Computer Vision

WPO (III) - Photometric Stereo

Professor: Prof. Hichem Sahli
TA: Dr. Abel Díaz Berenguer
Author: Hossein Dehghanipour (0582897)

January 2022

Contents

1 Introduction

In this project, we aimed on Optical Flow calculation and its representation using Quiver plot and Heat Map.

This report discusses some theoretical aspects of Lucas Kanade Method and describes each of the functions used in the code. In the last section, the results of the project is shown

2 Requirements and How To Run The Code

2.1 Requirements

Before running this code please make sure you have these tools:

- Anaconda 4.10.3 or higher.
- Python 3.7.9 or higher.
- OpenCV 4.0.1 or higher.
- Numpy 1.19.2 or higher.
- Scipy The newer version, the better.
- Matplotlib The newer version, the better.
- Spyder 4.1.5 or higher.

In order to do the installation easier and avoiding any conflicts with the packages you already have, it is suggested to use Anaconda as your package management.

2.2 Running The Code

In order to run the code, all you have to do is to open the **Main.py** in **Spyder** and then run the code by pressing F5 (or whatever your IDE settings is set to). If you want to see the source code of the functions, you can read **Main.py** file. You can take a look at the code to see how the mathematics is done. All parts of the code are well commented.

2.3 Directories

This project is consisted of Four directories names:

- **__pycache__** which is Python cache while we run the code.
- **Inputs** which contains the images we are using as input (8 frames).
- **Outputs** which contains the output images that are produced by our code.

3 Lucas and Kanade Method

In computer vision, the Lucas–Kanade method is a widely used differential method for optical flow estimation developed by Bruce D. Lucas and Takeo Kanade. It assumes that the flow is essentially constant in a local neighbourhood of the pixel under consideration, and solves the basic optical flow equations for all the pixels in that neighbourhood, by the least squares criterion.[1].
The Lucas–Kanade method assumes that the displacement of the image contents between two nearby instants (frames) is small and approximately constant within a neighborhood of the point p under consideration. Thus the optical flow equation can be assumed to hold for all pixels within a window centered at$p$. Namely, the local image flow (velocity) vector (V_x, V_y) must satisfy:

$$\begin{aligned} I_x(q_1)V_x + I_y(q_1)V_y &= -I_t(q_1) \\ I_x(q_2)V_x + I_y(q_2)V_y &= -I_t(q_2) \\ &\vdots \\ I_x(q_n)V_x + I_y(q_n)V_y &= -I_t(q_n) \end{aligned}$$

where $q_1, q_2, \ldots, q_n$ are the pixels inside the window, and$I_x(q_i), I_y(q_i), I_t(q_i)$ are the partial derivatives of the image I with respect to position x, y and time t, evaluated at the point $q_i$and at the current time. These equations can be written in matrix form $Av = b$, where

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

This system has more equations than unknowns and thus it is usually over-determined. The Lucas–Kanade method obtains a compromise solution by the least squares principle. Namely, it solves the 2^2 system

$$A^T A v = A^T b \text{ or}$$
$$\mathrm{v} = (A^T A)^{-1} A^T b$$

where A^T is the transpose of matrix A. That is, it computes

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

where the central matrix in the equation is an Inverse matrix. The sums are running from $i = 1$ to n. The matrix $A^T A$ is often called the structure tensor of the image at the point p.[1]

4 My Code

As you can see in the uploaded zip file, there is only a **main.py** file. In this file the following functions are implemented and we aim to go through them in this report.

- **main()**
- **plot_quiver(flow, imgGray, savePlotPathAndName, nvec)**
- **optical_flow(I1g, I2g, window_size, tau)**
- **readImage(imagePath)**
- **grayScaleImage(cvImage)**
- **gaussianFilter(image, kernel, border)**

4.1 Source Codes With Explanation

4.1.1 Main()

The main function is where it all happens. At first, we read two consecutive frames from the input folder and then we convert them to gray Scale images. In this example our images are already in gray scale but we still do that to make sure that any other given image would not interrupt this task. After converting the images to gray scale, we smoothen the sharp edges by applying a Gaussian filter. This is a very usual task that is almost done on every image in the image processing field. After smoothening the pictures, we calculate the optical flow between two consecutive frames.

```
def main():
    imgPath = "./Inputs/"
    outputPath = "./Outputs/"
    imagesNames = os.listdir(imgPath)
    print("Starting...")
    for i in range((len(imagesNames)- 1)):
        print("==========")

        # Obtain Images Paths
        imageFileNamepath_1 = imgPath + imagesNames[i]
        imageFileNamepath_2 = imgPath + imagesNames[i+1]
        print(f"Comparing Picture: ({imagesNames[i]}) and Picture: ({imagesNames[i+1]})")

        # Read The Images Into Vectors
        img1 = readImage(imageFileNamepath_1)
        img2 = readImage(imageFileNamepath_2)

        # Convert Images to Gray Scale
        img1_grey = grayScaleImage(img1)
        img2_grey = grayScaleImage(img2)

        # Smoothen The Image Using gaussian Blur
        img1_grey_blur = gaussianFilter(img1_grey)
        img2_grey_blur = gaussianFilter(img2_grey)


```

```
        # Calculate Optical Flow
        flow = optical_flow(img1_grey_blur, img2_grey_blur, 4)
        flow = np.array(flow)

        # Plot The Output and Save it
        pic1PureName = (imagesNames[i].split("."))[0]
        pic2PureName = (imagesNames[i+1].split("."))[0]

        savePlotPathAndName = outputPath + f"{pic1PureName}_{pic2PureName}.png"
        plot_quiver(flow, img1_grey_blur, savePlotPathAndName, nvec = 200 )
        print("Plot Saved")
        print("===============")
```

Listing 1: Main.py

4.1.2 plot_quiver(flow, imgGray, savePlotPathAndName, nvec)

The responsibility of this method is to get the optical flow and an image and draw a quiver plot along with a heat map on that image. The code is explained in detail in the below section.

```
def plot_quiver(flow, imgGray, savePlotPathAndName, nvec = 200 ):
    '''
    Parameters
    ----------
    flow :
        The optical Flow we have calculated.
    imgGray :
        The first frame of each calculation we use to show our quivers on.
    savePlotPathAndName :
        The name of the figure we are going to save as our final output.
    nvec : Integer, optional
        The number of vectors we wish to see on our image as the quivers. The default is 200.

    Returns
    -------
    None.

    '''
    u,v = flow
    fig, (ax1,ax2) = plt.subplots(1,2, figsize=(10, 5))
    nl, nc = np.array(imgGray).shape

    # Calculate number of steps for drawing quivers
    step = max(nl//nvec, nc//nvec)
    norm = np.sqrt(u ** 2 + v ** 2)
    y, x = np.mgrid[:nl:step, :nc:step]

    # Extract U and V vectors
    u_ = u[::step, ::step]
    v_ = v[::step, ::step]

    # Show Subplot 2
    ax2.imshow(norm)
    ax2.set_title("HeatMap")
    ax2.set_axis_off()

    # Show Subplot 1 along with the Quivers
```

```
    ax1.imshow(norm)
    ax1.quiver(x, y, u_, v_, color='r', units='dots',angles='xy', scale_units='xy', lw=3)
    ax1.set_title("Optical flow magnitude and vector field")
    ax1.set_axis_off()

    # Save the plots in the given destination
    fig.tight_layout()
    #plt.show()
    plt.savefig(savePlotPathAndName)
    plt.close(fig)
```

Listing 2: plot_quiver(...)

4.1.3 optical_flow(image1, image2, window_size, tershhold)

This method is the heart of our project. This function calculates the optical flow of two consecutive frames and outputs the u and v vectors at the end. The frames should have some aspects like having a little change in the motion and also the lighting should not differ.

```
def optical_flow(image1, image2, window_size, tershhold=1e-2):
    '''
    Parameters
    ----------
    I1g : Image
        The first Frame we want to calculate the optical flow with.
    I2g : Image
        The Second Frame we want to calculate the optical flow with.
    window_size : Tuple of integer
        The window size that we set our kernel to take the number of pixels as a window. window_size should be odd. all the pixels with offset in between [-w, w] are inside the window
    tershhold : TYPE, optional
        The treshhold we set for our gradient descent to avoid exceeding this amount. The default is 1e-2.

    Returns
    -------
    u : TYPE
        The u vector in Optical Flow
    v : TYPE
        The V vector in Optical Flow.

    '''

    # Intialize the Kernel
    kernel_x = np.array([[-1., 1.], [-1., 1.]])
    kernel_y = np.array([[-1., -1.], [1., 1.]])
    kernel_t = np.array([[1., 1.], [1., 1.]])

    # Set Width [-w, +w]
    w = int(window_size/2) #

    # normalize the pixels by dividing them by 255 as RGB max value
    I1g = image1 / 255. # normalize pixels
    I2g = image2 / 255. # normalize pixels

    # for each point, calculate I_x, I_y, I_t
```

```

    # Caluclate Convolution with Symmetric Boundy for Ex, Ey, Et
    Ex = signal.convolve2d(I1g, kernel_x, boundary='symm', mode='same')
    Ey = signal.convolve2d(I1g, kernel_y, boundary='symm', mode='same')
    minUsKernelConvoltion = signal.convolve2d(I1g, -kernel_t, boundary='symm', mode='same')
    Et = signal.convolve2d(I2g, kernel_t, boundary='symm', mode='same') + minUsKernelConvoltion

    # Initialize the U and V vectors
    u = np.zeros(I1g.shape)
    v = np.zeros(I1g.shape)


    # Calculate Ix, Iy, It
    # within window window_size * window_size
    for i in range(int(w), int(I1g.shape[0]-w)):
        for j in range(int(w), int(I1g.shape[1]-w)):
            begin1 = i - w
            end1 = i + w + 1

            begin2 = j - w
            end2 = j + w + 1

            Ix = Ex[begin1:end1, begin2:end2].flatten()
            Iy = Ey[begin1:end1, begin2:end2].flatten()
            It = Et[begin1:end1, begin2:end2].flatten()
            b = np.reshape(It, (It.shape[0],1)) # get b here
            A = np.vstack((Ix, Iy)).T # get A here

            # Calculate the Eigenvalues and compare to the Treshhold
            if np.min(abs(np.linalg.eigvals(np.matmul(A.T, A)))) >= tershhold:
                nu = np.matmul(np.linalg.pinv(A), b) # get velocity here
                u[i,j]=nu[0]
                v[i,j]=nu[1]
    return (u,v)
```

Listing 3: optical_flow(...)

4.1.4 readImage(imagePath)

This method reads the image by using OpenCV library and returns the image in a vector form.

```
def readImage(imagePath):
    '''
    Reads an Image using the path given by the user

    Parameters
    ----------
    imagePath : String
        The Path that the image is saved.

    Returns
    -------
    Image Vector
        Returns the read image.

    '''
```

```
    return cv.imread(imagePath).astype(dtype = np.float32)
```

Listing 4: readImage(...)

4.1.5 grayScaleImage(cvImage)

This method Converts the image into the gray scale format

```
def grayScaleImage(cvImage):
    '''
    Converts the image into the gray scale format

    Parameters
    ----------
    cvImage : TYPE
        DESCRIPTION.

    Returns
    -------
    TYPE
        DESCRIPTION.

    '''
```

Listing 5: grayScaleImage(...)

4.1.6 gaussianFilter(image, kernel = (1,1), border = 0)

This function applies a Gaussian filter on our picture in order to remove the sharp edges that might cause problems while calculating gradient. Removing the sharpness of our image by blurring it is something usual in image processing.

```
def gaussianFilter(image, kernel = (1,1), border = 0):
    '''
    Applies A Gaussian Filter on out image to remove the sharp edges that might
   cause problems while calculating gradient. Removing the sharpness of our
   image by blurring it is something usual in image processing.

    Parameters
    ----------
    image : TYPE
        The passed vector of the image that we want to apply the gaussian filter
     on.
    kernel : (int,int)
        Filter's Kernel Size. The default is (1,1).
    border : int
        Added border by the filter . The default is 0.

    Returns
    -------
    Image
        blured image.

    '''
    return cv.GaussianBlur(image, kernel, border)
```

Listing 6: gaussianFilter(...)

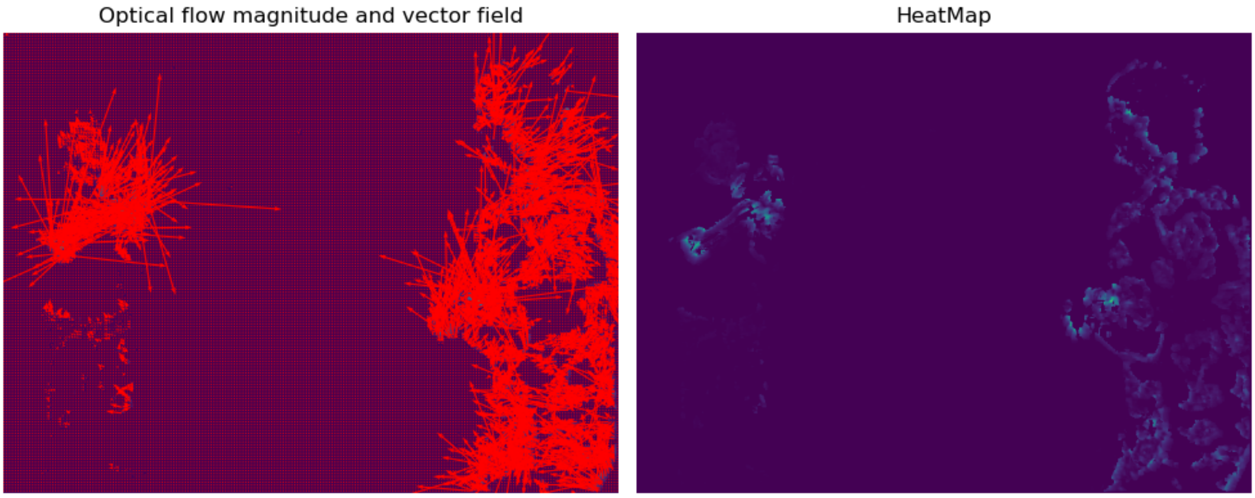
5 Experimental Results

You can see all of the outputs in the **Outputs** folder.

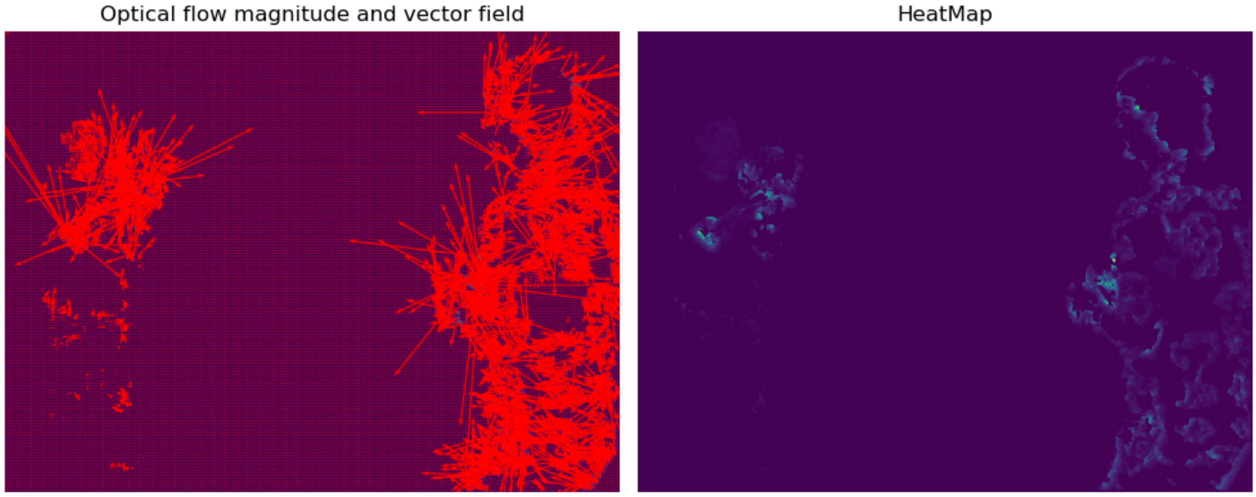
Figure 1: frame1.png

Figure 2: frame2.png

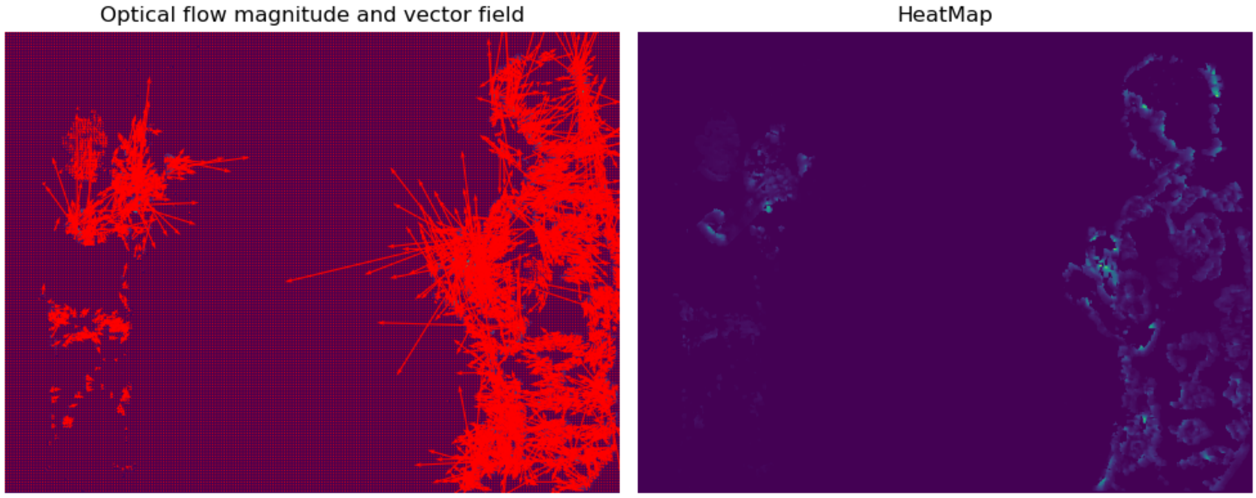
Figure 3: frame3.png



frame1_frame2.png



frame2_frame3.png



frame3_frame4.png

6 References

[1] https://en.wikipedia.org/wiki/Lucas%E2%80%93Kanade_method

Listing 7: Main.py