Computer Vision Coursework

Sparse Optical Flow for Tracking

University of Nottingham COMP3065

Jaesung Park #20121762

Shyjp1@nottingham.edu.cn

Word count: 2303

May 16, 2021

Abstract

This coursework aims to implement the sparse optical flow for tracking and to improve optical flow or tracking results. Through this process, I can get more experience in implementing computer vision programs, plus obtain an understanding of sparse optical flow and computer vision in general. In this project, I built the Lucas-Kanade optical flow program, plus the Gunnar Farneback dense optical flow program to help to understand the strength and weakness of the sparse optical flow. Both implementations were using OpenCV library on major functions like cv2.goodFeaturesToTrack() or cv2.calcOpticalFlowPyrLK. Moreover, there are efforts of improving tracking results, and those extra implementations on sparse optical flow are done without any cv2-library. Since this coursework more aim to implement a sparse optical flow program, this report also aims more to explain sparse optical flow functionalities.

Introduction

Optical flow is the pattern of motion of objects or edges in a scene caused by the relative motion between an observer and a scene [1]. There are two wide-known optical flow methods, Sparse and Dense optical flow. Sparse optical flow gives the flow vectors of corners or edges of an object while Dense optical flow gives the flow vectors of all pixels within a frame. Both methods have their strength and weaknesses, and details of strengths and weaknesses are going to be discussed in a later section.

Followings are the key features/functionalities of this sparse optical flow project:

1. Change the scene into a gray image and find n number of strongest corners in the scene by the Shi-Tomasi method.
2. Within a loop, calculate the distance between the previous scene and the current scene on each point by the sparse optical flow method. Draw a circle on each point and draw

lines on each move.

3. For the additional feature to improve visualization, minimize the number of unnecessary lines.
4. For another feature, build the extra feature that blinds the background scene to visualize only points and lines – obtaining a more clear view
5. Compare result against the Dense optical flow result to view differences in visualization.

Due to size limitation in Moodle, the zip file is stored in school shared box. All implementations and videos can be accessed through the following like:

Primary:https://nottinghamedu1-my.sharepoint.com/:f:/g/personal/shyjp1_nottingham_edu_cn/EixdahyXjfRCn01neCEUkJkBpuX0H62nGLestdr7eIOHiw?e=I4p1GH

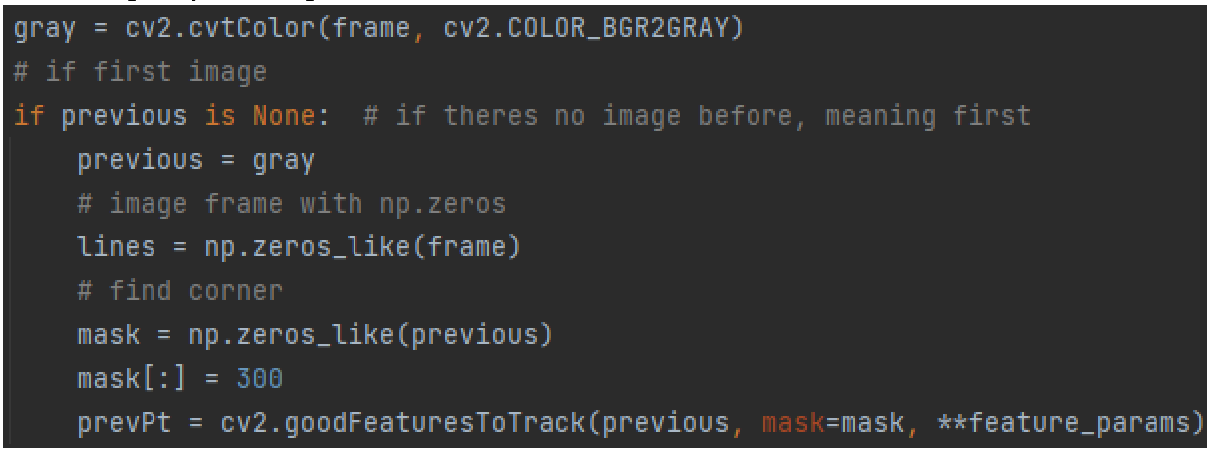
Secondary: https://www.dropbox.com/sh/ztq3gflhdf8qjq3/AABwJtPUChSzyzAaIN7SQz1Pa?dl=0

Implementations

This section specifies the key implementation of both sparse optical flow and an explanation of each feature in detail. Also, it explains the detailed steps of manual implementation, plus the reason why those steps were required.

Shi-Tomasi Corner Detector

To visualize the smooth flow of an object, it is mandatory to find "good" points in each scene. Compared to the similar corner detector, the Harris corner detector, the Shi-Tomasi corner detector is widely known for the advanced version of the Harris detector. In this project, a Shi-Tomasi corner detector was used to find good points by goodFeaturesToTrack from OpenCV. Simply, this function determines strong corners (good points) on a scene. Through this function, I could specify some parameters such as the maximum number of corners, the minimal



```
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
# if first image
if previous is None:  # if theres no image before, meaning first
    previous = gray
    # image frame with np.zeros
    lines = np.zeros_like(frame)
    # find corner
    mask = np.zeros_like(previous)
    mask[:] = 300
    prevPt = cv2.goodFeaturesToTrack(previous, mask=mask, **feature_params)
```

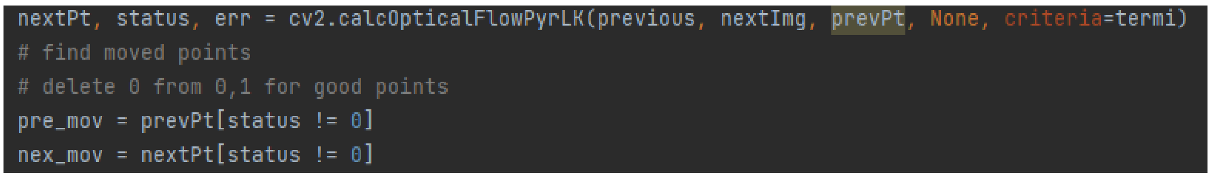
accepted quality of image corners, or minimum Euclidean distance between the corners.

Figure 1. Implementation of corner detection with OpenCV function. The parameter for feature_params are as follows: maxCorners=30, qualityLevel=0.01, minDistance=10, blockSize=8.

After a number of self-testing, the maximum number of corners let the scene include some unnecessary points. By limiting the number of corners, the program only works with reasonable points and it allows the viewer to focus on the main object. Testing result done with the corner detector will be discussed in the result section.

Lucas-Kanade Optical Flow Calculation

This step is to calculate the optical flow of each point in the two-consecutive scene. The calcOpticalFlowPyrLK() function from OpenCV is used and this algorithm calculates an optical flow for a sparse feature set using the iterative Lucas-Kanade method with pyramids. This function takes an input of the image, second input image, vector of 2D points containing the calculated new positions of input features in the second image, a parameter for specifying termination criteria of the search algorithm, and more, to find the flow between two points.



```
nextPt, status, err = cv2.calcOpticalFlowPyrLK(previous, nextImg, prevPt, None, criteria=termi)
# find moved points
# delete 0 from 0,1 for good points
pre_mov = prevPt[status != 0]
nex_mov = nextPt[status != 0]
```

Figure 2. Implementation of calculating optical flow. The parameter for termination criteria is as follows: TERM_CRITERIA_EPS | TERM_CRITERIA_COUNT, 15, 0.08.

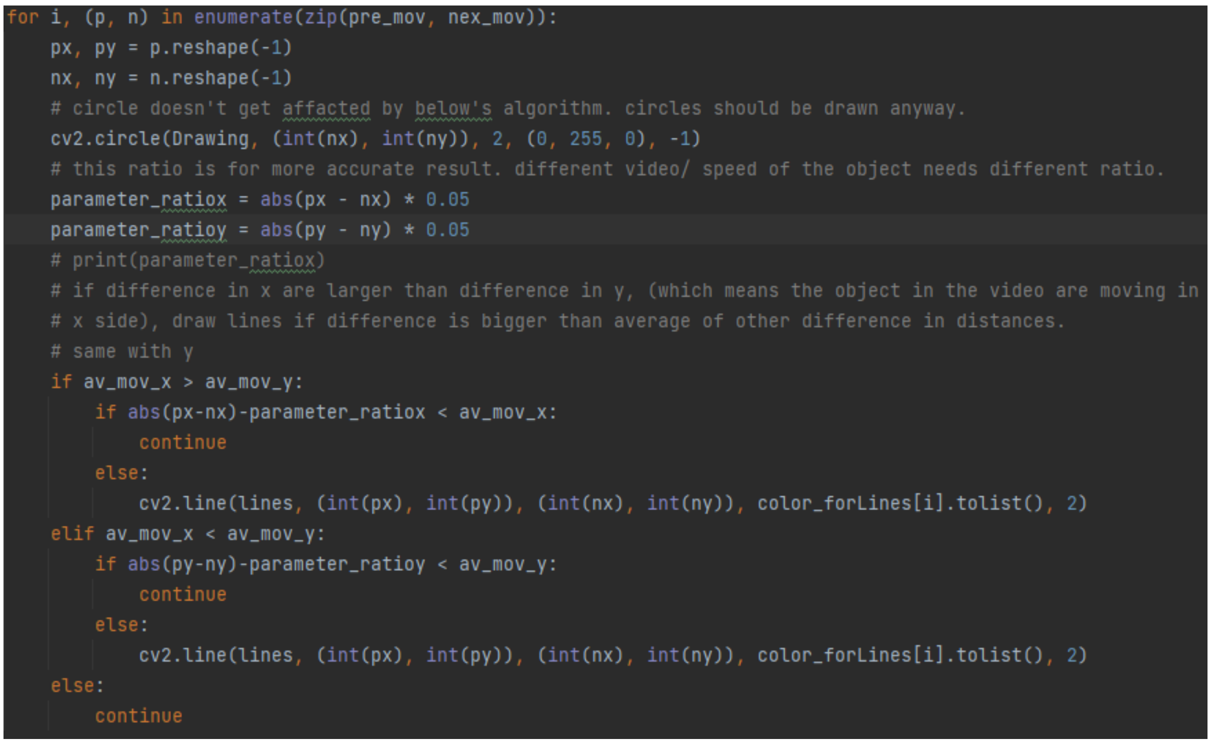
After calculation, it only picks for good points where status is 1. Unlike the dense optical flow algorithm, it only calculates with points that are obtained from corner detection.

Eliminating unnecessary lines in visualization

After plotting circles and drawing lines on the scene, the result was not similar to what I was expected. There was a great number of unnecessary “good” points and flows. By limiting the number of corners on the corner detection method, it could handle the unnecessary points, but the lines that are placed in the un-wanted area could not be handled with the OpenCV functions, as shown in Figure 5. For example, if the corner detector returns a point on an empty wall as a good point, the optical calculating function counts that point as an important point. If the video recording wasn’t recorded in a stable condition, the screen of the recording is constantly moving, and the optical calculating function counts that movement of the camera moving as points’ moving, then it leads to the result that containing other unnecessary flows. It could also happen in a real situation where the program is meant to find flows only on moving car, but rather it also contains a flow of moving bicycle. This situation also can be handled if we choose objects/points that we want to track in the first place, but in this coursework, I manage to implement to limit the flows/lines.

The main idea of eliminating unnecessary flows are as follows:

1. Place circles on “good” points that we get from the detection method.
2. In each scene of a video, find the sum and average of all “good” points location (x, y) for both the x and y-axis.
3. Get the average distance of all flows in a scene for both x and y-axis between two points.
4. Also, get the distance of the current two points.
5. Compare results from Step 2 and Step 3 to determine if the current flow is an outlier, relative to all flows in a scene.
6. Draw lines on which are not an outlier.



```
for i, (p, n) in enumerate(zip(pre_mov, nex_mov)):
    px, py = p.reshape(-1)
    nx, ny = n.reshape(-1)
    # circle doesn't get affacted by below's algorithm. circles should be drawn anyway.
    cv2.circle(Drawing, (int(nx), int(ny)), 2, (0, 255, 0), -1)
    # this ratio is for more accurate result. different video/ speed of the object needs different ratio.
    parameter_ratiox = abs(px - nx) * 0.05
    parameter_ratioy = abs(py - ny) * 0.05
    # print(parameter_ratiox)
    # if difference in x are larger than difference in y, (which means the object in the video are moving in
    # x side), draw lines if difference is bigger than average of other difference in distances.
    # same with y
    if av_mov_x > av_mov_y:
        if abs(px-nx)-parameter_ratiox < av_mov_x:
            continue
        else:
            cv2.line(lines, (int(px), int(py)), (int(nx), int(ny)), color_forLines[i].tolist(), 2)
    elif av_mov_x < av_mov_y:
        if abs(py-ny)-parameter_ratioy < av_mov_y:
            continue
        else:
            cv2.line(lines, (int(px), int(py)), (int(nx), int(ny)), color_forLines[i].tolist(), 2)
    else:
        continue
```

Figure 3. The part of the implementation of eliminating unnecessary flows.

Figure 3 is only a part of the implementation of this topic. As shown in Figure 3, it counts on both the x and y-axis. If the average movement/distance of all flows on the x-axis are greater than on the y-axis, it means the object, or the camera is moving horizontally. Then, if the current flow movement is too small relative to other flows in the scene, it omits to draw the flow. Also, there are ratios that control how much of small distances should be omitted (parameter_ratiox and paramter_ratioy in Figure 3). If the ratio is too low, it will draw every flow and if it's too high, it will affect the flows of the main object that we are looking for. The different results with different ratios will be discussed in a later section.

Omitting the background

This own method comes from the idea of OpenCV's background subtraction methods with the desire of wanting to only focus on an object. It does the same work but much simpler, only in this project. Basically, if the user clicks the key 'b' while running the program, the background image is replaced to NumPy array with full of zeros, which is shown as a black background. In this way, the user can view only points and lines as shown in Figure 7.

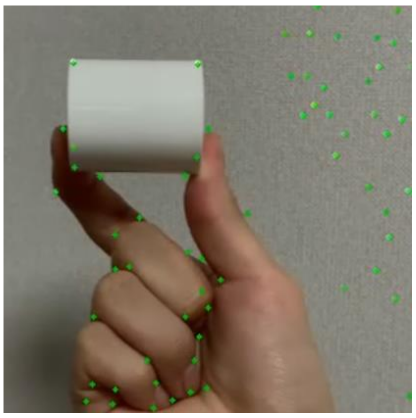
There is one more similar approach, which deletes lines/flows from the previous scenes. If a user clicks the key 'd', it renews the previous image so that the program deletes the past record of those flows as shown in Figure 7.

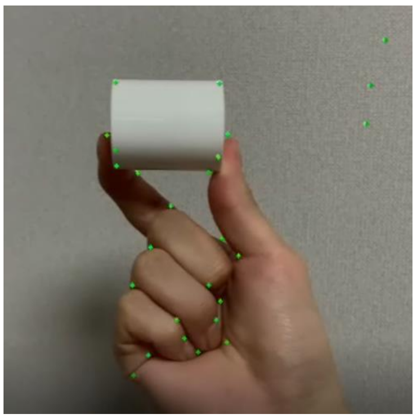
Results

This section gives results from different techniques and methods. Plus, comparison between different ratio/parameter settings to improve the tracking results.

Shi-Tomasi Corner Detector

As stated above, the goodFeaturesToTrack() function's maxCorners is being used as an approach to eliminate unnecessary good points. The below figures show how a large number of maxCorners are pointing to unnecessary points in this specific project.



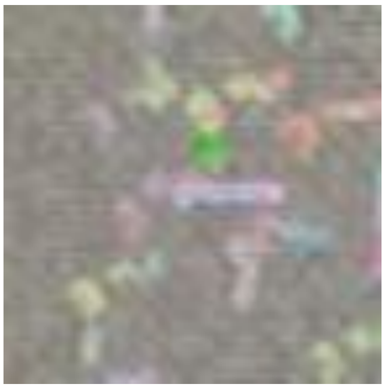


maxCorners = 100 maxCorners = 30

Figure 4. images of how results are different by the number of maxCorners.

If we look at the top right corner of the left image, there are a much greater number of unnecessary points, compared to the right image.

Eliminating unnecessary lines in visualization


Original output

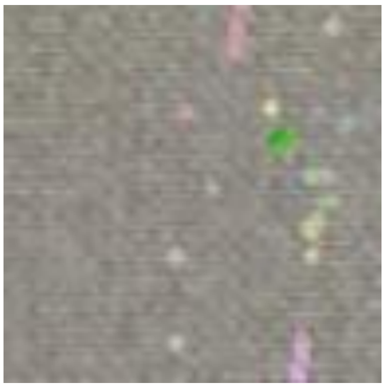
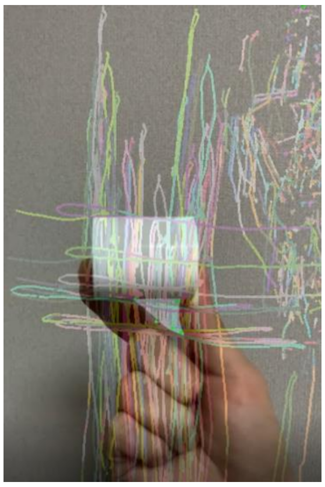
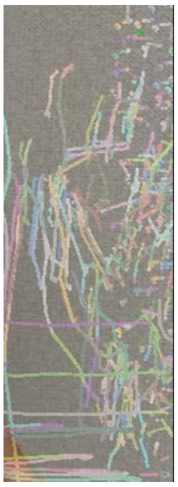

After applying algorithms

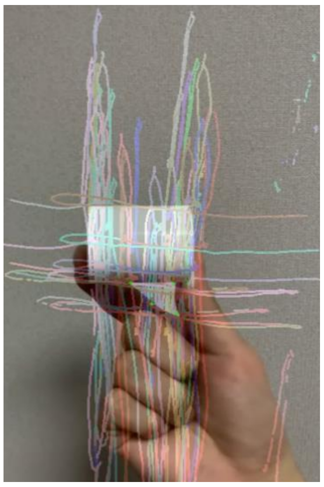
Figure 5. Comparison between before and after the elimination of unnecessary lines algorithm is applied.

Figure 5 shows the comparison between before and after the elimination of flows. On the left image, flows are clearly shown on the wall, in contrast, on the right image, a lot of flows are skipped, and each flow is not consecutively connected to each other. Thus, it does eliminate some unnecessary flows.

By applying both limiting maxCorners and eliminating flows, the result as follows:


Original Output


Flows in-wall


After Eliminating

Flows in wall

Figure 6. Images of the result before the improvement and after.

The original output (left two images) is with 100 good points and the right image is with 30 points and an elimination algorithm applied. In fact, there is a limitation to completely eliminate all unnecessary flows with my algorithm, because if I apply for a strict number on the ratio, it also affects the main object's flow. Those flows remaining on the right image's wall are the ones coming from when the camera is moved tough enough to have a similar moving distance as my hand move.

Omitting the background

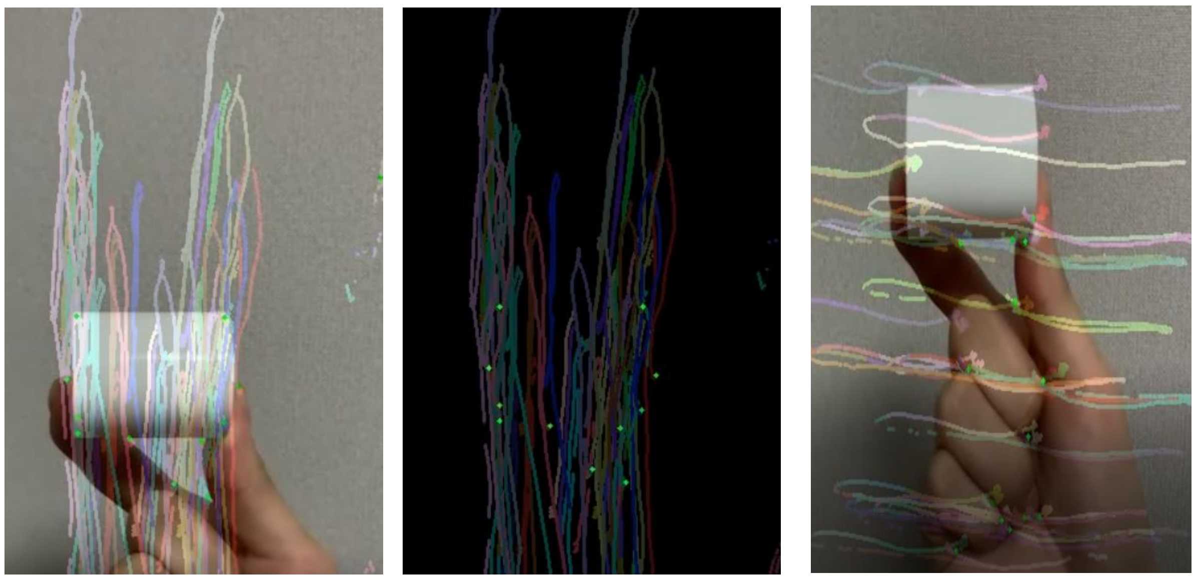
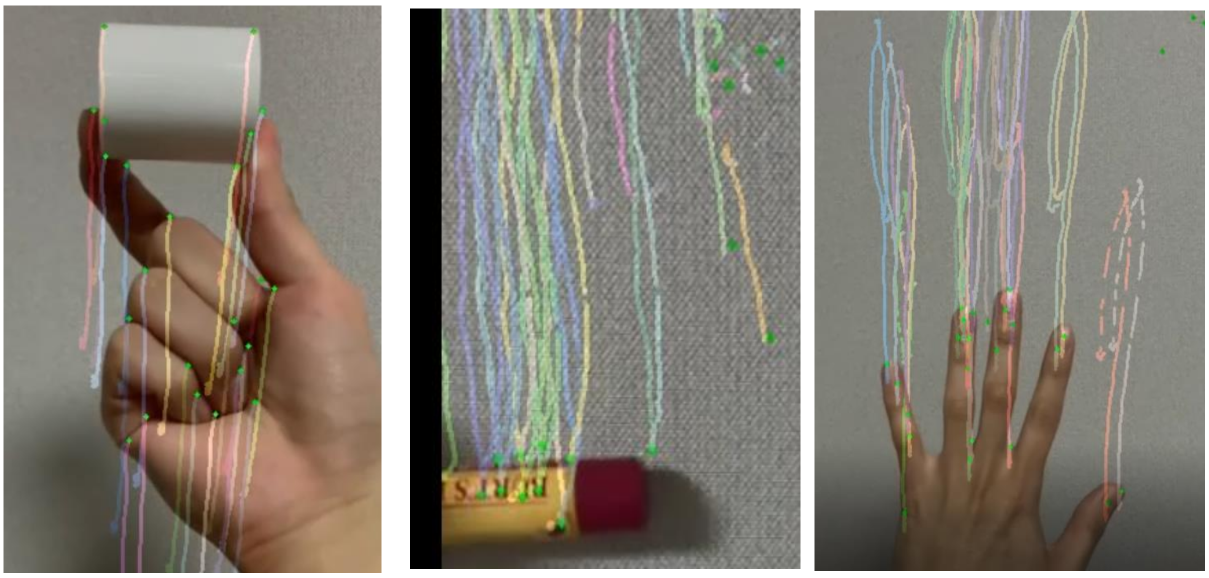

Original
Omitting background (press 'b')
Removing past flows (press 'd')

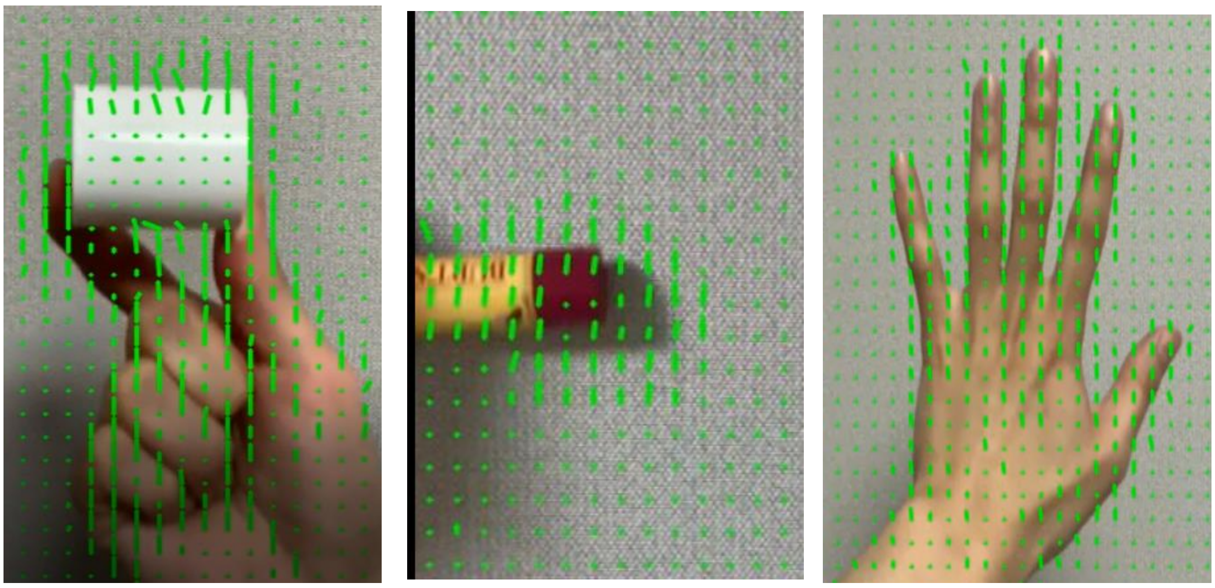
Figure 7. result of omitting background and removing past lines.

As mentioned above, this step shows a more clear view of points and tracking results. The second image in Figure 7 shows how eliminating background gives a more clear view and the third image shows only the horizontal movement where, originally, it should have had both vertical and horizontal lines. Refer to the code for better understanding.

Comparison with the Dense Optical Flow



Sparse Optical Flow



Dense Optical Flow

Figure 8. Sparse optical flow vs. Dense optical flow.

The dense optical flow program is implemented additionally to find out more about optical flow. The first thing noticed is since the dense optical flow calculates all pixels on the scene, it is more accurate than Sparse optical flow. The downside of calculating all pixels is that the program itself is slower than the Sparse program. The dense optical flow program uses calcOpticalFlowFarneback() function from OpenCV. The explanation of the implementation of the dense optical flow program does not include in this report since it is not purely related to this topic. It has a similar implementation with the sparse program except for the detection method. For the detail of the implementation of dense optical flow, refer to the code.

Discussion

This section discusses the strengths and weaknesses of the program, in addition to sparse optical flow in general. Most of the discussions are handled from the result section and here are the main points of strengths and weaknesses.

Strengths

- Fast and relatively accurate result.

  Compared to the dense optical flow program, the sparse program runs smoothly and still output reasonable results. It’s because the sparse optical flow provides features like edges or corners when Dense optical flow is designed to work on all pixels. When sparse optical flow only calculates the flow with those interesting points, Dense calculates all pixels in each scene.

- The program eliminates unnecessary lines.

  The program has its own implementation of eliminating unnecessary lines. It gives a more accurate view of a result. Moreover, it provides background subtraction method and deleting previous flows method to give a movement-focused view.

Weaknesses

- The program is still weak to omit unnecessary points.

  Although my program has the additional feature of eliminating points/lines on each scene, it is still weak to outliers in both choosing good points and lines. It is also one of the weaknesses of optical flow since it counts the point caused by lighting changes without any actual motion.

- Accuracy of frame differencing depends on object speed and frame rate

  One another weakness is that if the movement speed of an object is too fast, it is unable to follow the tracking. It is because the program is not capable of looking for a specific next point when the distance between points from the old and new scene is far away from each other [2]. For example, if good point A from the old scene moved long distance in a new scene, located near Point B in the new scene. The program is unable to decide where the new point came from. One of the known ways to solve this issue is to reduce the resolution of scenes/video and then apply the sparse method[3].

- Movement of two objects moving in different speed causes ambiguity.

Since there is an extra parameter ratio like shown in Figure 3, to obtain a more accurate result on each input(video), a tuning process is required. The problem arises when there is more than one object which is moving at a different speed. Since my eliminating lines implementation is using the ratio of the average distance on all good points, it is not accurate to use the same ratio on different speed objects.

References

[1] Burton, Andrew, Radford, John. 1978. Thinking in Perspective: Critical Essays in the Study of Thought Processes. Routledge.

[2] Bruce D Lucas, 1984 Generalized Image Matching by the Method of Differences.

[3] J. Y. Bonguet. 2001. Pyramidal Implementation of the Affine Lucas Kanade Feature Tracker Description of the Algorithm. Intel Corporation, 5.