Optical flow for estimating velocity

Carolyn Holz (cjholz), Sachit Saksena (sachit)

December 12, 2018

1 Introduction

Optical flow is a powerful method useful for a wide range of tasks in machine vision. Popular implementations of optical flow include facial tracking and motion detection. In the medical field, optical flow can be used to find edges in ultrasound for understanding architecture of organs and tissues. Here we present a simple implementation of optical flow for estimating velocity of objects moving past a stationary camera.

2 Methods

2.1 Horn-Schunk optical flow

The Horn-Schunk method for optical flow is the main algorithm we worked with this semester. The method we substantively explored was a modified version that assumes constant velocity across an entire image or frame. Either way, the method begins with the constant brightness assumption as follows:

$$\frac{d}{dt}E(x,y,t) = 0$$

From a Taylor expansion of this equation with, we can derive the familiar brightness change constraint equation:

$$uE_x + vE_y + E_t = 0$$

where,

$$u = \frac{dx}{dt}$$

$$v = \frac{dy}{dt}$$

Which can be solved to estimate velocities in the image.

In the Horn method, these derivatives are numerically estimated using first differences in the x, y, and t direction. Below is an example from class in the x-direction:

$$Ex_{i,j,k} = \frac{1}{\epsilon_x}(\frac{1}{4}(E_{i,j+1,k}+E_{i+1,j+1,k}+E_{i,j+1,k+1}+E_{i+1,j+1,k+1})-\frac{1}{4}(E_{i,j,k}+E_{i+1,j,k}+E_{i+1,j,k+1})$$

With the Horn method, a least squares approach can be used across the whole image where each derivative can be estimated. The least squares approach can be formulated as:

$$\sum_{i=1}^{n-1}\sum_{i=1}^{m-1}(uE_x + vE_y + E_t)^2 dxdy$$

And u and v can now be estimated by minimizing this equation.

2.2 Harris Corner Detection

To detect objects in an image, we used a method known as Harris Corner Detection. This method is highly robust due to invariance to rotation, scale, image noise, and image brightness. Using a Gaussian window given by:

$$w(x, y)$$

Harris corner detector algorithm checks a shift in image intensity in multiple directions within that window using the following equation:

$$I(u, v) = \sum_{x,y} w(x, y)[E(x + \delta u, y + \delta v) - E(x, y)]$$

From this, a least squares problem can be set up in which a 2x2 matrix represents the intensity structure of each Gaussian window. If both eigenvalues of this matrix are small, this indicates constant brightness (neither corner or edge). If one is high and one is low, this indicates constant brightness in one direction (an edge). If both are high, this indicates different brightness in all directions, or a corner.

2.3 Lucas-Kanade Optical Flow

The Lucas-Kanade algorithm is similar to the algorithm discussed in class in that it uses the brightness change constraint equation to determine velocity in the x and y directions. However, it does not assume the same flow over the entire image frame. Instead, it assumes similar flow for a window size around a particular point or points provided as input to the algorithm. It optimizes output for this window size using least squares. We chose this method for our problem both due to performance considerations on Android as well as it's suitability for our particular problem. Since we wish to detect the velocity of an object moving through a constant background determining optical flow for a portion of the image is appropriate.[3]. The window size is given by:

$$I_x(x + \Delta x, y + \Delta y) \cdot u + I_y(x + \Delta x, y + \Delta y) \cdot v = -I_t(x + \Delta x, y + \Delta y)$$
$$\text{for } \Delta x = -1, 0, 1 \text{ and } \Delta y = -1, 0, 1$$

Using a least squares approach with a matrix representing these windows can solve the brightness constraint equation for u and v.

3 Implementation

3.1 Android SDK and OpenCV

We chose to implement optical flow in Android since the camera callbacks allow us to easily access subsequent frames of video. We used a minimum SDK of 21 and a target SDK of 27. Our experiments and primary testing were done on a Google Pixel 1 running Android Pi. We included OpenCV 3.4.3 as a dependency in order to use their algorithm implementations for parts of the project.

3.2 Horn-Schunk implementation

We began using the ViewFinder code provided on Stellar and implemented the Horn-Schunck optical flow algorithm discussed in class (as shown in "opticalflow1.zip". We had anticipated performance issues with this methods since it calculates optical flow over the whole image. We found that calculating optical flow for each pixel slowed the frame rate such that the video was not coherent and we could not see useful output. Essentially, the computations could not occur fast enough to give real time optical flow readouts. We tried a couple different methods for decreasing computational time: converting the image to gray-scale and lowering resolution by increasing the number of pixels between calculation. Eventually, we decided to pivot and try a more targeted approach.

```
public void calculateBrightnessGradients() {
    if (mLastData == null || mNewData == null) {
        return;
    }
    float epx = getResources().getDisplayMetrics().xdpi;
    for (int i = 0; i < mLastData.length-1; i++) {
        for (int j = 0; j < mLastData[i].length-1; j++) {
            mEx[i][j] = (1/epx)*(.25*(mLastData[i][j+1] + mLastData[i+1][j+1] + mNewData[i][j+1] + mNewD
                                .25*(mLastData[i][j] + mLastData[i+1][j] + mNewData[i][j] + mNewData[i+1
            mEy[i][j] = (1/epx)*(.25*(mLastData[i+1][j] + mLastData[i+1][j+1] + mNewData[i+1][j] + mNewD
                                .25*(mLastData[i][j] + mLastData[i][j+1] + mNewData[i][j] + mNewData[i][
            mEt[i][j] = (1/epx)*(.25*(mNewData[i][j] + mNewData[i][j+1] + mNewData[i+1][j] + mNewData[i+
                                .25*(mLastData[i][j] + mLastData[i][j+1] + mLastData[i+1][j] + mLastData
            if (mEt[i][j] != 0) {
                int l=9;
            }
        }
    }
}
```

Figure 1: Estimating brightness gradients across image

```
// Let's put a function here that converts the rgb pixel matrix into a greyscale intensity matrix
// Or let's do a function that can extract the Y values of the YUV420SP object
// does this already do this? How do I check the output??
public void decodeYUV420SPGrayscale (int[][] rgb, byte[] yuv420sp, int width, int height) { // extract grey RGB format image
    final int frameSize = width * height;
    mLastData = mNewData.clone();
    // This is much simpler since we can ignore the u and v components
    for (int i = 0; i < height; i++) {
        for (int j=0; j < width; j++) {
            int y = (0xFF & ((int) yuv420sp[(i*width)+j])) - 16;
            if (y < 0) y = 0;
            if (y > 0xFF) y = 0xFF;
            rgb[i][j] = y;
        }
    }
}
```

Figure 2: Calculating U

3.3 Feature selection

We used on OpenCV implementation of the above mentioned Harris corner detection method for identifying features (GFFT detector). For this application, this works well, since we just need a cloud of points for a given object in the frame. This is because we are only interested in isolating important objects in the foreground of the frame. For parameter tuning, we detected 100 points in every frame and we made the minimum pixel distance between points detected to be 5 pixels in order to ensure that corners and edges of similar objects are clustered together (Figure 1).

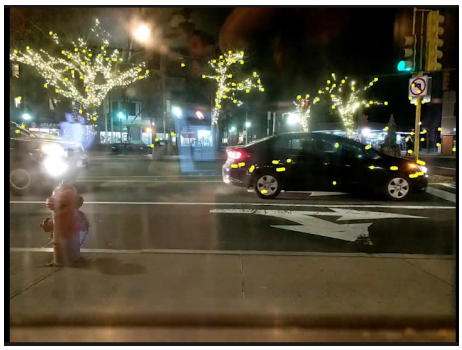

Figure 3: Yellow points represent detected features

3.4 Lucas-Kanade in OpenCV

We used the OpenCV implementation of Lucas-Kanade with the previous and current camera frames as input along with the matrix of feature points calculated from the previous frame in section 3.3 and a window size of 21. The output of the algorithm is a matrix of points in the current frame which correspond to the input features. Since the algorithm does not output velocities we must calculate them as described in section 3.5.[6].

3.5 Calculating Velocity

Lucas-Kanade returns optical flow output for several input points. Therefore we must determine from the output which point or points belong to our pedestrian. We do this by determining the largest flow vector and calculating velocity from that vector. Below is an example of the velocities calculated during an experiment in which the subject ran across the screen. The arrows represent maximum displacement, which is used to calculate velocity from the largest flow vector.

In order to determine the most likely feature point and corresponding optical flow output point that belong to our pedestrian we calculate the largest difference between feature and output point. This means we calculate a displacement vector for each pair of points and determine the vector with the largest magnitude. The x and y values of this vector then represent the x and y displacement in pixels for our object. We divide each displacement by the time between frames to determine u and v in units of pixels per second. We then use our measured width of view in the real world to calculate the real world distance reflected by a pixel and multiply this meter per pixel value by the pixel per second velocity to get meter per second velocity.

To account for some noise in optical flow measurement we average the velocity over a sliding window of the previous five velocities.

Figure 4: Initial acceleration

Figure 5: Running motion

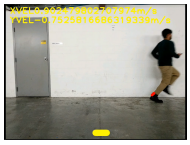

Figure 6: Deceleration

Figure 7: Bike velocity detected

Figure 8: Car velocity detected

Further, we can show that the application can successfully pick the fastest moving object when multiple are present in the frame. In the first image below (taken at Main St. and Ames St.), the bicycle is moving faster than the car, since it has a pedestrian walking sign, while the car is decelerating to a stop at the Ames St. traffic light. For this reason, the bicycle speed is detected (4.5 m/s). Once the bike leaves the frame of the image, the velocity of the decelerating car is detected.

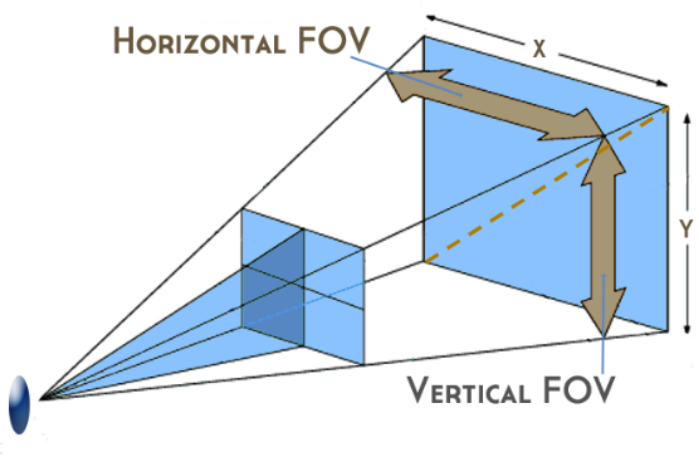

Figure 9: Determining field of view

3.6 Measuring Pixel Width

To convert pixels per second velocity into meters per second velocity we need to know the real world height and width of a pixel (Figure 7) in the image or

$$pixelHeight = \frac{realY}{ypixels}$$

$$pixelWidth = \frac{realX}{xpixels}$$

To do this we calculated the real world diagonal length shown in the image from the FOV angle θ of the camera and depth of the object being tracked which we measured by hand.

$$realDiagonal = 2 * tan(\frac{\theta}{2}) * depth$$

We then calculated the angle of the diagonal with the baseline ϕ of the image from

$$tan(\phi) = \frac{xpixels}{ypixels}$$

Finally, we calculated the real world x and y distances captured in the image with

$$sin(\phi) * realDiagonal = realY$$

$$cos(\phi) * realDiagonal = realX$$

.

3.7 On Screen Output

Our app outputs two visual indicators of optical flow. We display the calculated u and v values in meters per sec at the top right of the video frame. We also

display the largest displacement vector output by the Lucas Kanade algorithm as an arrow vector from the feature point to the output point. In addition, there is a simple bar depiction of velocity in the direction of motion (positive/negative).

3.8 Experiment Setup

In order to validate the velocity measurements from our optical flow estimation, we set up the following experiment.

1. Set up stationary camera facing evenly lit background
2. Measure distance to wall and estimate field of view
3. Measure distance for traveler walking
4. Carry out trials, recording time elapsed walking
5. Log maximum velocity measured from application for each trial

   Measure true speed using wearable device (phone)

Table 1: Add caption

Time	Distance	Velocity	Estimated Velocity	Error	Squared error
4.68	6.55	1.39957265	1.45	-0.0504274	0.00254292
4.78	6.55	1.37029289	1.42	-0.0497071	0.0024708
5.25	6.55	1.24761905	1.02	0.22761905	0.05181043
5.8	6.55	1.12931034	1.34	-0.2106897	0.04439013
6.6	6.55	0.99242424	0.9	0.09242424	0.00854224
3.64	6.55	1.79945055	1.94	-0.1405495	0.01975415
3.96	6.55	1.6540404	1.82	-0.1659596	0.02754259
4	6.55	1.6375	1.4	0.2375	0.05640625
4.13	6.55	1.58595642	1.5	0.08595642	0.00738851
4.31	6.55	1.51972158	3.2	-1.6802784	2.82333558
2.43	6.55	2.69547325	3.4	-0.7045267	0.49635794
2.37	6.55	2.76371308	1.95	0.81371308	0.66212898
2.31	6.55	2.83549784	5.12	-2.2845022	5.21895014
2.31	6.55	2.83549784	1.96	0.87549784	0.76649646
2.21	6.55	2.9638009	3.25	-0.2861991	0.08190992
2.45	6.55	2.67346939	1.85	0.82346939	0.67810183
4.92	6.55	1.33130081	1.35	-0.0186992	0.00034966
4.73	6.55	1.38477801	1.18	0.20477801	0.04193403
7.04	6.55	0.93039773	0.8	0.13039773	0.01700357
2.9	6.55	2.25862069	2.56	-0.3013793	0.09082949
2.6	6.55	2.51923077	2.01	0.50923077	0.25931598
6	6.55	1.09166667	1.01	0.08166667	0.00666944
Mean Squared Error					0.494097

3.9 Qualitative Experiments

We also carried out a qualitative experiment observing cars driving along Main Street (see Section 3.5 for example) and the average car moving through a green light was estimated by the application to have a velocity of 5-8 m/s, which is about 11.1847-17.9 mph, which seems reasonable for that street. For cars decelerating, the decrease in speed was detected by the application.

4 Discussion

In this implementation of optical flow, we show the utility of variations on the optical flow algorithm we worked on this semester. To reduce the noisy measurements, we tried implementing a simple center of gravity from the features detected by the Harris corner detector and then generated a tight, Gaussian point cloud around this center to estimate multiple vectors via optical flow. This, for some reason made measurements inaccurate.

References

[1] Horn, B.K.P. and B.G. Schunck "Determining Optical Flow — a Retrospective" Artificial Intelligence, Vol. 59, No. 1–2, February 1993, pp. 81–87.

[2] Horn, B.K.P. "Determining Constant Optical Flow" Stellar Paper.

[3] B.D. Lucas, T. Kanade, "An Image Registration Technique with an Application to Stereo Vision", in Proceedings of Image Understanding Workshop, 1981, pp. 121-130.

[4] C. Harris and M.J. Stephens. A combined corner and edge detector. In Alvey Vision Conference, pages 147–152, 1988.

[5] Harris Detector OpenCV documentation `https://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html?highlight=goodfeaturestotrack`

[6] Lucas-Kanade OpenCV Documentation `https://docs.opencv.org/3.4/d7/d8b/tutorial_py_lucas_kanade.html`