CSE578 Computer Vision

Assignment 5 : Optical Flow

Karnik Ram
2018701007

The procedure for implementing the Lucas-Kanade algorithm for estimating optical flow, along with the code is explained in this report. Observations, and some possible improvements to the algorithm are then discussed. Motion segmentation and object tracking using optical flow are also discussed at the end.

The code files are provided in the `src` directory along with this submission. The code is written in C++ and is written as a class (`OpticalFlow`) for modularity. To run the programs,

```
mkdir build && cd build
cmake ..
make
./run_of <image1-path> <image2-path>
./run_motion_seg <video-path>
./run_tracking <video_path>
# eg: ./run_of ../data/eval-data-gray/Basketball/frame10.png ../data/eval-data-gray/
    Basketball/frame11.png
```

1 Lucas-Kanade Optical Flow

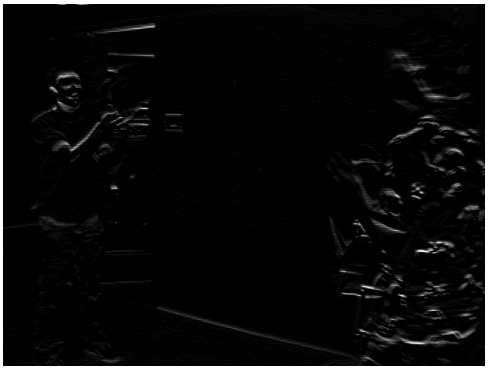
As we know the brightness constancy constraint equation - $\nabla I.(v_x, v_y)^T$ is under-determined and we need additional constraints to solve for the motion $(v_x, v_y)^T$. Lucas and Kanade proposed a local smoothness constraint, where we assume all nearby pixels have the same motion. For instance, if we consider a 3×3 window around a pixel, we get 9 equations for the same unknown $(v_x, v_y)^T$, which makes it an over-constrained system. We then solve for the motion using least-squares.

1.1 Spatio-temporal image gradients

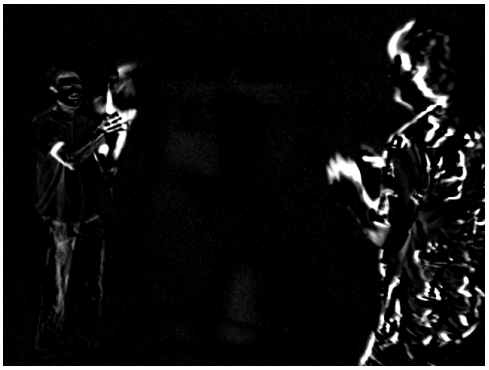
The algorithm begins by computing the I_x, I_y, I_t gradients between the two input images. This is done by applying gradients kernels across the images using OpenCV's `filter2D` function.

```
void OpticalFlow::computeImageGradients()
{
    // img_x
    cv::Mat kernel = cv::Mat::ones(2, 2, CV_32FC1);
    kernel.at<float>(0, 0) = -1.0;
    kernel.at<float>(1, 0) = -1.0;

    cv::Mat dst1, dst2;
    filter2D(img1, dst1, -1, kernel);
    filter2D(img2, dst2, -1, kernel);

    img_x = dst1 + dst2;

    // img_y
    kernel = cv::Mat::ones(2, 2, CV_32FC1);
    kernel.at<float>(0, 0) = -1.0;
    kernel.at<float>(0, 1) = -1.0;

    filter2D(img1, dst1, -1, kernel);
    filter2D(img2, dst2, -1, kernel);

    img_y = dst1 + dst2;

    // img_t
    kernel = cv::Mat::ones(2, 2, CV_32FC1);
    kernel = kernel.mul(-1);

    filter2D(img1, dst1, -1, kernel);
    kernel = kernel.mul(-1);
    filter2D(img2, dst2, -1, kernel);

    img_t = dst1 + dst2;
}

```

(a) I_x



(b) I_y



(c) I_t

1.2 Solving for v_x, v_y

Once the gradients are estimated, we form our linear system of equations of the form $Av = b$, where

$$A = \begin{bmatrix} I_x(q_1) & I_y(q_1) \\ I_x(q_2) & I_y(q_2) \\ \vdots & \vdots \\ I_x(q_n) & I_y(q_n) \end{bmatrix} \qquad v = \begin{bmatrix} V_x \\ V_y \end{bmatrix} \qquad b = \begin{bmatrix} -I_t(q_1) \\ -I_t(q_2) \\ \vdots \\ -I_t(q_n) \end{bmatrix}$$

Applying least-squares, we get

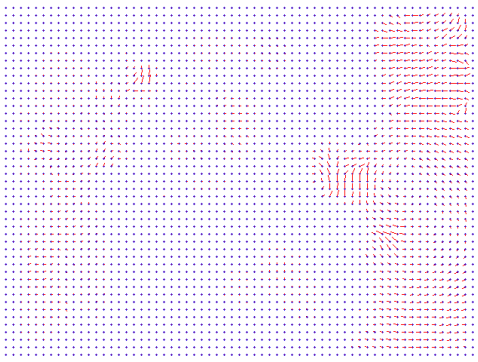
$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} \sum_i I_x(q_i)^2 & \sum_i I_x(q_i)I_y(q_i) \\ \sum_i I_y(q_i)I_x(q_i) & \sum_i I_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_x(q_i)I_t(q_i) \\ -\sum_i I_y(q_i)I_t(q_i) \end{bmatrix}$$

In code, this is implemented as follows:

```

std::vector<float> OpticalFlow::getWindow(const cv::Mat &mat, const int &i, const int &j
    )
{
    std::vector<float> window;
    for(size_t ii = i - window_size/2; ii <= i + window_size/2; ii++)
        for(size_t jj = j - window_size/2; jj <= j + window_size/2; jj++)
        {
            window.push_back(mat.at<float>(ii, jj));
        }

    return window;
}

cv::Mat OpticalFlow::sumOverWindow(const cv::Mat &mat)
{
    cv::Mat sum = cv::Mat::zeros(mat.size(), CV_32FC1);

    for(int i = window_size/2; i < mat.rows - window_size/2; i++)
        for(int j = window_size/2; j < mat.cols - window_size/2; j++)
        {
            std::vector<float> window = getWindow(mat, i, j);
            sum.at<float>(i, j) = std::accumulate(window.begin(), window.end(), 0.0);
        }
```

```
24
25      return sum;
26  }
27  void OpticalFlow::computeOpticalFlow(cv::Mat &u, cv::Mat &v)
28  {
29      computeImageGradients();
30
31      cv::Mat img_xx = img_x.mul(img_x);
32      cv::Mat img_yy = img_y.mul(img_y);
33      cv::Mat img_xy = img_x.mul(img_y);
34      cv::Mat img_xt = img_x.mul(img_t);
35      cv::Mat img_yt = img_y.mul(img_t);
36
37      cv::Mat sum_img_xx = sumOverWindow(img_xx);
38      cv::Mat sum_img_yy = sumOverWindow(img_yy);
39      cv::Mat sum_img_xy = sumOverWindow(img_xy);
40      cv::Mat sum_img_xt = sumOverWindow(img_xt);
41      cv::Mat sum_img_yt = sumOverWindow(img_yt);
42
43      cv::Mat tmp = sum_img_xx.mul(sum_img_yy) - sum_img_xy.mul(sum_img_xy);
44      u = sum_img_xy.mul(sum_img_yt) - sum_img_yy.mul(sum_img_xt);
45      v = sum_img_xt.mul(sum_img_xy) - sum_img_xx.mul(sum_img_yt);
46
47      cv::divide(u, tmp, u);
48      cv::divide(v, tmp, v);
49  }
50
```

1.3 Results



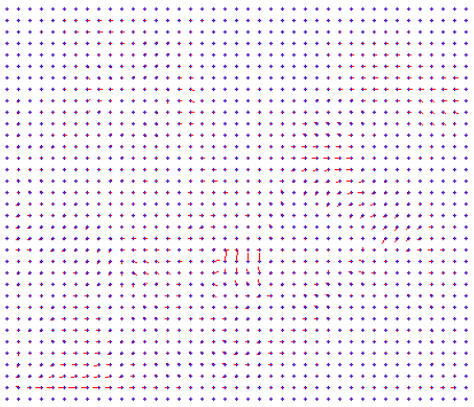
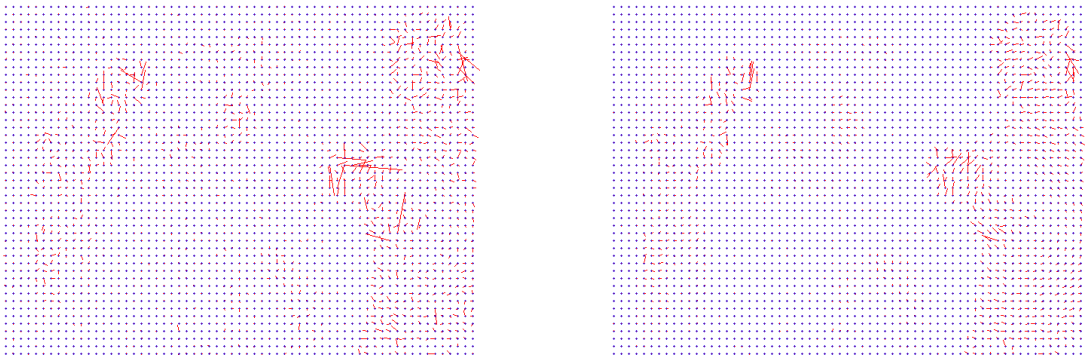


Figure 2: Blue dots indicate intial location, red lines indicate motion. The images (in order) are from the `Basketball, Teddy, Backyard, Evergreen, and Schefflera` sequences respectively.

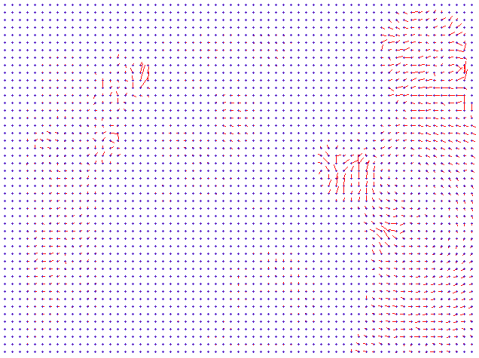
1.4 Observations

Window size

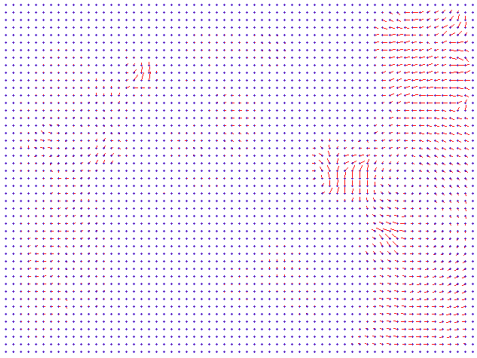
As the window size increases, the flow estimates are more consistent. For 3×3 size, the flow estimates are most noisy where as for 31×31 size they are most consistent.



(a) 3×3 (b) 7×7



(c) 15×15



(d) 31×31

Figure 3: Effect of different window sizes on the `Basketball` sequence.

As expected, the run time increases by a factor of almost 3 when the window size is doubled.

Size	Time elapsed
3×3	2.17 s
7×7	4.74 s
15×15	12.21 s
31×31	38.50 s

Sparse optical flow

Instead of computing optical flow for every pixel in the image, we can compute the flow only for "good" features. These are features for which the second moment matrix has large and relatively similar eigen values. This is the basis of the Kanade-Lucas-Tomasi feature tracker. However this could not be implemented completely for this assignment.

Iterative LK, Hierarchial LK

LK optical flow is based on the fundamental assumption that motion is small. However this need not be the case since motion can be large and nonlinear. To handle such scenarios, an iterative version of LK can be applied. LK is also susceptible to local minima, which can be handled by applying a hiearchial (coarse-to-fine) approach.

Camera motion

Optical flow is the relative motion between the objects in the scene and the camera, so when there is camera motion it appears as flow as well. The `Backyard` sequence is one example where there's camera motion. This is undesireable in motion detection applications, while desirable in image stabilization applications.

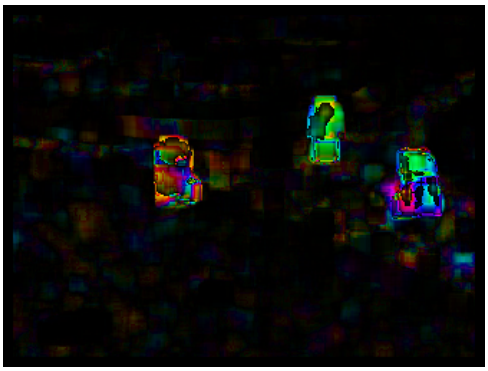
2 Motion segmentation

After optical flow computation, motion segmentation can be carried out by thresholding and then separating the optical flow vectors based on magnitude and direction. One way to do this is as follows,

```
void drawMotionSeg(cv::Mat &u, cv::Mat &v)
{
    cv::Mat mag, angle;
    cv::cartToPolar(u, v, mag, angle);

    for(int i = 0; i < u.rows; i++)
        for(int j = 0; j < u.cols; j++)
        {
            if(u.at<float>(i, j) < 50)
                u.at<float>(i, j) = 0;

            else if(v.at<float>(i, j) < 50)
                v.at<float>(i, j) = 0;
        }

    cv::Mat hsv = cv::Mat::zeros(u.size(), CV_8UC3);

    for(int i = 0; i < u.rows; i++)
        for(int j = 0; j < v.cols; j++)
        {
```

```
            cv::Vec3b value;
            value[0] = 2 * angle.at<float>(i, j) * 180 / 3.14;
            value[1] = 255;
            value[2] = mag.at<float>(i, j) * 255;
            hsv.at<cv::Vec3b>(i, j) = value;
        }

    cv::Mat bgr;
    cv::cvtColor(hsv, bgr, CV_HSV2BGR);
    cv::namedWindow("motion", CV_WINDOW_NORMAL);
    cv::imshow("motion", bgr);
    cv::waitKey(0);
}
```

(a) Input frame



(b) Motion segmentation

3 Object tracking

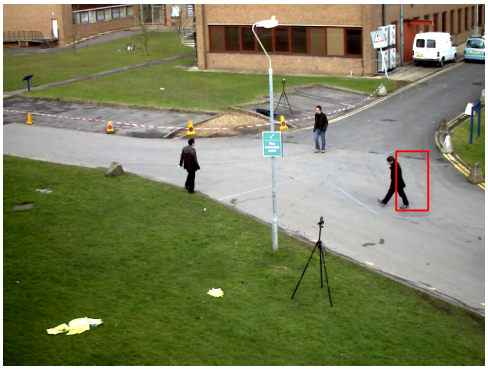
For object tracking, an object of interest is first selected by the user by drawing a bounding box. Then the mean flow within the bounding box is used to update the location of the box in the subsequent frames. However this approach failed to work reliably and the object was lost track of after 6 frames as shown.

(a) First frame with user selection

(b) Track 1

(c) Track 2

(d) Track 3



(e) Track 4

(f) Track 5

(g) Track 6

(h) Track 7

Some immediate improvements that can be tried next - use median instead of mean flow since the mean flow is almost zero (but computing median is expensive), use iterative LK to deal with nonlinear motion, and use hierarchial LK to handle large motion.