Report on: **Optical Flow Estimation (Lucas-Kanade Method)**

Atanda Abdullahi Adewale

INTRODUCTION:

Optical flow is the pattern of apparent motion of image objects between two consecutive frames caused by the movement of object or camera. It is 2D vector field where each vector is a displacement vector showing the movement of points from first frame to second

This exercise is to determine and display the optical flow field (u, v) direction between two or more sequence or frames of images (subsequent frames from a video).

This involves finding the motion (u, v) that minimizes the sum-squared error of the brightness constancy equations for each pixel in a window

KEY ASSUMPTIONS:

- Brightness constancy for intensity images
- Small Motion
- Flow is constant for all pixels (Neighboring Pixel has same displacement)

METHOD:

Optical Flow Constraint:

$$\frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t} = 0$$

$$\boxed{I_x u + I_y v + I_t = 0}$$

Using a 5x5 image Patch, gives 25 equations

$$\begin{bmatrix} I_x(\boldsymbol{p}_1) & I_y(\boldsymbol{p}_1) \\ I_x(\boldsymbol{p}_2) & I_y(\boldsymbol{p}_2) \\ \vdots & \vdots \\ I_x(\boldsymbol{p}_{25}) & I_y(\boldsymbol{p}_{25}) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_t(\boldsymbol{p}_1) \\ I_t(\boldsymbol{p}_2) \\ \vdots \\ I_t(\boldsymbol{p}_{25}) \end{bmatrix}$$

$$\underset{25 \times 2}{A} \qquad \underset{2 \times 1}{x} \qquad \underset{25 \times 1}{b}$$

The Least Square Approximation

$$\hat{x} = \arg\min_x ||Ax - b||^2 \text{ is equivalent to solving } A^\top A\hat{x} = A^\top b$$

$$\begin{matrix} A^\top A & & \hat{x} & & A^\top b \end{matrix}$$

$$\begin{bmatrix} \sum_{p\in P} I_x I_x & \sum_{p\in P} I_x I_y \\ \sum_{p\in P} I_y I_x & \sum_{p\in P} I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum_{p\in P} I_x I_t \\ \sum_{p\in P} I_y I_t \end{bmatrix}$$

where the summation is over each pixel $\boldsymbol{p}$ in patch $\boldsymbol{P}$, obtain [u, v]

METHODS:
With random colors, we show the displacement in 5 consecutive video using the Lukas-Kanade algorithm and Shi-Tomasi Corner detector

Step 1:
Load the video file that will be analyzed, get the width and height of the frames in the video, needed to set the size of the output video.

```
cap = cv2.VideoCapture('walking.avi')
width = int(cap.get(3))
height = int(cap.get(4))
out = cv2.VideoWriter('optical_flow_walking.avi', cv2.VideoWriter_fourcc('M','J','P','G'), 30, (width, height))
```

Step 2:
Obtain points $P_1 \ldots P_{25}$ in the first frame using Shi-Tomasi corner detection algorithm; corners in the image that will be tracked

Shi-Tomasi R-score $\quad R = \min(\lambda_1, \lambda_2)$

By setting these Shi-Tomasi parameters (*maxCorners, qualityLevel, MinDistance, blockSize*), the Shi-Tomasi corner detection algorithm will detect up to 100 corners in the image with a minimum quality level of 0.3 based on their eigenvalues. Corners closer than 7 pixels to each other will be rejected, and a 7×7 pixel window will be used for corner detection.

Detected and selected good Corners in random color points


```
# Set parameters for ShiTomasi corner detection
feature_params = dict( maxCorners = 100,
                       qualityLevel = 0.3,
                       minDistance = 7,
                       blockSize = 7 )

# Set parameters for lucas kanade optical flow
lucas_kanade_params = dict( winSize  = (15,15),
                  maxLevel = 2,
                  criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 0.03))
```

Step 3:
To estimate motion vector with *Lukas-Kanade in cv2*, we set Lucas-kanade parameters;

- *winSize*: the size of the search window for motion vectors to 15×15, specified as a tuple of (width, height) in pixels.
- *maxLevel*: set the maximum pyramid level number to 2. The algorithm computes the optical flow at multiple resolutions
- *criteria*: the termination criteria for the iterative Lucas-Kanade algorithm (*type, max_iter, epsilon*), where type specifies the type of termination criteria, In this case, type is a combination of cv2.TERM_CRITERIA_EPS and cv2.TERM_CRITERIA_COUNT

The algorithm terminates if either the maximum number of iterations (max_iter) is reached or the minimum change in parameters (epsilon) is below a certain threshold. max_iter is set to 10, and epsilon is set to 0.03.

Step 4:
With *cv2.goodFeaturesToTrack(),* we select only the best corners from *Shi-Tomasi* initial corners, for optical flow estimation.

Create *prev_gray, a* grayscale version of the previous frame, which is needed for calculating optical flow and *prev_corners* are the points of interest from the previous frame, and these points are used as the starting points for the Lucas-Kanade algorithm to track the motion of those points in the next frame.

Step 5:
we calculate the optical flow using *cv2.calcOpticalFlowPyrLK(), pass in .prev_gray, frame_gray, prev_corners and lucas_kanade_params*

- then select only the good points using the ***status*** (0 or 1 array received from *cv2.calcOpticalFlowPyrLK()* i.e., indicates points that were successfully tracked) and
- A while loop is used to iterate over each frame in the video. The next frame is read using cap.read(), and it is converted to grayscale
- then calculate the **motion vectors** by subtracting the old positions from the new positions.
  u and ***v*** variables represent the x and y components of the motion vectors, respectively.

Step 6:
Finally, we visualize the motion vectors

- Create a mask image for drawing the motion vectors on the image, and it is initialized to a black image using np.zeros_like().
- Generate a list of random colors, to draw or color the motion vectors of the tracked corners on the output video.

  Start while loop for all frames, to runs until no more frames to process in video

.

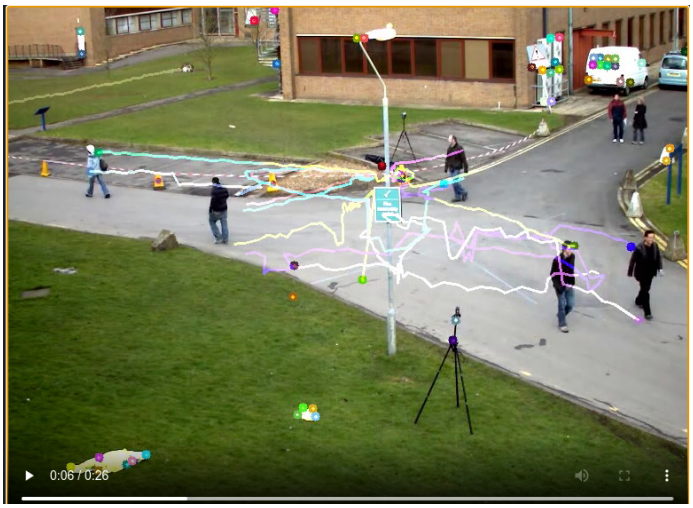
- The motion vectors and the tracks are then drawn on the image using cv2.line() and cv2.circle() functions. The mask image is updated with the new tracks using cv2.line(), and the circles are drawn on the current frame using cv2.circle().
- The tracked image is obtained by adding the mask to the current frame using cv2.add().
- The tracked image is then written to the output video using out.write().

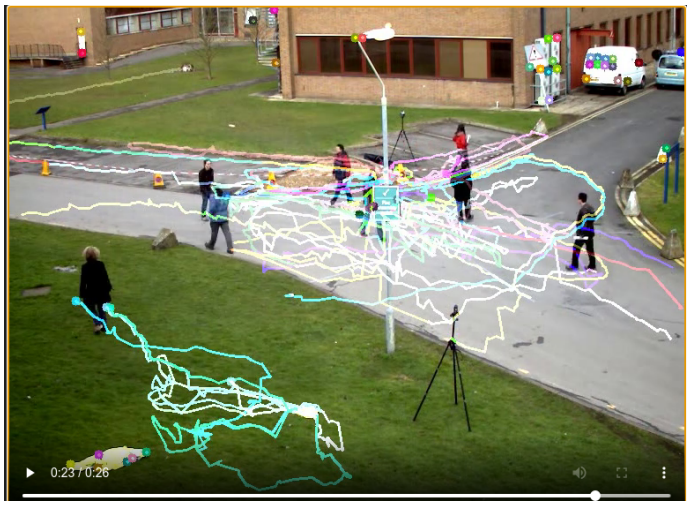
Step 7:
inside the while loop, we need to use the current frame as the previous frame in the next iteration. To get the OF for the next frame :

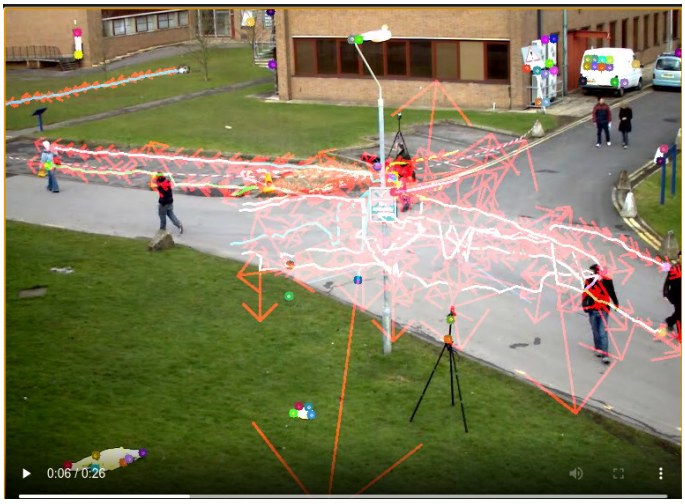
- the previous grayscale frame is updated to the current grayscale frame, and the previous corners are updated to the good new corners.
- While loop continues until there are no more frames in the video, at which point the video capture is released using cap.release() and the output video writer is released using out.release().

```
# Draw the tracks
for i, (new, old) in enumerate(zip(good_new, good_old)):
  a, b = new.ravel()
  c, d = old.ravel()

  u = good_new[:,0] - good_old[:,0]
  v = good_new[:,1] - good_old[:,1]

  # Calculate the end points of the arrows
  arrow_len = 10
  arrow_scale = 3
  points = np.int32(good_new)
  endpoints = points + arrow_scale*np.int32([u, v]).T


  # Draw the line and arrow on the mask
  mask = cv2.line(mask, (int(a), int(b)), (int(c), int(d)), color[i].tolist(), 2)

  '''
  uncomment below Mask to show direction arrow
  '''
  #mask = cv2.arrowedLine(mask, (int(a), int(b)), tuple(endpoints[i]), (0, 0, 255), thi

  # Draw the circle on the frame
  frame = cv2.circle(frame, (int(a), int(b)), 5, color[i].tolist(), -1)
```

OF tracked images in the generated video







OF with [U, V] motion direction arrows

Summary:

The Optical flow program demonstrated with the Lucas-Kanade algorithm works fine and successfully estimates the motion of objects in a video stream.

However, when there are are larger instantaneous movement in the scene, the Lukas-Kanade algorithm fails to generate good Optical flow.

To make this OF algorithm better, several improvements can be made:

- a more robust corner detection method such as the Harris corner detection algorithm.
- a more sophisticated optical flow algorithm that can handle larger displacements and more complex motions, such as the Farneback (Dense Optical flow) algorithm.

Additionally, post-processing techniques such as outlier rejection and data smoothing can be applied to improve the accuracy and consistency of the optical flow estimates.

Attached codes file:

- Atanda_Adewale_Optical Flow.ipynb
- Walking.avi