University of Nottingham
UK | CHINA | MALAYSIA

COMP3065 CV Coursework

Submitted May 2022, in partial fulfillment of
the conditions for the award of the degree
BSc Computer Science and BSc Computer Science with Artificial Intelligence.

Shengnan Hu, 20126376

Contents

List of Tables

List of Figures

Chapter 1

Key features

1.1 Optical Flow

The objective of this feature is to use the optical flow to track and illustrate the motion of the objects.

1.1.1 Implementation steps

Firstly, find the feature points on the first frame with cv2.goodFeaturesToTrack(). Then the Optical Flow will be computed with the Lucas-Kanade algorithm using the feature points' coordinates information.

The optical flow is calculated in the following steps:
1. Calculate the Ix, Iy, and It of all the pixels of the image by the code below:

```
kernel_x = np.array([[0.5, 0, -0.5]])
kernel_y = np.array([[0.5], [0], [-0.5]])
kernel_t = np.array([[-1]])
```

```
Ix = scipy.ndimage.convolve(input=firstImage,
   weights=kernel_x, mode="nearest")
Iy = scipy.ndimage.convolve(input=firstImage,
   weights=kernel_y, mode="nearest")
It = scipy.ndimage.convolve(input=secondImage,
   weights=kernel_t, mode="nearest") + scipy.ndimage
   .convolve(
    input=firstImage, weights=-kernel_t, mode="
       nearest"
)
```

2. Obtain the Ix, Iy, and It of each feature point and use these values to calculate the flow by the following code, which implements the function 1.1 and 1.2.

```
Ix_win = Ix[j - window: j + window + 1, i - window:
   i + window + 1,].flatten()
Iy_win = Iy[j - window: j + window + 1, i - window:
   i + window + 1,].flatten()
It_win = It[j - window: j + window + 1, i - window:
   i + window + 1,].flatten()

A[0][0] += np.dot(Ix_win, Ix_win)
A[0][1] += np.dot(Ix_win, Iy_win)
A[1][0] += np.dot(Ix_win, Iy_win)
A[1][1] += np.dot(Iy_win, Iy_win)
b[0][0] += np.dot(Ix_win, It_win)
b[1][0] += np.dot(Iy_win, It_win)
```

```
Ainv = np.linalg.pinv(A)
(u, v) = np.dot(Ainv, b)
```

$$Ax = -b \tag{1.1}$$

$$A^T Ax = A^T(-b) \tag{1.2}$$

Then the new position of each feature point is calculated by (u, v). Then check whether the feature points have been tracked successfully. The second image is used as the initial frame, the first image is used as the next frame, and the transformed feature points are passed into the LK algorithm as the feature points of the initial frame to obtain the optical flow value. Then the new position of the feature points on the first image is found. By comparing the original feature point positions with these positions, I determined that the feature point has been tracked if the difference between them is less than one. The implementation code is shown below:

```
(u, v) = lk(firstImage, secondImage, p0, 5, 3, 4)
flow = np.array(list(zip(list(u), list(v))))
p0 = np.array(np.reshape(p0, (-1, 2)))
flow = np.array(np.reshape(flow, (-1, 2)))
p1 = np.array(p0 + flow)

(u1, v1) = lk(secondImage, firstImage, p1, 5, 3, 4)
flow1 = np.array(list(zip(list(u1), list(v1))))
p11 = np.reshape(p1, (-1, 2))
flow1 = np.reshape(flow1, (-1, 2))
p0r = np.array(p11 + flow1)
```

```
p0 = np.array(p0)
d = abs(p0 - p0r).reshape(-1, 2).max(-1)
cornerIsDetected = d < 1
```

If the feature point is tracked, then the position of the transformed feature point is recorded in the track array. A line is drawn from the position of the previous frame's feature point to the position of the current frame's feature point as a representation of the optical flow.

```
for track, (x, y), is_detected in zip(self.tracks,
   p1, cornerIsDetected):
    if not is_detected: continue
    track.append((np.float32(x), np.float32(y)))
    if len(track) > self.track_len: del track[0]
    new_tracks.append(track)
    cv2.circle(current_frame, (int(x), int(y)), 2,
       (0, 255, 0), -1)
self.tracks = new_tracks
cv2.polylines(current_frame, [np.int64(track) for
   track in self.tracks], False, (0, 255, 0), 1)
```

1.1.2 Results and Analysis

As shown in Figure 1.1, there are several good feature points, not the consistent optical line to show the tracks. I found that each feature point has been translated by (u, v); however, large errors appear on the flow values. While debugging, I found that these key points oscillated within a certain range after several transformations by (u, v). The main reason for this appearance is that the cars in this video remove so fast, violating the Lucas-Kanade algorithm's constraints, small velocity, and regional consistency.

The short optical line shown in Figure 1.2 demonstrates that because of the relatively lower speed, the flow value could be correct in a short period. However, the rate is still so fast. The subsequent assumptions will have large deviations when the object moves more quickly, making significant errors in the final derived luminous flow values.

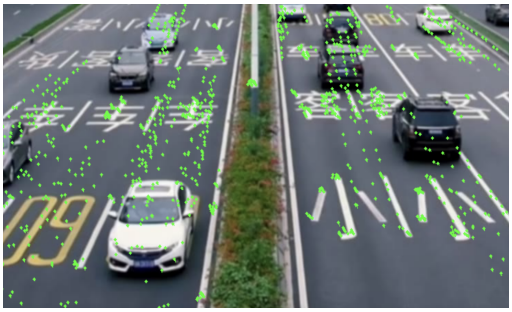

Figure 1.1: LK for high speed Cars

Figure 1.2: LK for relatively low speed Cars

1.1.3 Evaluation

This algorithm is easy to implement and has fast computation speed. However, when the object is moving fast, the algorithm is not able to generate optical flow tracking properly, and the algorithm is subject to many constraints and factors, as I mentioned in the previous section. Also, it is hard for each frame to satisfy three assumptions in the actual video shoot. Objects that need to be tracked often move so fast that the eye cannot observe them in real-world applications. The main target for optical flow tracking

is a fast-moving car. Therefore, the disadvantages of this algorithm are obvious. From my point of view, it is not widely applicable.

1.2 Lucas-Kanade algorithm with the Gaussian Pyramid- additional feature

The LK algorithms are prone to rapid movements. A multi-scaling trick is a typical strategy in practice. The objective of this feature implementaion is solving this problem by using the image pyramid. The implementation appraoch is following the pesuodo-code of the pyramidal affine tracking algorithm proposed by Bouguet et al. (2001).

1.2.1 Implementation steps

Initialize the image pyramids

A Gaussian pyramid is a collection of photos that have been weighted using a Gaussian blur and 1/2 downsampling. Firstly, the lowpass filter (Gaussian filter) is used to smooth the image and then downsample the smoothed images to obtain a series of reduced-size images. The implementation code is shown below:

```
gaussian_matrix = cv2.GaussianBlur(matrix, KERNEL,
   SIGMA, SIGMA)
downsampled = gaussian_matrix[::2, ::2]
```

A stack of half-sized images is created. Each pixel holds a local average that corresponds to a pixel neighborhood on a lower pyramid level by the iteration code shown below. The higher the height of the Gaussian pyramid,

the minor detail and noise there is, and the more abstract the perspective. Two Gaussian pyramids with a user-specified level number are created from the two input images. And the initial optical flow on the top level of the pyramids is set to be zero.

```
for level in range(1, level_number):
    firstImage = downSample(firstImage)
    secondImage = downSample(secondImage)
    firstImagePyramid[0: firstImage.shape[0], 0:
       firstImage.shape[1], level] = firstImage
    secondImagePyramid[0: secondImage.shape[0], 0:
       secondImage.shape[1], level] = secondImage
```

On the higher levels of the pyramid, a pixel's movement outside the neighbor window gets smaller, allowing it to be tracked. On the top level of the pyramids, the algorithm computes optical flow using the Lucas-Kanade method and utilizes the result as the initial value for the following level. Because the size of optical flow computed from level n is half that of level (n+1), both height and breadth must be bilinear interpolated by a factor of two. This technique is repeated until all pyramid levels have been processed.

Iteration on each level of the pyramids

At every level L in the pyramid, the iterations are implemented to obtain a more accurate flow estimation. In each iteration, I first generate the window (5*5) of the feature points of the first frame and translate them by (u, v) to obtain the image slice on the second frame. By translating the neighborhood window (5*5) of the layer according to the optical flow values of the previous layer, the motion could be very small, so the standard LK

algorithm can be used (Tamgade & Bora, 2009). The residual optical flow $(u1, v1)$ returned by the LK algorithm is very small, and the optical flow estimate for the next iteration can be obtained by summing the residual optical flow obtained for each iteration and the optical flow estimate of the previous one, which is $(u + u1, v + v1)$.

```
u_res, v_res = lucas_kanade_tradition(
   firstImageFrame, secondImageFrame, N)
u[i, j] = u_res[window, window] + u_previous[index]
v[i, j] = v_res[window, window] + v_previous[index]
```

Find the optical flows of all levels of pyramid

After each iteration on one level of the pyramid, the results of the optical flow estimate at the previous layer are passed as initial values after upsampling to the next layer of images until the visual flow of the original image (the final layer) is calculated.

1.2.2 Results and Analysis

After applying the pyramid, the optical flow could successfully track the fast-speed car, as shown in Figure 1.3. The improvement is evident and significant. To demonstrate that this method is much less constrained than a single algorithm, I experimented with several videos to analyze the performance of these new features. Figures1.4 and 1.5 show that the feature points could be nearly fully tracked, and the performance is much better than simple LK. The main reasons for this remarkable progress are that the optical flow computation starts with the lowest resolution images at the highest pyramidal level. The abrupt movement on the small-sized image is

more evident than on the huge one in the fixed-size window (Bouguet et al., 2001).

Figure 1.3: LK with pyramids1

Figure 1.4: LK with pyramids2

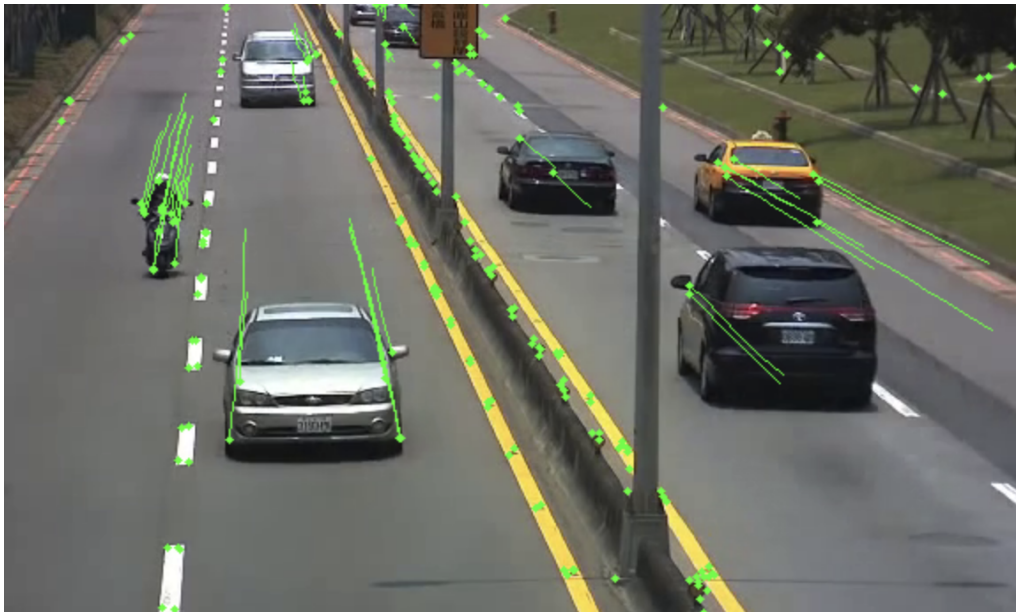

Figure 1.5: LK with pyramids3

1.2.3 Evaluation

As I expected, the Lucas-Kanade algorithm with the Gaussian Pyramid can successfully perform optical flow tracking on fast-moving objects. This extension and improvement to the LK algorithm are successful and effective. The disadvantage of this method itself is that it is more complex to implement. The drawback of my implementation of optical flow tracking is that the algorithm runs extremely slowly for videos with a particularly large number of feature points, such as many trees and other environmental factors. Therefore, the videos I use for the evaluation algorithm are edge-blurred. However, in real life, the problem of a huge number of feature points is difficult to circumvent when tracking, and the blurring may result in important areas not being tracked.

1.3 Muti-object tracker - additional feature

The objective of implementing this feature is to track multi-objects as the same time. The MOSSE filter was introduced by Bolme et al. (2010). The main tracking algorithm I implemented is the mosse tracking algorithm following the steps.

1.3.1 The general Implementation steps of Mosse

This subsection describe the general steps to implementing the Mosse, the detail ideas will be discussed in the next subsection.

Initialize the filter

1. The center (x, y) of the selected rectangle is used to calculate the Gaussian response map g1 of the selected image f1.
2. Pre-training to obtain the original Ai and Bi, which can be used to calculate the most suitable Gaussian filter Hi.

Update the filter

1. Obtain the current filter $Hi = Ai/Bi$
2. Set a frame $f2$ based on the center of the previous target frame
3. Preprocess the current frame $f2$
4. Calculate the Gaussian response $G2$ of $f2$
5. The real gi is calculated by taking the inverse Fourier transform of Gi

6. The position of the maximum value in the gi is the position of the target in the new image frame

7. Calculate the center of this target

8. Use the new center to take a new frame

9. Update the filter

1.3.2 The details of main implementations of Mosse - additional feauture

Pre-processing

A preprocessing step is performed on each video frame before initializing the filter and tracking the object. The algorithm will take the FFT of the template obtained from the preprocessing step and the frame. The image is multiplied by a cosine window so that the pixel values near the edges are gradually reduced to zero. This also has the advantage of putting the focus close to the center of the target. Because the Fourier transform is periodic, it does not respect the boundaries of the image. A significant discontinuity between the relative edges of a non-periodic image will result in a noisy Fourier representation.

Pre-training

Using only the initial filter h1 derived from the first frame is only adequate for a relatively independent task, and only small changes can lead to non-recognition. Therefore, after determining the tracking target, the frame needs to be transformed to produce multiple training sets. In this way,

many filters hi can be obtained for multiple, and the one that is suitable for most of the images can be found among the multiple filters. I process the selected image fi by some stochastic affine transformations and obtain the training set. Then take the Fast Fourier Transform (FFT) of the current image f1 obtained after pre-processing. Then update H1 and H2. The code for pre-training is shown below. After several iterations of training, the most suitable Hi is equal to $H1/H2$, which implements the function shown in Figure 1.6 proposed by Bolme et al. (2010)

```
for r in range(100):
        fi = randomWarpImage(self.fi)
        Fi = np.fft.fft2(self.preprocess(fi))
        H1 += G * np.conj(Fi)
        H2 += Fi * np.conj(Fi)
```

$$H^* = \frac{\sum_i G_i \odot F_i^*}{\sum_i F_i \odot F_i^*}$$

Figure 1.6: pre-training

Filter Update

The filter is updated from the second frame by implementing the equation proposed by Bolme et al. (2010) shown in Figure 1.7 as the code shown below. is the learning rate, and the paper takes a value of 0.125 (Bolme et al., 2010). This algorithm puts more weight on recent frames and lets the effect of previous structures decay exponentially over time.

```
Fi = np.fft.fft2(fi)
H1 = learningRate * (self.G * np.conj(Fi)) + (1 -
   learningRate) * self.H1
```

```
H2 = learningRate * (Fi * np.conj(Fi)) + (1 -
    learningRate) * self.H2
```

$$
\begin{aligned}
H_i^* &= \frac{A_i}{B_i} \\
A_i &= \eta G_i \odot F_i^* + (1-\eta) A_{i-1} \\
B_i &= \eta F_i \odot F_i^* + (1-\eta) B_{i-1}
\end{aligned}
$$

Figure 1.7: update filter

1.3.3 Results and Analysis

Firstly, pausing the video and drawing several rectangles around the objects which will be tracked, which is shown in Figure 1.8. Then start the video to track the object's movement, as shown in Figure 1.9. Also, more time selections are allowed. You can pause the video to select another part you are interested in and track it, as shown in Figure 1.10. And Figure 1.11 shows the tracking.

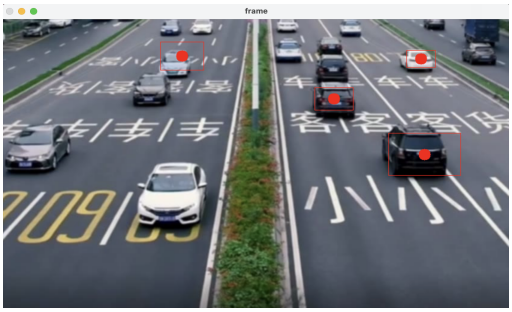

Figure 1.8: select objects

As shown in the above figures, the multi-object trackers perform pretty well. To further analyze the performance of this algorithm, I use the following video. I found that If the rectangles of different objects are not intersected with others, the performance is good, as shown in Figure 1.12 and 1.13. However, If the two frames are very close together, there will be an offset in the tracked object. As shown in Figure 1.14 and 1.15, the tracking

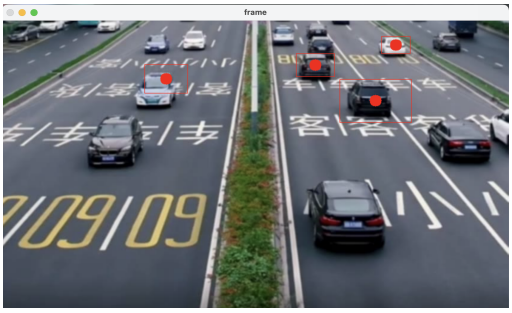

Figure 1.9: track

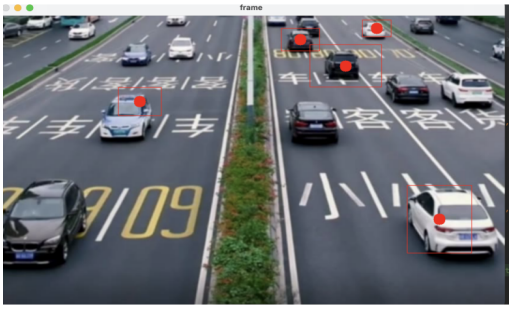

Figure 1.10: pause and select others

Figure 1.11: track again

frame moved to track the wrong objects after the two had brushed past each other. This problem may occur because when finding the maximum value of the Gaussian response map, the algorithm misuses the center of the closest objects to update the center.

1.3.4 Evaluation

As expected, the algorithm succeeds in tracking the selected multiple objects. Moreover, the fast-tracking process can be achieved because the cor-

Figure 1.12: select objects

Figure 1.13: track

Figure 1.14: select objects nearby

relation is computed in the Fourier domain Fast Fourier Transform (FFT) (Bolme et al., 2010). During the experiments, I found that my implementation of this function was subject to misjudgment, especially when the neighboring objects were similar. In the real-world application, a more precise differentiation is needed when tracking.

Figure 1.15: track with misrecognition

1.4 Counting the vehicle number for both up and down - additional feature

1.4.1 Implementation steps

Create the background subtractor, change the mask obtained from the subtractor, perform the opening actions to remove the noise, and then perform the closing action to join some white region. And after the preprocessing, the list of contours is obtained by $findContours()$.

```
mog_mask = mog_subtractor.apply(frame)
ret, dst = cv2.threshold(mog_mask, 150, 255, cv2.
    THRESH_BINARY)
mask = cv2.morphologyEx(dst, cv2.MORPH_OPEN, kernel1
    )
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE,
    kernel2)
contours, hierarchy = cv2.findContours(mask, cv2.
    RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
```

Then according to the potential relative size of one car in the frame, set the restrictions and filter the potential car contours. Then for each contour,

use the $cv2.moment()$ to calculate the centroid and size of the shapes. And calculate the position of the centroid by the code below.

```
M = cv2.moments(cnt)
cx = int(M['m10'] / M['m00'])
cy = int(M['m01'] / M['m00'])
```

In each frame, for the vehicle contours, it is necessary to check if they have been added to the car's array. If there is a contour in the array close to this contour, then it could be considered that this contour has been recorded and then start detecting whether the car is going up or down. If there is no other contour near it, the centroid of this contour is added to the array.

```
 notContainInCars = True
for i in cars:
    if abs(x - i.getX()) <= w and abs(y - i.getY())
       <= h:
        notContainInCars = False
        i.updateCoords(cx, cy)  # Update the
           coordinates in the object and reset age
        if i.up(line):
            count_number_for_up += 1

        elif i.down(line):
            count_number_for_down += 1
        break
if notContainInCars:
    p = Cars(cx, cy)
    cars.append(p)
```

The method for determining up or down is first to draw the line at a

position and make sure the centroid of the car could touch or just near the line. If the position of the centroid of this car in the previous frame is greater than the position of the line and greater than that in the next frame, the direction is considered to be down.

```
self.tracks[-1][1] < line <= self.tracks[-2][1] #up
self.tracks[-1][1] > line >= self.tracks[-2][1] #
    down
```

1.4.2 Results and Analysis

As shown in Figure 1.16, by adjusting the parameters, such as the line height, the cars could be counted accurately in the video. Due to the limited conditions, I can not stand in the middle of the road and hold my phone high enough to record both reversing traffic at the same time accurately. I choosed to anaylsis one direction. To analyze the performance

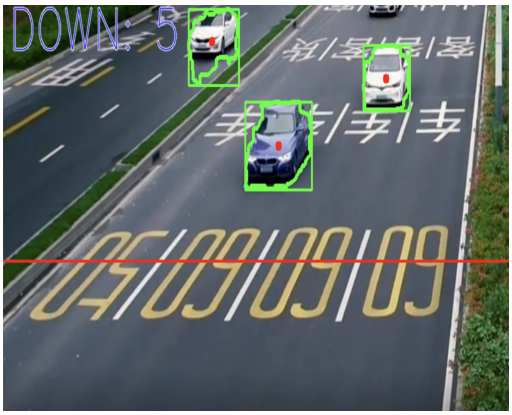

Figure 1.16: counting numbers1

of the function, I adjusted the function's parameters for the online video and obtained slightly inaccurate results, as shown in Figure 1.17. I think it is because the parameters need tuning further.

Figure 1.17: counting numbers2

1.4.3 Evaluation

The experimental results show that if the height of the line is adapted to the angle of the video and the threshold for vehicle determination is suitable for the ratio of vehicle size to video size, the results are accurate. Still, it takes a lot of time and experience to adjust the parameters, which is the drawback of this feature. This feature is susceptible to the angle of the video and the ratio of vehicle size to video size. However, once we get the parameters right, this will be a convenient way to count traffic.

Bibliography

Bolme, D. S., Beveridge, J. R., Draper, B. A., & Lui, Y. M. (2010). Visual object tracking using adaptive correlation filters. In *2010 ieee computer society conference on computer vision and pattern recognition* (pp. 2544–2550).

Bouguet, J.-Y., et al. (2001). Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. *Intel corporation*, *5*(1-10), 4.

Tamgade, S. N., & Bora, V. R. (2009). Notice of violation of ieee publication principles: Motion vector estimation of video image by pyramidal implementation of lucas kanade optical flow. In *2009 second international conference on emerging trends in engineering technology* (p. 914-917). doi: 10.1109/ICETET.2009.154