For each level,
    a. Scale Img1 and Img2 by a factor of 2ˆ(1 - level)
    b. Upscale the previous layer‘s optical flow by a factor of 2
    c. Compute u and v by calling OpticalFlowRefine with the previous level‘s optical flow
```