Using Optical Flow to Measure The Frequency of Vibrating Objects

Image Processing and Computer Vision - Mini-Project

Andreas Poulsen

April 2022

1 Introduction to Optical Flow

Optical Flow is a concept used when looking for motion in a video. Usually, the camera and background are static, and only the object is in motion. The motion between images (or frames) is described using flow vectors, which point in the direction of the movement. The magnitude of the vectors describe the speed of movement. [1]

Optical flow uses the brightness of pixel values to determine whether there is movement, which is why good features are necessary. Good features are typically corners of an object, where motion in any direction can be measured. For edges, only motion perpendicular to the line can be measured, and for patches of constant value, no change can be measured. [1]

There are different algorithms for implementing optical flow, but on of the most well known is the Lucas-Kanade Algorithm. [1]

1.1 Lucas-Kanade Algorithm

The Lucas-Kanade algorithm is an image registration algorithm which can take rotation and linear distortions into account. It assumes that the two images are in approximate registration. This means that there cannot be large motion between frames. [2]

Translational image registration can be characterized as described by Lucas and Kanade [2]: We are given functions $F(x)$ and $G(x)$ which give the respective pixel values at each location x in two images, where x is a vector. We wish to find the disparity vector h which minimizes some measure of the difference between $F(x+h)$ and $G(x)$, for x in some region of interest R. [2]

The Lucas-Kanade algorithm solves the image translation problem as follows: Assuming the object moves Δp (pixel position difference) over Δt (time difference) [1]:

$$f(p - \Delta p, t) = f(p, t + \Delta t) \tag{1}$$

Where $p = \begin{bmatrix} x \\ y \end{bmatrix}$ is the observation point, Δp is the movement at p, t is the time, Δt is the change, so $t + \Delta t$ is the second frame and f is the pixel brightness. [1]
Approximating equation 1 with a first-order Taylor expansion:

$$m(p - \Delta p) + b \approx f(p, t + \Delta t) \tag{2}$$

$$mp + m\Delta p + b \approx f(p, t + \Delta t) \tag{3}$$

Subtracting $f(p, t)$ from both sides:

$$mp + m\Delta p + b - f(p, t) \approx f(p, t + \Delta t) - f(p, t) \tag{4}$$

Since $f(p, t) = mp + b$, that yields:

$$-m\Delta p \approx f(p, t + \Delta t) - f(p, t) \tag{5}$$

$f(p, t)$ and $f(p, t + \Delta t)$ are known, $-m$ is the slope of the first-order Taylor expansion, and Δp is the unknown. [1]
$-m$ can be described as the derivative of the function $mp + b$, and thus:

$$m = dp = \begin{bmatrix} dx & 0 \\ 0 & dy \end{bmatrix}$$

m is then substituted by dp:

$$-dp\Delta p \approx f(p, t + \Delta t) - f(p, t) \tag{6}$$

Which, without vector notation can be written as:

$$dx * u + dy * v = I_t[x, y] - I_{t+\Delta t}[x, y] \tag{7}$$

Clearly, equation 7 is underspecified as there is a single equation with two unknowns, and therefore it is necessary to find another way to represent it.
By assuming that neighbors move together, they will also have the same u and v. Graphically, this can be described as looking in a small neighborhood (or window) around a pixel, for each position. This will give many more equations (9 in a 3x3 neighborhood), but with the same unknowns; u and v. The problem then becomes overspecified, and can be solved with least-square approximation. [1]
First, the notation is changed to a matrix representation:

$$S = \begin{bmatrix} dx_1 & dy_1 \\ dx_2 & dy_2 \\ \vdots & \vdots \\ dx_w & dy_w \end{bmatrix}$$

$$\Delta p = \begin{bmatrix} u \\ v \end{bmatrix}$$

$$T = \begin{bmatrix} I_t[x_1, y_1] - I_{t-\Delta t}[x_1, y_1] \\ I_t[x_2, y_2] - I_{t-\Delta t}[x_2, y_2] \\ \vdots \\ I_t[x_w, y_w] - I_{t-\Delta t}[x_w, y_w] \end{bmatrix}$$

Then

$$S\Delta p = T \tag{8}$$

Equation 8 can be solved with least-squares:

$$||S\Delta p - T||^2 = 0 \tag{9}$$

$$\Delta p = (S^T S)^{-1} S^T T \tag{10}$$

Where $(S^T S)$ must be invertible for a solution to be present. [1]

2 Lucas-Kanade Implementation

This version of the Lucas-Kanade algorithm takes four inputs: An image, the image after, some points to observe and a window size. When running the program, the image in figure 1 opens, and some points are chosen on the edge of the object - this is done by left-clicking on the image. After that, the user will press the escape button to close the image, and the algorithm runs.

Figure 1: Initial frame to pick points.

A new window opens which plots the points and their paths as they move during the video, as shown in figure 2. This is where the Optical Flow implementation comes in. For each image in the video, the algorithm runs for each point. This will be shown and explained with code snippets below. The algorithm is written in Python, and it uses the NumPy and OpenCV libraries. On lines 6-8 of listing 1, some kernels are initialized. These will be used to calculate the derivatives of the image.

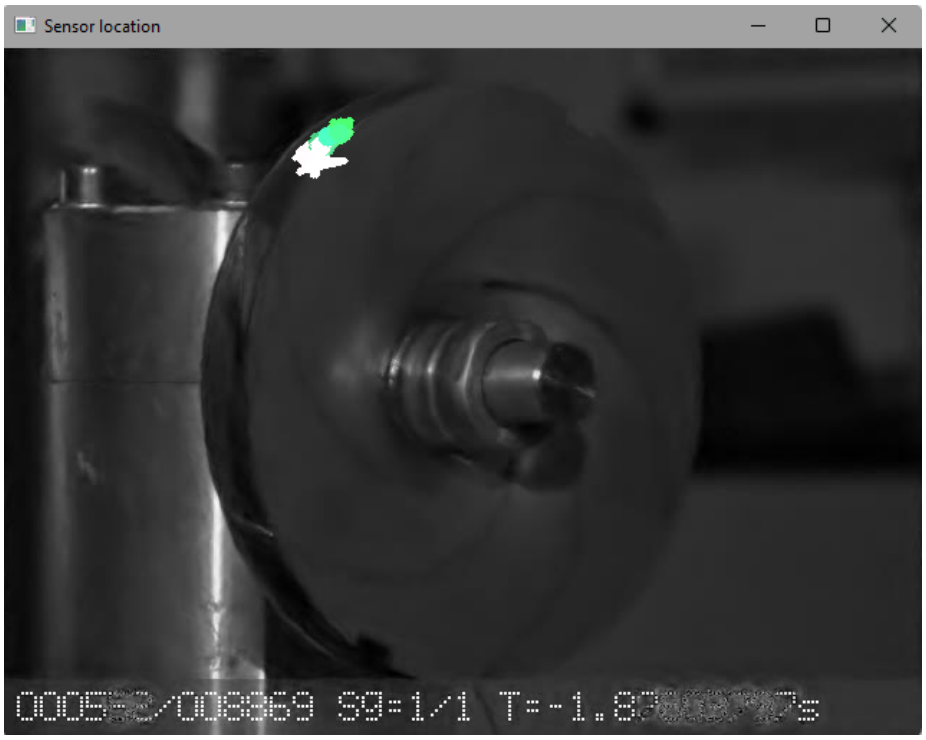

Figure 2: Points and their paths shown on top of the video.

```
6     kernel_x = np.array([[-1, 1], [-1, 1]])
7     kernel_y = np.array([[-1, -1], [1, 1]])
8     kernel_t = np.array([[1, 1], [1, 1]])
```

Listing 1: Kernel initialization

The image derivatives are calculated by convolution with the kernels that were just initialized. This can be seen in listing 2, where the OpenCV function **filter2D()** is used to convolve the image with the given kernels.

```
11    fx = cv2.filter2D(im1, -1, kernel_x)
12    fy = cv2.filter2D(im1, -1, kernel_y)
13    ft = cv2.filter2D(im2, -1, kernel_t) - cv2.
          filter2D(im1, -1, kernel_t)
```

Listing 2: Image convolution

In listing 3, lines 11 and 12 are simple variable declaration for later use. On line 15, a for-loop is initialized, which will go over each point from the input. These points are the ones set by the user on the initial image pop-up. As seen on line 17, the coordinates are retrieved using the **ravel()** function. Then, since

floating points cannot be used for array slicing, on line 18 they are converted to integers. On lines 21-23, the image derivatives for the window around the specific point are sliced.

```
15      op_flow = np.zeros(pts.shape)
16      count = 0
17
18      # For each feature point
19      for p in pts:
20          # Get coordinates
21          j, i = p.ravel()
22          j, i = int(j), int(i)
23
24          # Get derivatives for window
25          I_x = fx[i-win:i+win+1, j-win:j+win+1].flatten
                ()
26          I_y = fy[i-win:i+win+1, j-win:j+win+1].flatten
                ()
27          I_t = ft[i-win:i+win+1, j-win:j+win+1].flatten
                ()
```

Listing 3: Derivatives for given window

These smaller arrays of derivatives from listing 3, are then used to create the matrices A and b, as seen in 5 on lines 30 and 31. The image displacements are then calculated on line 34, where the pseudoinverse of A is multiplied by b as dictated in equation 10.

```
30          b = np.reshape(I_t, (I_t.shape[0],1))
31          A = np.vstack((I_x, I_y)).T
32
33          # Calculate u and v
34          U = np.matmul(np.linalg.pinv(A), b)
```

Listing 4: Making A and b and calulating U

Lastly, the displacements are stored in an array, as seen on lines 36 and 37. When the loop has gone through all points, the list of displacement is returned.

```
36          op_flow[count, 0, 0] = U[0, 0]
37          op_flow[count, 0, 1] = U[1, 0]
38          count += 1
39
40      return op_flow
```

Listing 5: Storing and returning

3 Results

Fast Fourier Transform is used to calculate the frequency of the vibrating object. In the test video, the object was vibrating at a frequency of 580Hz. The program keeps track of the frequencies of each feature point throughout the video and calculates an average in the end. These can be seen in the figures below. As is clear, all three feature points peak at $\approx$ 580Hz, with a recurring smaller peak at $\approx$ 100Hz. It is suspected that this 100Hz peak is caused by the lighting in the scene. As optical flow uses the brightness values of the pixels to calculate translation, it is obvious that a light flashing at 100Hz would be confused for movement. During testing, it also became clear that the positioning of the feature points plays a very big role in the result. For some points, even if placed on an edge, large displacements were happening, which could be because the lighting has a greater effect on those areas of the object. The OpenCV version of Lucas-Kanade was not as prone to these lighting changes, but this could be because they use pyramids in their version, so it is more robust to greater motion between frames, and perhaps also lighting changes in the scene.

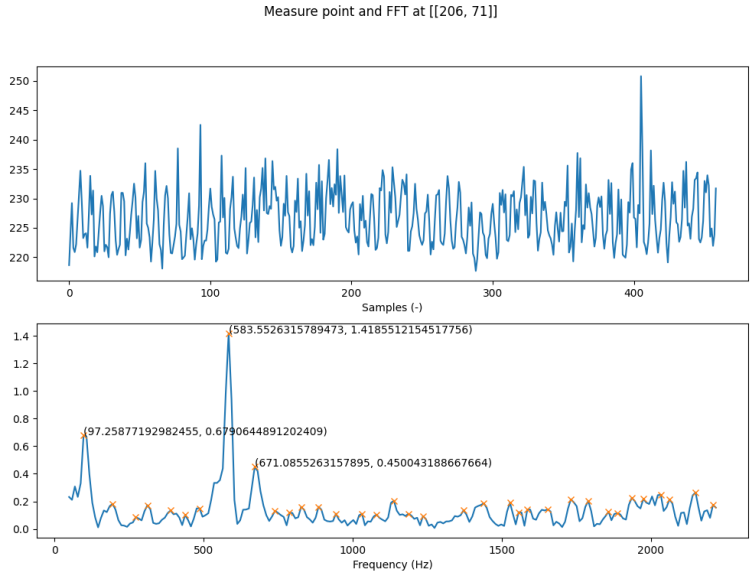

Figure 3: Measured frequencies for first feature point.

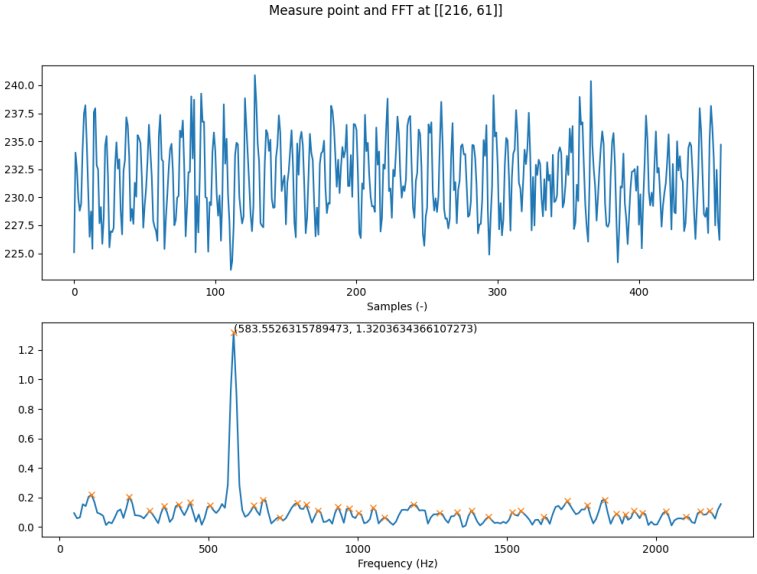

Figure 4: Measured frequencies for second feature point.

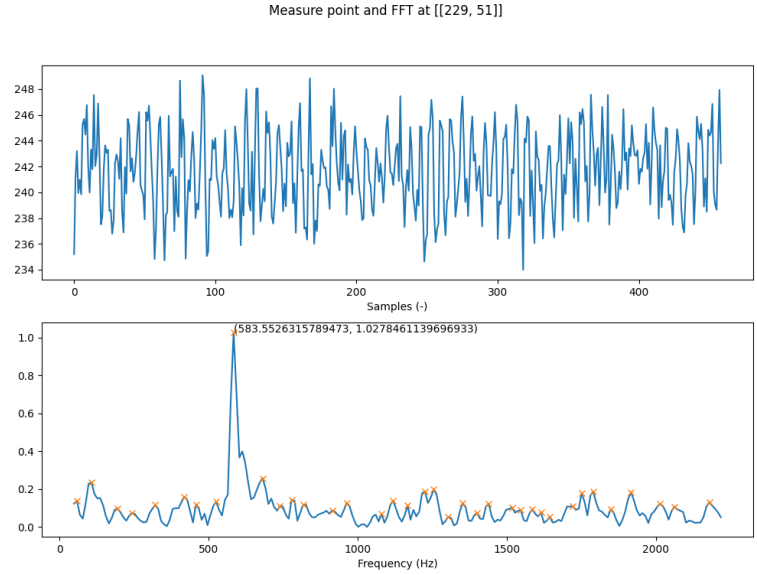

Figure 5: Measured frequencies for third feature point.

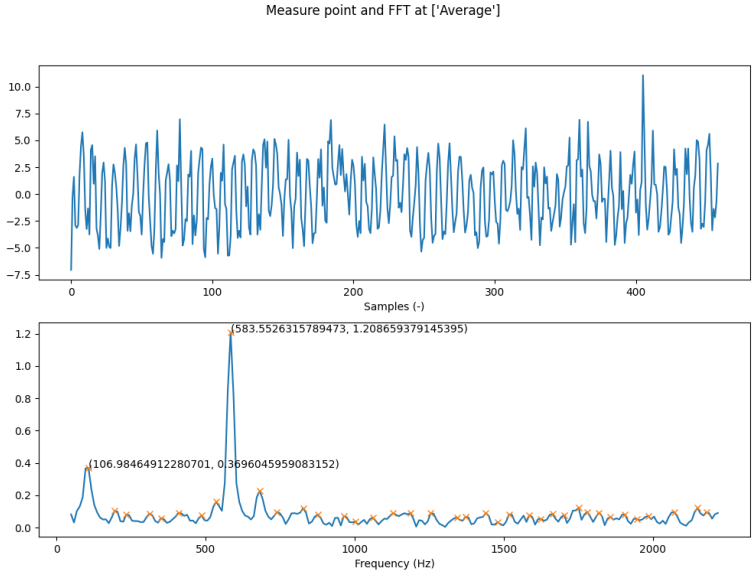

Figure 6: Average measured frequencies.

References

[1] Rikke Gade. *Lecture on Motion Analysis*. 2021.

[2] Bruce Lucas and Takeo Kanade. "An Iterative Image Registration Technique with an Application to Stereo Vision (IJCAI)". In: vol. 81. Apr. 1981.