**Date:** 11/05/2021

**CS 6643 -** Computer Vision, Spring 2021 **Prof.James Fishbaugh**

# PROJECT III: OPTICAL FLOW: LUCAS-KANADE, HORN-SCHUNCK

**Pruthviraj R Patil, Computer Science, MS.**
**New York University, Tandon School of Engineering**
**Email:** prp7650@nyu.edu

## A) Optical Flow

In this project, you will implement the Lucas-Kanade and Horn-Schunck optical flow methods. Both methods are based on spatial and temporal derivatives, differing mainly in the optimization approach.

### 1.1. Implementation:

1. Assume two images with the same dimensions, e.g. im1 = sphere0.png and im2 = sphere1.png.

2. Compute spatial partial derivative images Ix and Iy for im1 using any method of your choice
3. Compute temporal partial derivative It. This can be approximated by image difference im2 - im1.

**SOLUTIONS:**

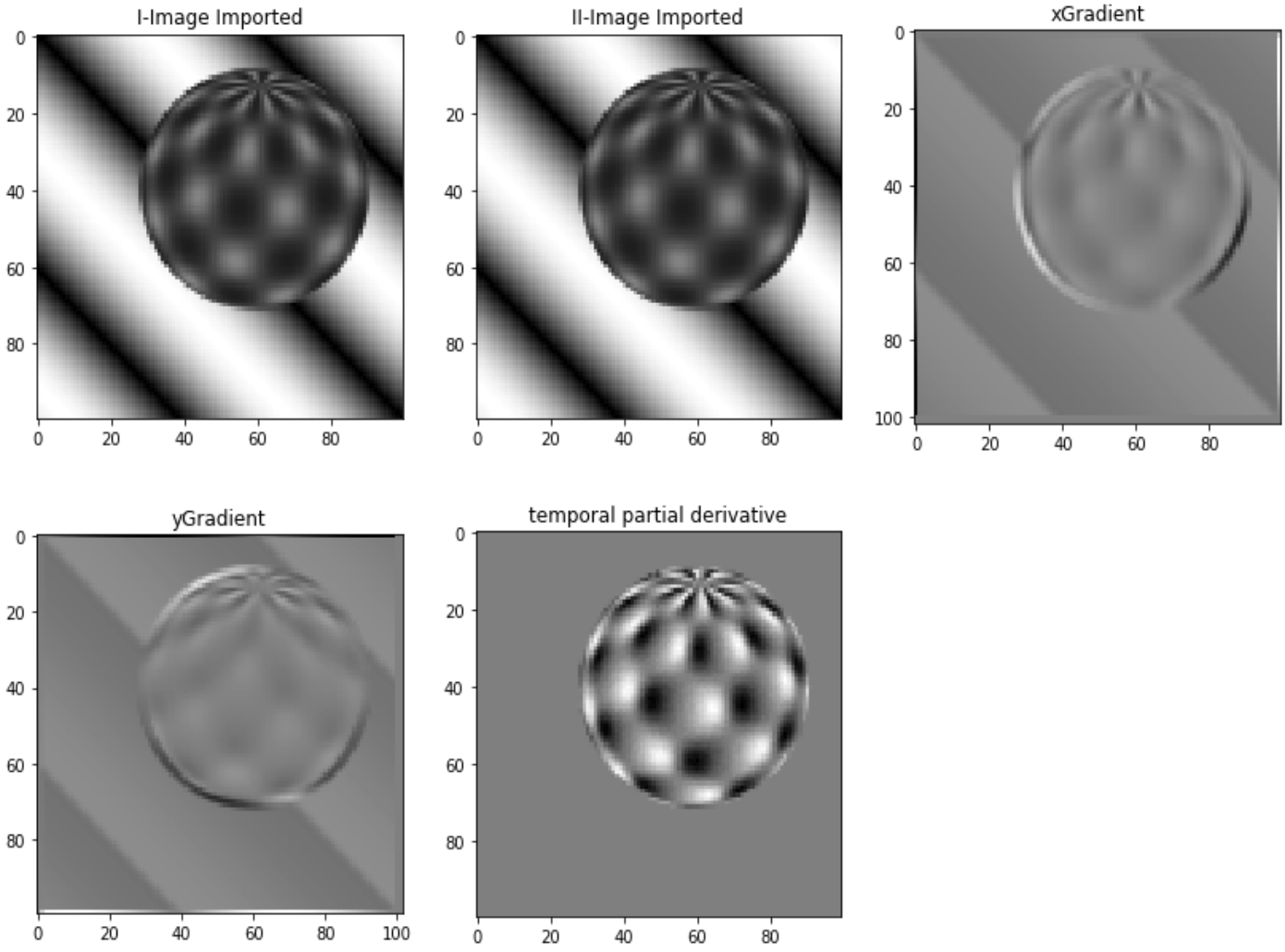


**Figure 1: Solutions to 2 and 3**

### 1.2. Implement Lucas-Kanade and Horn-Schunck

#### 1.21. Synthetic Sphere Images

I. Using sphere0.png and sphere1.png, explore the Lucas-Kanade method using a neighborhood size of 3, 5, 11, and 21.

- For each neighborhood size, plot optical flow as a vector field and the magnitude of the optical flow field at every pixel.
- Describe the impact of the neighborhood size. Do you conclude that it plays a large role in the estimation of the optical flow field?

**SOLUTION:**

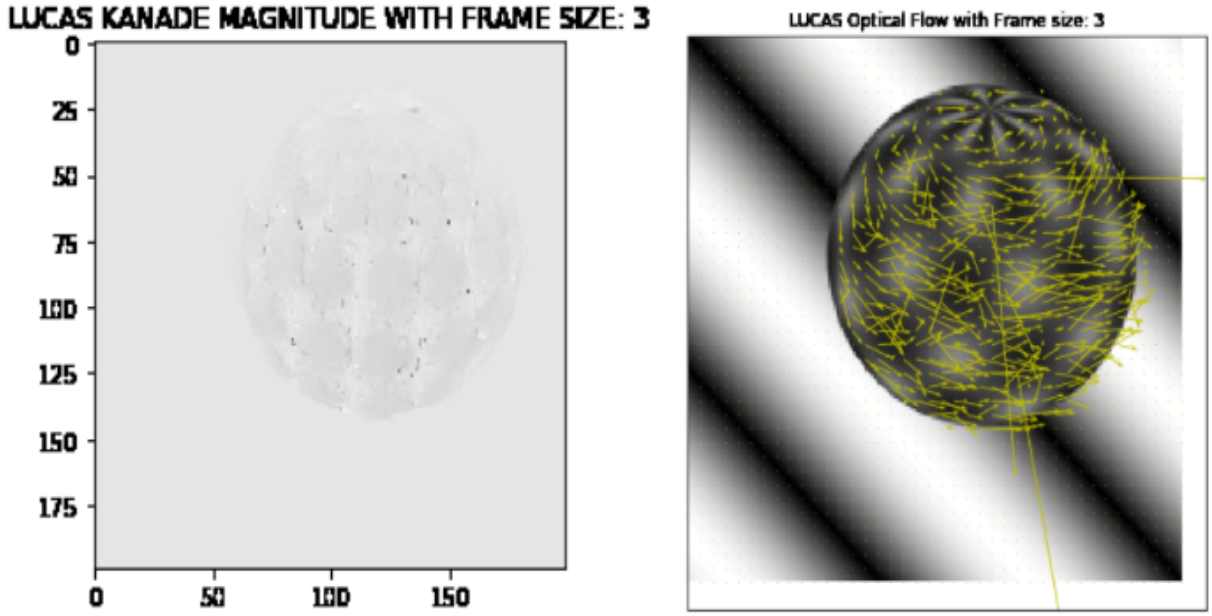


**Figure: 1.21: optical flow as a vector field and the magnitude at every pixel for neighborhood size: 3**

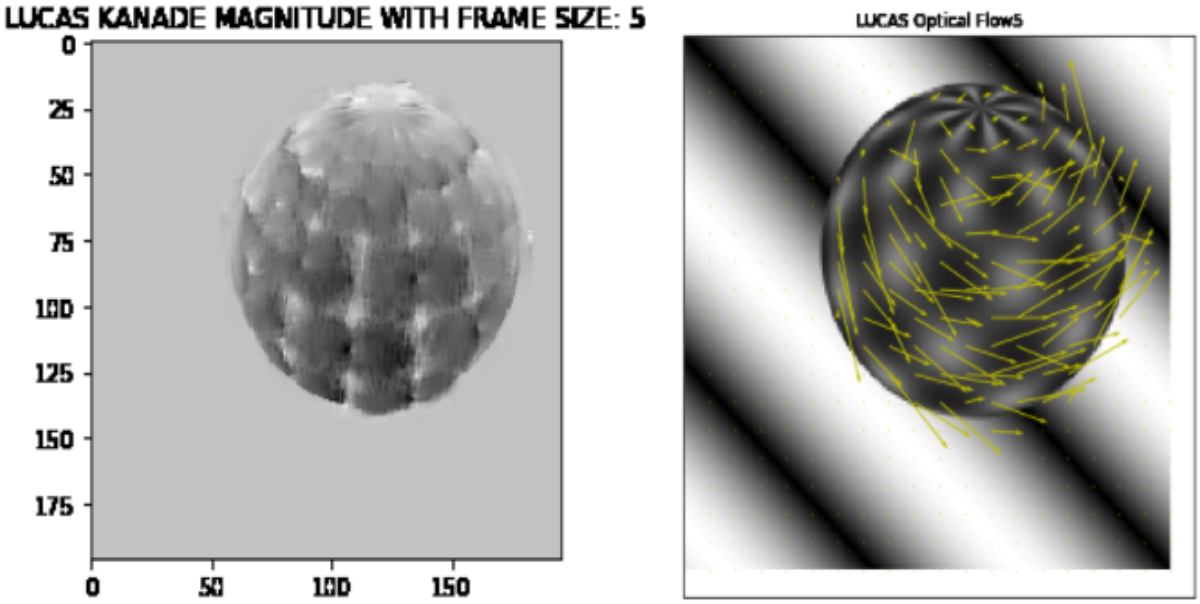


**Figure: 1.22: optical flow as a vector field and the magnitude at every pixel for neighborhood size: 5**

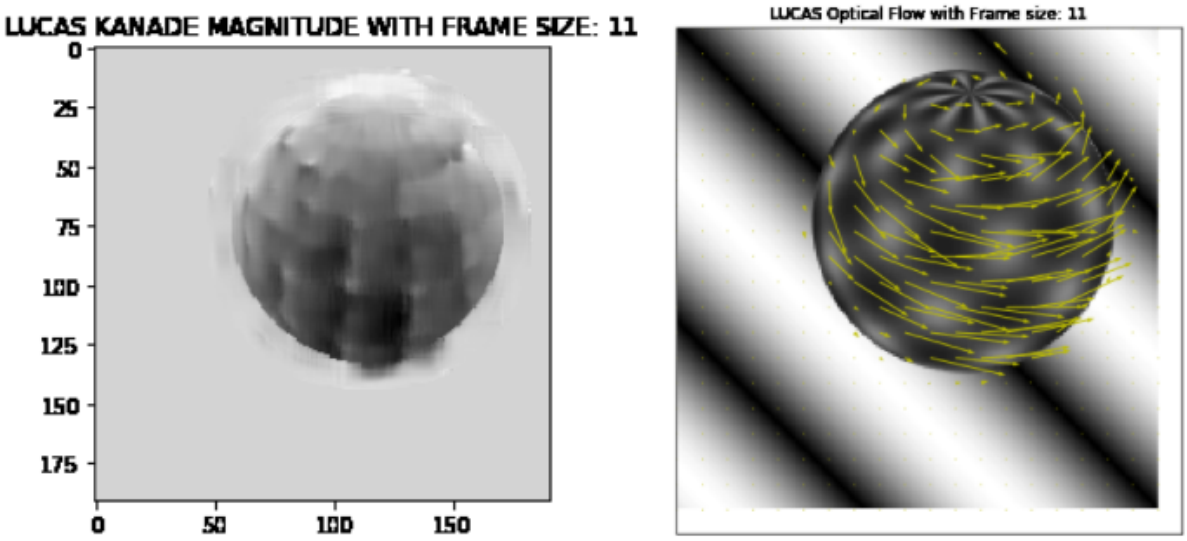


**Figure: 1.23: optical flow as a vector field and the magnitude at every pixel for neighborhood size: 5**

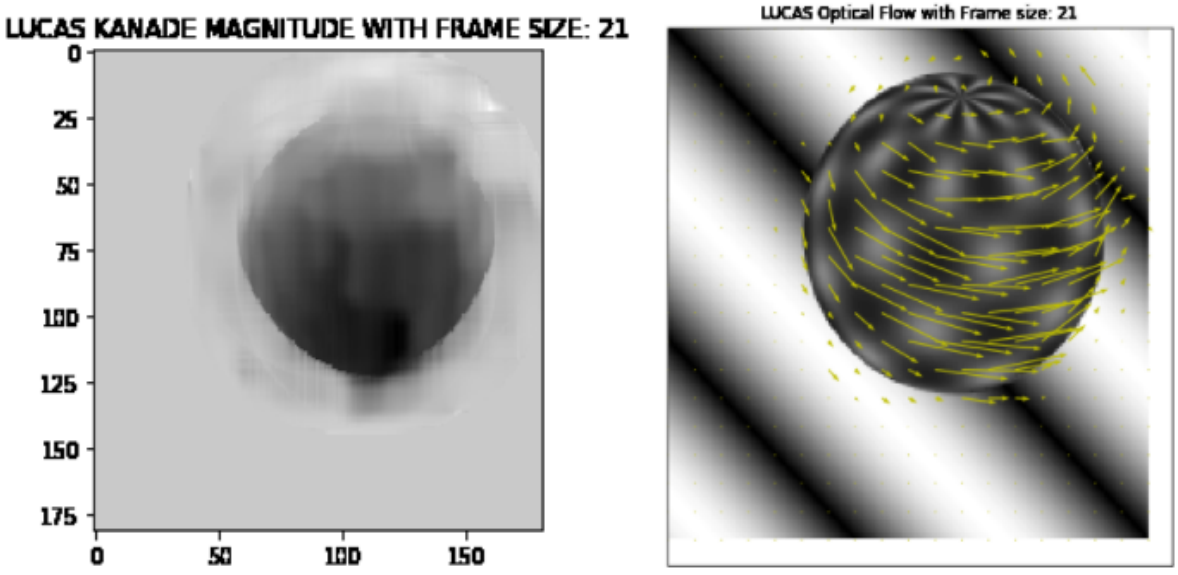


**Figure: 1.24: optical flow as a vector field and the magnitude at every pixel for neighborhood size: 5**

The frame (neighborhood) size of the do impact in a very decisive manner because of many factors. If we use 5*5 neighborhood window, we get total of 25*3 equations to calculate u, v. Hence, more the frame size (say size = 11), the magnitude of the optical flow between the two consecutive temporal images is seen more. This results in the vectors to be seen in a proper direction (referring to figure 1.23, 1.24). But, if in the case of the lower sized window, the magnitude of the optical flow is not properly visible (figure 1.21, 1.22). Hence, some of the vectors go in the random direction. However, if the frame size increases more, then there can be too much of the generalization given to the direction of the movement hence, we can miss the minute details of temporal change but only get the bigger picture of the flow.

II. Using sphere0.png and sphere1.png, explore the Horn-Schunck method using values of λ = 100, 10, 1, and 0.1.

- For each value of λ, plot optical flow as a vector field and the magnitude of the optical flow field at every pixel.
- Describe the impact of λ. Do you conclude that it plays a large role in the estimation of the optical flow field?

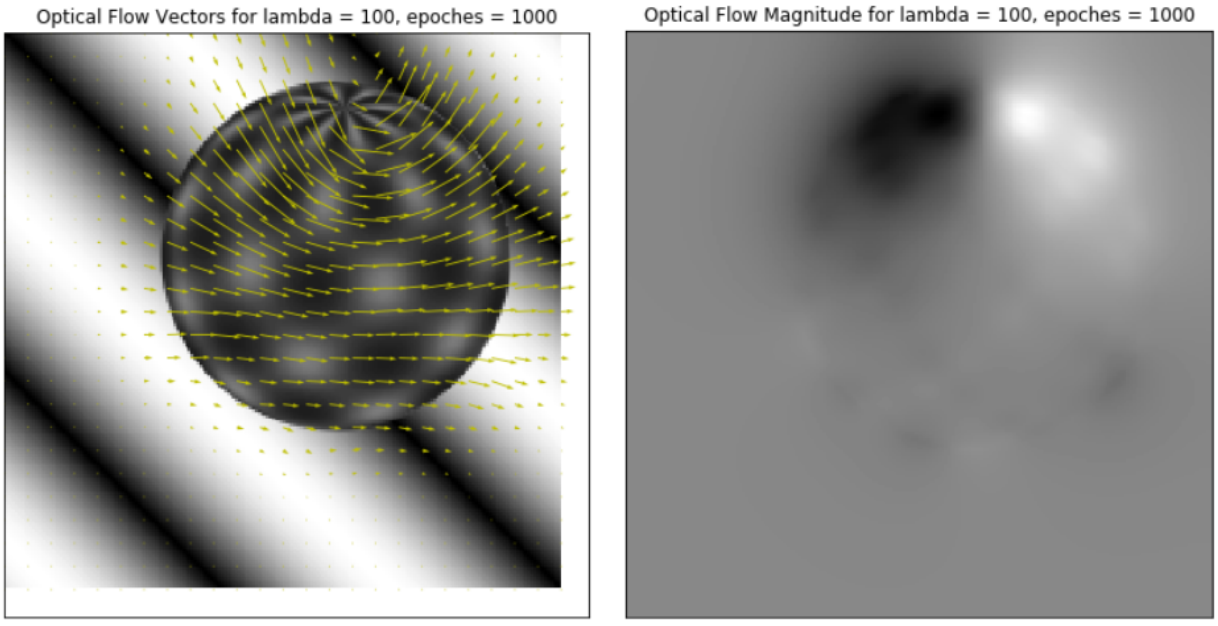


**Figure 1.25: Optical flow as vector and magnitude with lambda=100 , epochs=1000**

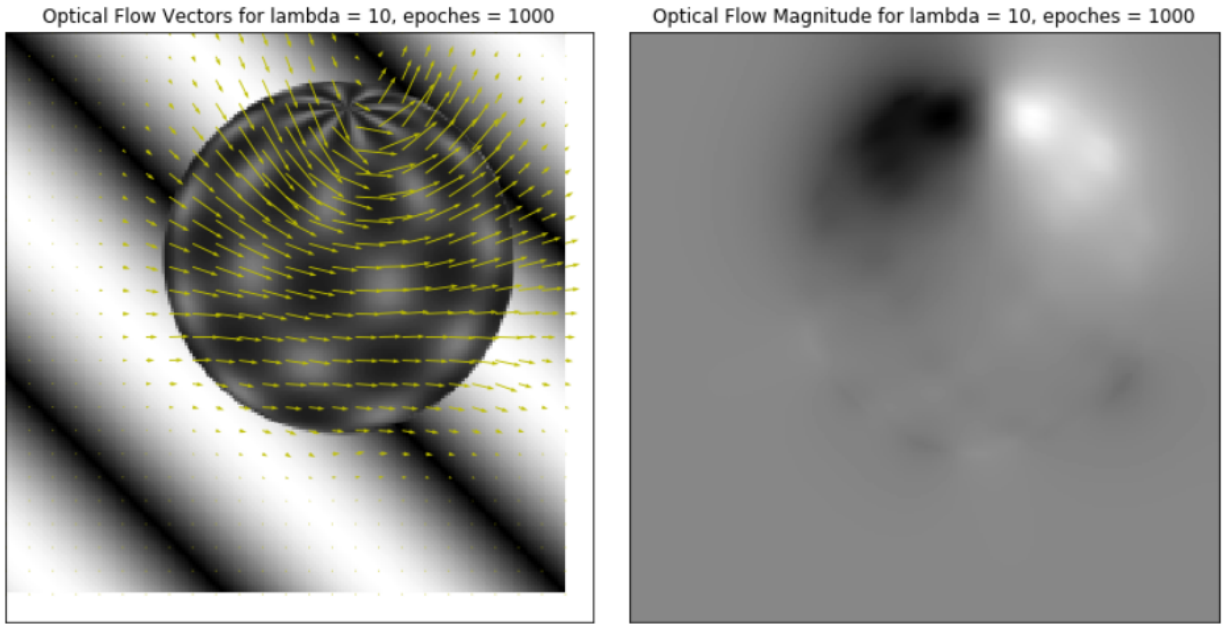


**Figure 1.26: Optical flow as vector and magnitude with lambda=10 , epochs=1000**

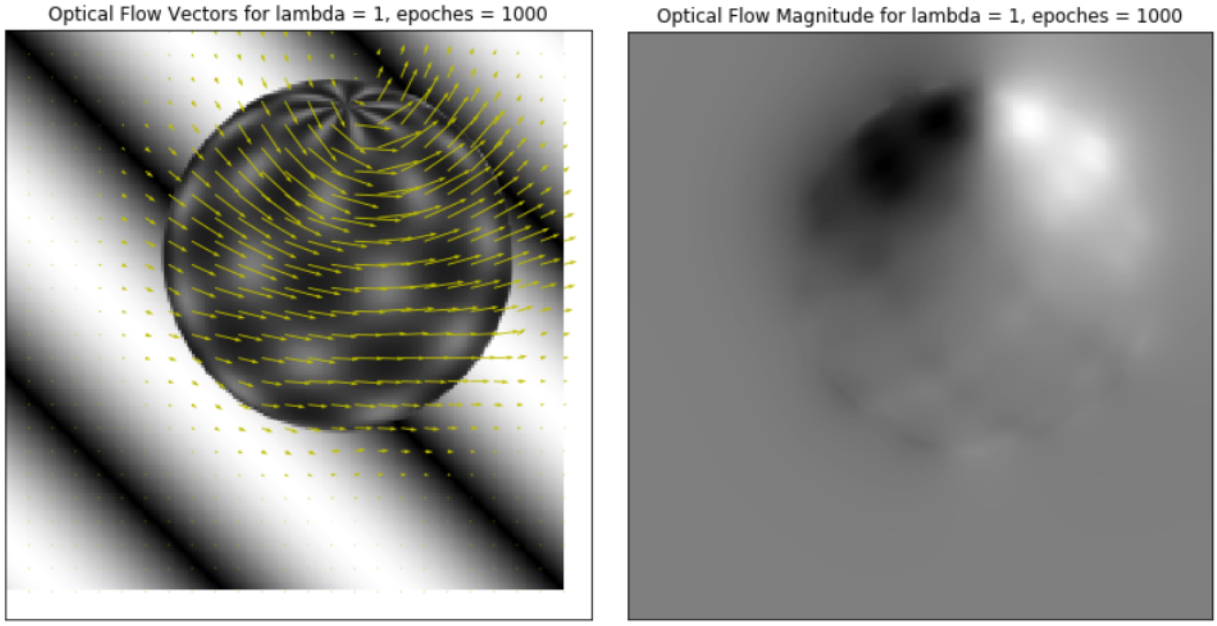


**Figure 1.27: Optical flow as vector and magnitude with lambda=1 , epochs=1000**

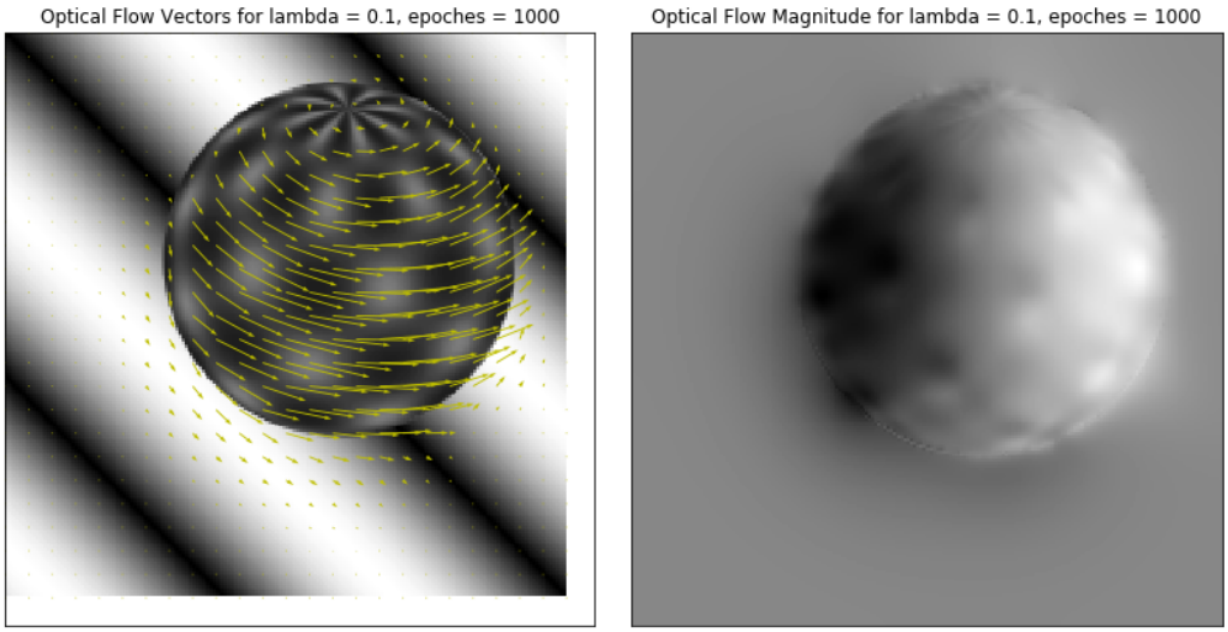


**Figure 1.28: Optical flow as vector and magnitude with lambda=0.1 , epochs=1000**

The value of λ does play the major role in the calculation of optical flow vectors. The values of u and v in the process are calculated by subtracting their values in previous epochs by the factor which is inversely proportional to λ. This makes the process converge more early if the value of lambda is decreased. Because the new values of U and V are used to calculate the Energy values as shown below. However, if the lambda value is decreased too much, then too the energy value can shift to the other side, becoming more negative hence yielding the incorrect results. However, if the lambda value increases a lot, the number of epochs needed might be even more, this might also cause problems of increase in time complexity. This process would be similar to Gradient descent. This can be seen in the above sphere vector and magnitude of optical flow outputs as well.

$$u_{i,j}^{k+1} = \bar{u}_{i,j}^{k} - \frac{I_x(i,j)(I_x(i,j)\bar{u}_{i,j}^{k} + I_y(i,j)\bar{v}_{i,j}^{k} + I_t(i,j))}{\gamma^2 + I_x(i,j)^2 + I_y(i,j)^2}$$

$$v_{i,j}^{k+1} = \bar{v}_{i,j}^{k} - \frac{I_y(i,j)(I_x(i,j)\bar{u}_{i,j}^{k} + I_y(i,j)\bar{v}_{i,j}^{k} + I_t(i,j))}{\gamma^2 + I_x(i,j)^2 + I_y(i,j)^2}$$

Smoothness on optical flow:

$$E_s(i,j) = \frac{1}{4}\Big[(u_{i,j} - u_{i+1,j})^2 + (u_{i,j} - u_{i,j+1})^2 + (v_{i,j} - v_{i+1,j})^2 + (v_{i,j} - v_{i,j+1})^2\Big]$$

Brightness constancy:

$$E_d(i,j) = [I_x(i,j)u_{i,j} + I_y(i,j)v_{i,j} + I_t(i,j)]^2$$

The overall cost function:

$$E(\mathbf{u},\mathbf{v}) = \sum_{i,j}\Big[E_d(i,j) + \gamma E_s(i,j)\Big]$$

### 1.3. Real Traffic Images

I. Using traffic0.png and traffic1.png, get the best results you can using the Lucas-Kanade method.

- Show the best result you were able to obtain along with any parameter values used. Plot optical flow as a vector field and the magnitude of the optical flow field at every pixel.
- Discuss the process of obtaining good results, i.e. was it easy/difficult?

**SOLUTION:**

The best results that I was able to obtain along with a parameter value of 15 neighborhood size is as follows:

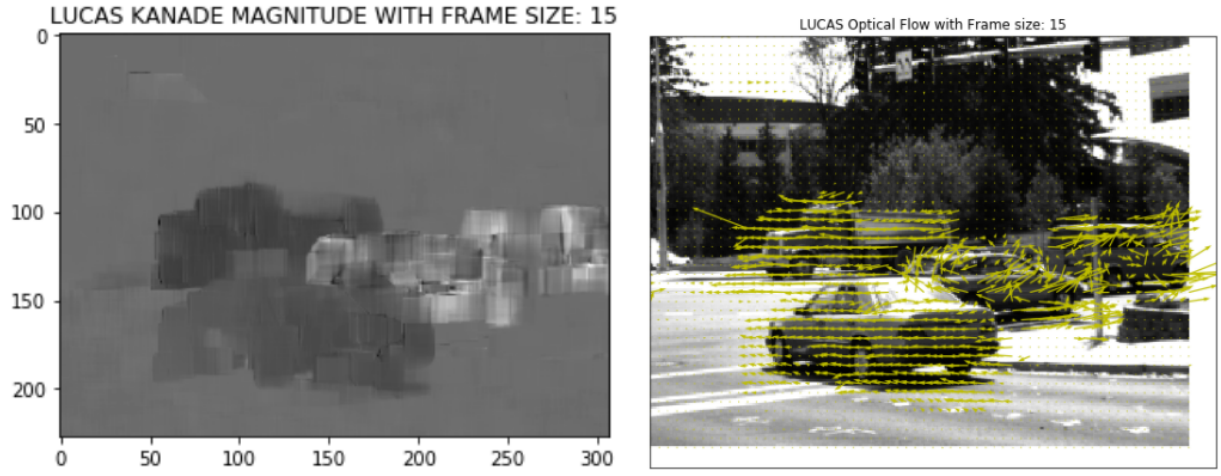


**Figure: 1.31, 32 : magnitude of the optical flow field at every pixel for neighborhood size: 15**

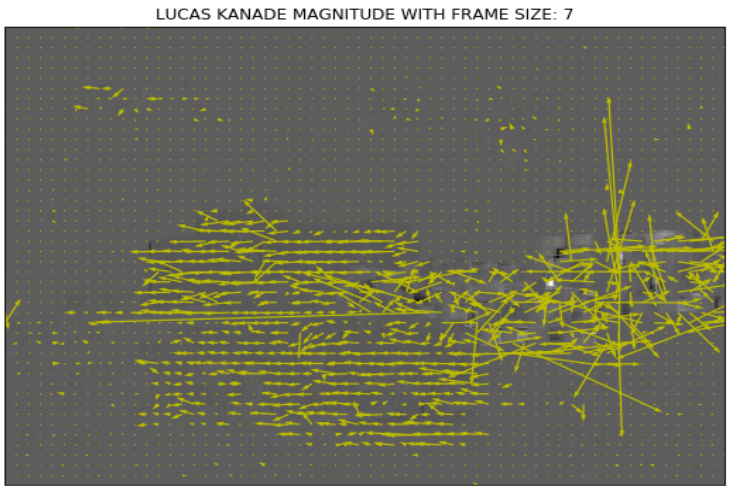


**Figure: 1.33: optical flow as a vector field over magnitude at every pixel for neighborhood size: 7**

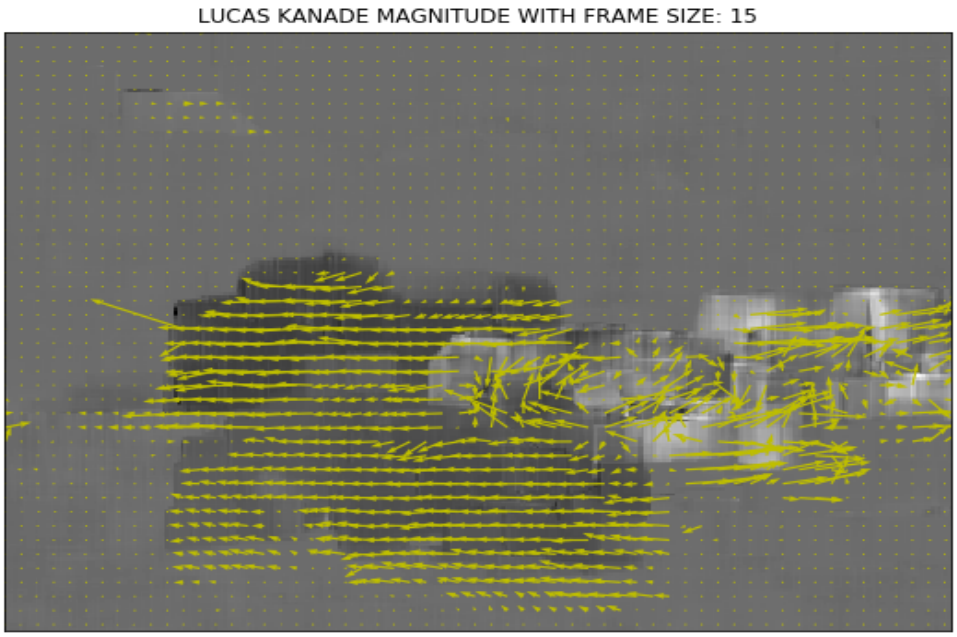


**Figure: 1.34: optical flow as a vector field over magnitude at every pixel for neighborhood size: 15**

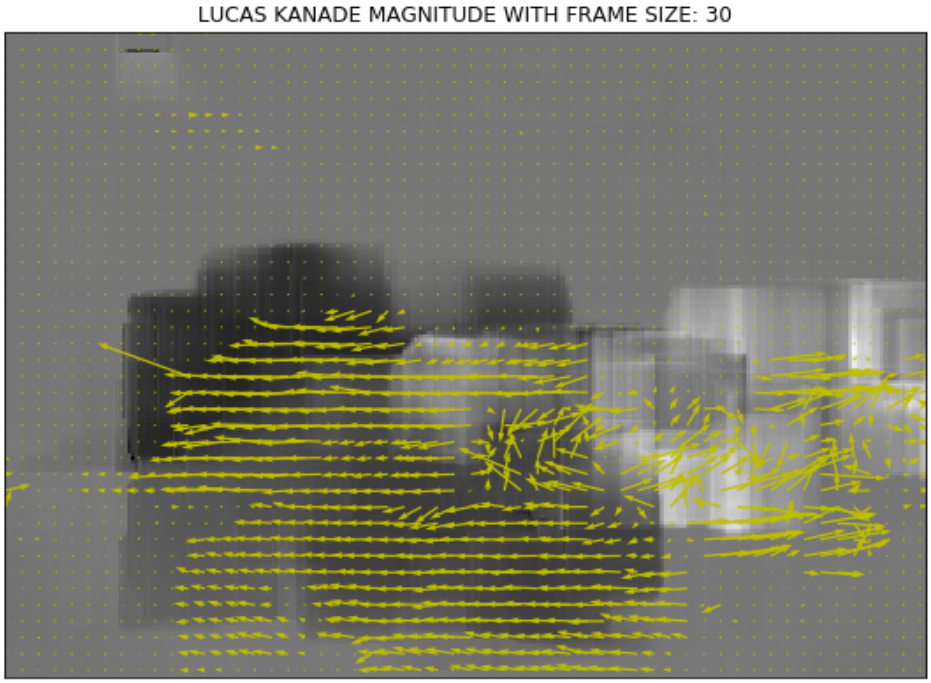


**Figure: 1.35: optical flow as a vector field over magnitude at every pixel for neighborhood size: 30**

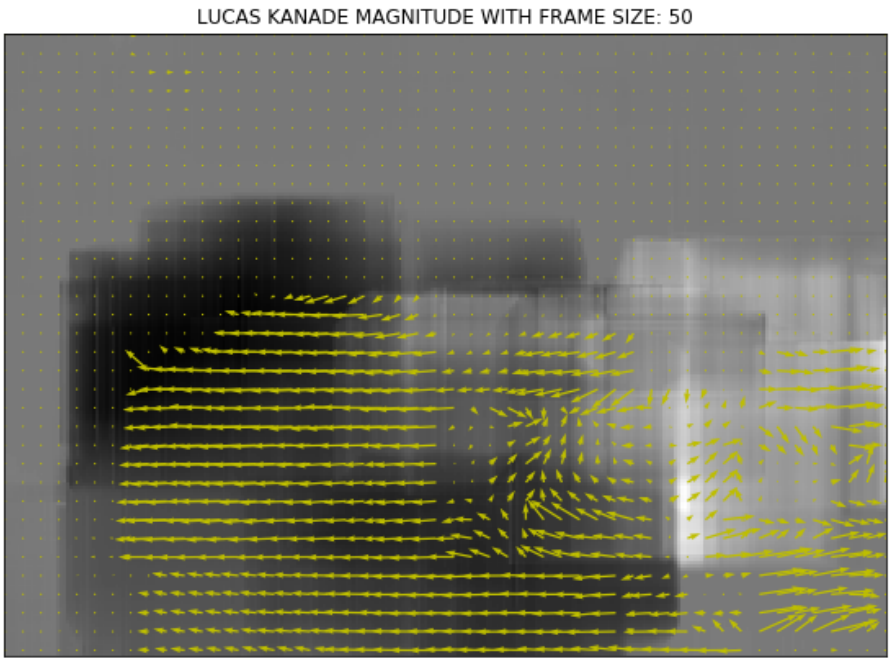


**Figure: 1.36: optical flow as a vector field over magnitude at every pixel for neighborhood size: 50**

Yes, it was difficult inorder to speculate the proper number of neighborhoods to be considered in order to gain viable results. This is because the objects in the traffic temporal images moved in the three different directions. The two cars moved towards the left whereas the truck towards the right and the other car gliding towards the center. Hence, the optical flow in the lower order frames showed less accurate changes in terms of the optical flow (Can be seen in the figure 1.33). But, even if the neighborhood size is increased a lot, (figure 1.36), even though the directions can be in the proper directions, they are not accurate. That is why, the neighborhood sizes have to be in between those sizes. Thus, the array of [7, 15, 30, 50] neighborhood sizes are considered in order to try and the best amongst us is considered to be of the frame size 15 (output of magnitude and direction of the flow are shown in the figure 1.31, 1.32).

II. Using traffic0.png and traffic1.png, get the best results you can using the Horn-Schunck method.

- Show the best result you were able to obtain along with any parameter values used. Plot optical flow as a vector field and the magnitude of the optical flow field at every pixel.
- Discuss the process of obtaining good results, i.e. was it easy/difficult?

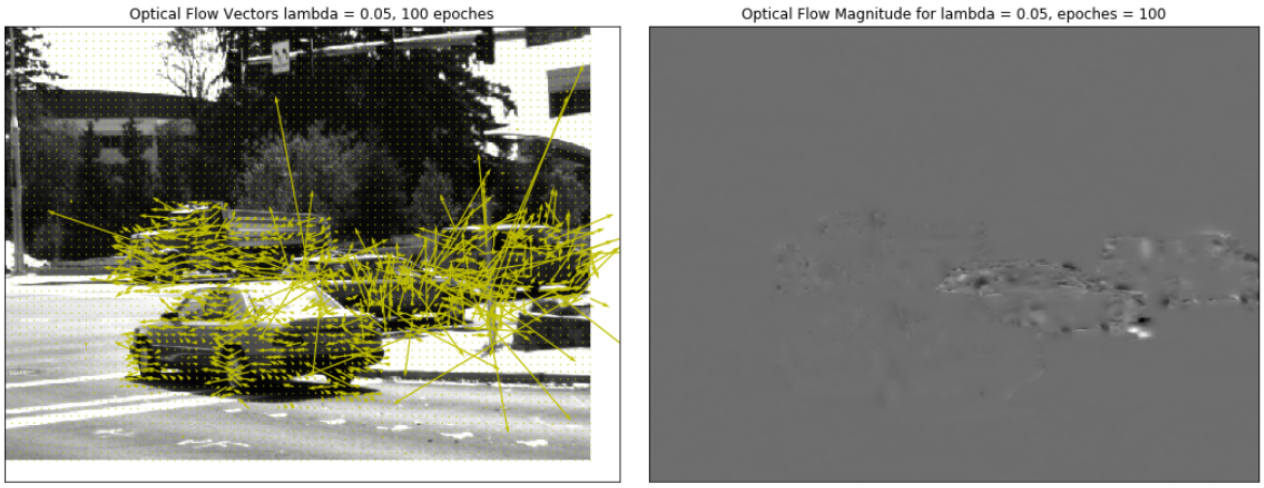


**Figure 1.37: Optical flow as vector and magnitude with lambda=0.05 , epochs= 100**

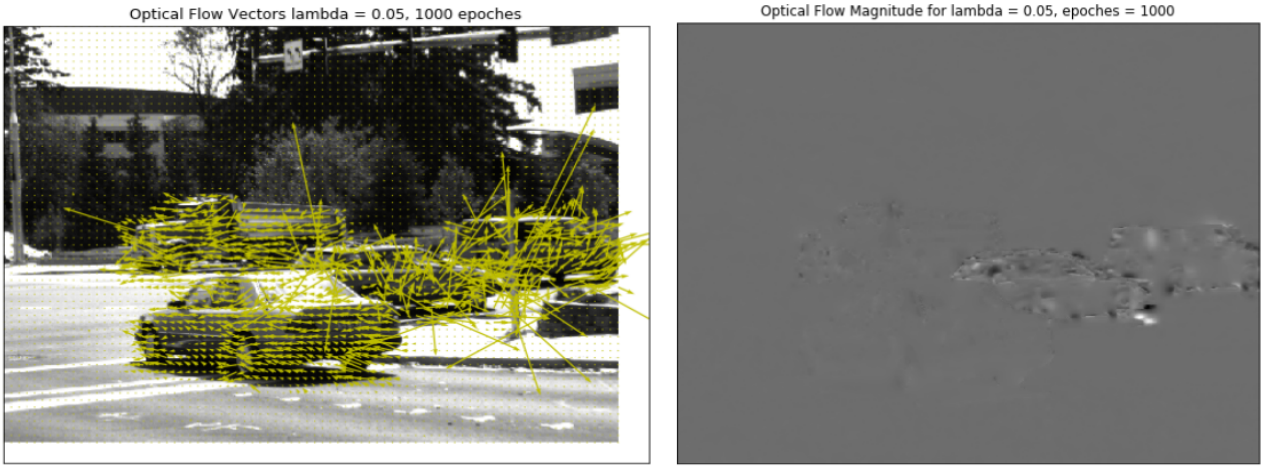


**Figure 1.38: Optical flow as vector and magnitude with lambda=0.05 , epochs= 1000**

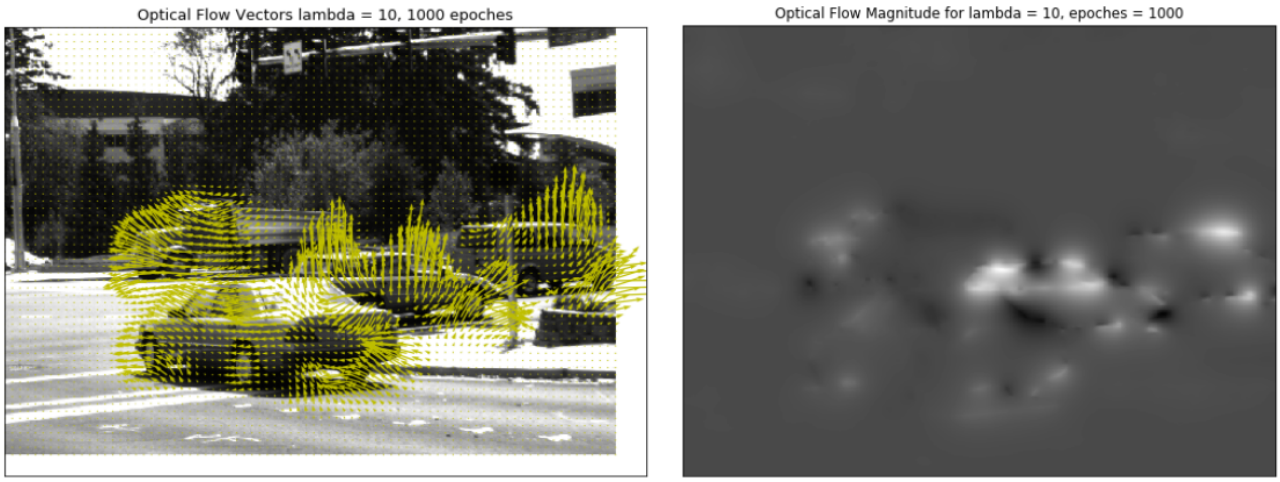

**Figure 1.39: Optical flow as vector and magnitude with lambda=10 , epochs=1000**

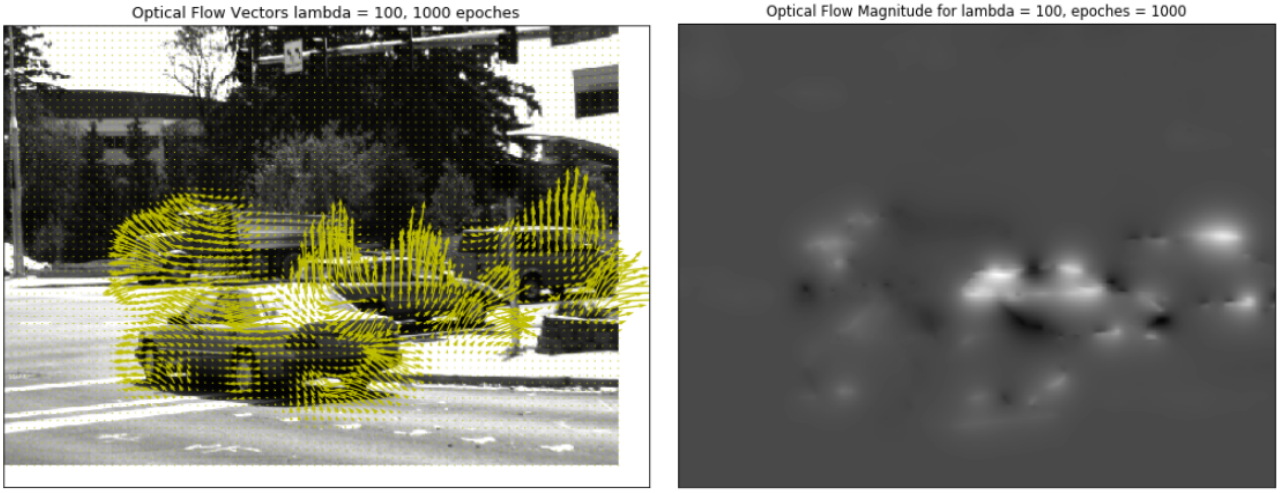


**Figure 1.40: Optical flow as vector and magnitude with lambda=100 , epochs= 1000**

Yes, it was difficult to get the outputs because of the run-time the algorithm took to process the optical flow vectors. As seen in the figure 1.37, 1.38, even though the lambda value is the same, the increase in the number of epochs yields better results. This is because the more the number of epochs, the more the convergence occurs. But, in the case of higher lambda values, a more number of epochs may be needed; Because lower the lambda, convergence rate is high. This can be seen when compared to figures 1.38 and 1.39. Even though the number of epochs are constant, the higher lambda value in the case of figure 1.39 gave better results with low epoches. However, if the lambda value increases a lot, the number of epochs needed might be even more, this might also cause problems of increase in time complexity as it would be better to land on global minimum error rather than local minimum which is what this method's aim is similar to Gradient descent algorithm.

## 1.4. Summary

I. Give your overall thoughts of the two methods:

• Do each have certain strengths and weaknesses, or is there one method that you find to be superior?

• Compare the methods in terms of difficulty in choosing parameters.

• Compare the methods (qualitatively) in terms of run time.

• Were both methods equally challenging to implement. Did you find one to be easier?

**SOLUTION:**

Even though it is persuasive enough to say that Horn Schunck method is superior to the Lucas method, it has its own disadvantages. Horn Schunck algorithm outputs very high density of optical flow vectors u, v. This also considers the flow of even smaller objects as the boundary of motion. But, this method is more sensitive to the noise(if it exists) in the image than the local temporal difference. On the other hand, Lucas Kanade methodology assumes that the motion is small. Thus, if there is a large temporal difference (say more than 10 pixels in corresponding pixel objects) between two consecutive frames, then the technique fails. In that case, we might have to consider reducing the pixel resolution of the images and rerunning the algorithm again.

In the case of the Lucas method, neighborhood size is the main parameter that matters. The more the neighborhood size, more the equations to calculate the flow vectors. However, as mentioned, Lucas Kanade methodology assumes that the motion is small. Thus, if the motion is large enough, then reducing the pixel resolution and then applying the Lucas method to be considered. Whereas, in the case of Horn Schunck, the values of lambda and number of epochs ran, matters a lot. As seen in the figure 1.37, 1.38, even though the lambda value is the same, the increase in the number of epochs yields better results. This is because the more the number of epochs, the more the convergence occurs. But, in the case of higher lambda values, a more number of epochs may be needed; Because lower the lambda, convergence rate is high. This can be seen when compared to figures 1.38 and 1.39. Even though the number of epochs are constant, the higher lambda value in the case of figure 1.39 gave better results with low epoches. However, if the lambda value increases a lot, the number of epochs needed might be even more, this might also cause problems of increase in time complexity as it would be better to land on global minimum error rather than local minimum which is what this method's aim is similar to Gradient descent algorithm. This is also why Lucas Kanade's method takes less time than that of the Horn Schnuck process to find the optical flow vectors. But, if in the case of higher neighborhood size, then Lucas' method might take higher time to finish with respect to Horn Schunck.

I personally found Lucas' method to be easier to implement. This is because this needs no optimization process and everything happens in one single epoch. However, this needed several trial and error methods in choosing proper neighborhood size to calculate the optical flow vectors.

## APPENDIX

## PYTHON CODE FOR LUCAS KANADE AND HORN SCHUNCK

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import math
from skimage import io
from skimage.color import rgb2gray
import matplotlib.patches as patches


def importImage(path, resize):
    #status:complete
    #task: imports image, converts it into grayscale and returns
image, #rows, #cols
    #parameter: path, resize=true/false
    #returns: grayscaled image

    im = io.imread(path)
    im = rgb2gray(im)
    if(resize==True):
        scale_percent = 50 # percent of original size
        width = int(im.shape[1] * scale_percent / 100)
        height = int(im.shape[0] * scale_percent / 100)
        dim = (width, height)
        im = cv2.resize(im, dim, interpolation = cv2.INTER_AREA)
    im = ((im - np.min(im)) * (1/(np.max(im) - np.min(im))
*1.0)).astype('float')
    return im


def adjustInputs(image, Mask):
    #status: complete
    #task: adjust inputs by padding
    #parameters: image, Mask=kernel
    #returns: x pad size, y pad size, Mask
    #output_size= (input_size- filter_size + 2*Padding_size)+1
```

```
    #if input and output size are same: then
Padding_size=(filter_size-1)/2 (ceil it)

    (init_x_size, init_y_size)=image.shape
    Mask=np.flipud(np.fliplr(Mask))
    x_pad_size=math.ceil((Mask.shape[0]-1)/2)
    y_pad_size=math.ceil((Mask.shape[1]-1)/2)
    pad_sizes=[x_pad_size, y_pad_size]
    image=np.pad(image, (max(pad_sizes)), mode='constant')
    return x_pad_size, y_pad_size, image, Mask, max(pad_sizes)


def CommonUtil(image, init_x_size, init_y_size, max_pad, x_pad,
y_pad, Mask, isGrad):
    #status: complete
    #task: utility function to multiply kernel to image patch
    #output: adjusted image

    if(isGrad):
        grad_output_image= np.zeros((init_x_size+2*y_pad,
init_y_size+2*x_pad))
    else:
        grad_output_image= np.zeros((init_x_size, init_y_size))
    out_row=0
    out_col=0
    for x_pivot in range(x_pad, init_x_size+x_pad):
        out_col=0
        for y_pivot in range(y_pad, init_y_size+y_pad):
            patch=image[x_pivot-max_pad: x_pivot+max_pad+1,
y_pivot-max_pad: y_pivot+max_pad+1]
            grad_output_image[out_row][out_col] =
np.sum(np.multiply(Mask, patch))
            out_col+=1
        out_row+=1
    return grad_output_image


def Convolve(image, Mask):
    #status: complete
```

```
    #task: perform Convolution or CrossCorrelation
    #parameters: image, Mask=kernel, opType= True if Convolution
False if CrossCorrelation
    #returns: convolved image

    (init_x_size, init_y_size)=image.shape
    x_pad, y_pad, image, Mask, max_pad=adjustInputs(image, Mask)
    output_image= CommonUtil(image, init_x_size, init_y_size,
max_pad, x_pad, y_pad, Mask, False)
    return output_image


def FindDerivativeImages(image, xDerivativeMask, yDerivativeMask):
    #status: complete
    #task: finds x and y derivatives, magnitude of the derivatives of
the image
    #parameters: image, xDerivative, yDerivative
    #returns: xgradient, ygradient, magnitude

    init_x_size, init_y_size=image.shape
    x_pad, y_pad, ximage, xDerivativeMask,
max_pad=adjustInputs(image, xDerivativeMask)
    xgrad_output_image = CommonUtil(ximage, init_x_size, init_y_size,
max_pad, x_pad, y_pad, xDerivativeMask, True)
    x_pad, y_pad, yimage, yDerivativeMask,
max_pad=adjustInputs(image, yDerivativeMask)
    ygrad_output_image = CommonUtil(yimage, init_x_size, init_y_size,
max_pad, x_pad, y_pad, yDerivativeMask, True)
    return xgrad_output_image, ygrad_output_image


def displayCorner(nRows, nCols, imgArray, nameArray):
    #status: complete
    #params: nrows, ncols, images array, names of the images,
    #returns: subplots of given format

    c = 1  # initialize plot counter
    fig = plt.figure(figsize=(14,10))
    for i in imgArray:
```

```
        plt.subplot(nRows, nCols, c)
        plt.title(nameArray[c-1])
        plt.imshow(imgArray[c-1], cmap='gray')
        c = c + 1
    plt.show()


def optical_flow_kankade(image1, image2, frame_size):
    #status: complete
    #params: img2, img2, neighborhood size
    #returns: LUCAS OPTICAL FLOW ARRAYS u, v

    x_sobel_filter=(np.array([[-1, 0, 1]]))
    y_sobel_filter=np.array([[-1, 0, 1]]).T
    flowsx=[]
    flowsy=[]
    for i in range(frame_size//2, image1.shape[0]-frame_size//2+1):
        out_rowx=[]
        out_rowy=[]
        for j in range(frame_size//2,
image1.shape[1]-frame_size//2+1):
            frame1=[]
            frame2=[]
            A=[]#for every fs**2 points
            B=[]
            for k in range(-frame_size//2, frame_size//2):
                flow_row1=[]
                flow_row2=[]
                for l in range(-frame_size//2, frame_size//2):
                    flow_row1.append(image1[i+k][j+l])
                    flow_row2.append(image2[i+k][j+l])
                frame1.append(flow_row1)
                frame2.append(flow_row2)
            Ix, Iy=np.gradient(np.matrix(frame1, dtype='float'))
            Ix=np.reshape(Ix, (frame_size**2, 1))
            Iy=np.reshape(Iy, (frame_size**2, 1))
            A=np.hstack((Ix, Iy))
            It=np.matrix(frame2, dtype='float')-np.matrix(frame1,
dtype='float')
```

```
            B=-1*np.reshape(It, (frame_size**2, 1))
            linalg=np.linalg.lstsq(A, B)[0]
            u, v=(linalg[0][0], linalg[1][0])
            out_rowx.append(u)
            out_rowy.append(v)
        flowsx.append(out_rowx)
        flowsy.append(out_rowy)
    return np.array(flowsx), np.array(flowsy)


def print_vectors(im1, subsample, x, y, frame_size, HS, lbda=0.01,
epoches=1000):
    #status: complete
    #params: img2, subsample, u, v flow vectors, neighborhood size
    #returns: quiver plot of u, v vectors

    rows=im1.shape[0]
    cols=im1.shape[1]
    sub_u = y[0:rows:subsample, 0:cols:subsample]
    sub_v = -x[0:rows:subsample, 0:cols:subsample]
    xc = np.linspace(0, cols, sub_u.shape[1])#changed rows
    yc = np.linspace(0, rows, sub_v.shape[0])
    # Locations of the vectors
    xv, yv = np.meshgrid(xc, yc)
    fig1 = plt.figure(figsize = (10,7))
    plt.imshow(im1 ,cmap = 'gray')
    if(HS==False):
        plt.title('LUCAS Optical Flow with Frame size:
'+str(frame_size)), plt.xticks([]), plt.yticks([])
    else:
        plt.title('Optical Flow Vectors for lamda = ' + str(lbda) + "
epochs = "+ str(epoches))
    # Plot the vectors
    plt.quiver(xv, yv, sub_u, sub_v, color='y')


def calcUtil(frame1, frame2):
    #status: complete
    #task: calculates Ix, Iy, It
```

```
    Ix=(1/4)*(frame1[1][0]+frame2[1][0]+frame1[1][1]+frame2[1][1]) -
(1/4)*(frame1[0][0]+ frame2[0][0] + frame1[0][1]+frame1[0][1])
    Iy=(1/4)*(frame1[0][1]+frame2[0][1]+frame1[1][1]+frame2[1][1]) -
(1/4)*(frame1[0][0]+ frame2[0][0] + frame1[1][0]+frame2[1][0])
    It=(1/4)*(frame2[0][0]+frame2[0][1]+frame2[1][0]+frame2[1][1]) -
(1/4)*(frame1[0][0]+ frame1[0][1] + frame1[1][0]+frame1[1][1])
    return Ix, Iy, It


def utility_flow(image1, image2, U, V):
    #status:complete
    #task: computes u, v

    flowsx=[]
    flowsy=[]
    for i in range(0, image1.shape[0]-1):
        out_rowx=[]
        out_rowy=[]
        for j in range(0, image1.shape[1]-1):
            frame1=[]
            frame2=[]
            A=[]#for every fs**2 points
            B=[]
            for k in range(0, 2):
                flow_row1=[]
                flow_row2=[]
                for l in range(0, 2):
                    flow_row1.append(image1[i+k][j+l])
                    flow_row2.append(image2[i+k][j+l])
                frame1.append(flow_row1)
                frame2.append(flow_row2)
            Ix, Iy, It=calcUtil(frame1, frame2)
            numerator = (Ix*U[i][j] + Iy*V[i][j] + It)
            denominator = (lbda**2 + Ix**2 + Iy**2)
            U[i][j] = U[i][j] - Ix*(numerator/denominator)
            V[i][j] = V[i][j] - Iy*(numerator/denominator)
    return U, V
```

```
def HornSchunckFlow(img1, img2, lbda, epochs):
    #status: complete
    #params: image1, image2, lambda, epochs
    #returns: U, V calculated using Horn Schunck process

    U, V = (np.zeros((img1.shape[0], img1.shape[1])),
np.zeros((img1.shape[0],img1.shape[1])))
    for epoch in range (epochs):
        print(epoch)
        U, V=utility_flow(img1, img2, U, V)
    return U, V


def main(image_paths, resize):
#     status: incomplete
#     params: conv_image_path= image path for convolution,
filter_size= filter size
#     1. partial derivative using sobel

    image1=importImage(image_path[0], resize)
    x_sobel_filter=(np.array([[-1, 0, 1]]))
    y_sobel_filter=np.array([[-1, 0, 1]]).T
    xGradImage, yGradImage = FindDerivativeImages(image1,
x_sobel_filter, y_sobel_filter)
    image2=importImage(image_path[1], resize)
    temp_diff=image1-image2
    image_list=[image1, image2, xGradImage, yGradImage, temp_diff]
    names_list=["I-Image Imported", "II-Image Imported", "xGradient",
"yGradient", "temporal partial derivative"]
    displayCorner(2, 3, image_list, names_list)


    #2. LUCAS KANADE
    frame_sizes=[7, 7]
    for frame_size in frame_sizes:
        image1=importImage(image_path[0], resize)
        image2=importImage(image_path[1], resize)
        x, y=optical_flow_kankade(image1, image2, frame_size)
```

```
        y=np.squeeze(y)
        x=np.squeeze(x)
        plt.imshow(x, cmap='gray')
        plt.imshow(y, cmap='gray')
        plt.title("LUCAS KANADE MAGNITUDE WITH FRAME SIZE: " +
str(frame_size))
        image1=importImage(image_path[0], resize)
        print_vectors(image1, 5, x, y, frame_size, False)

    #3.HORN SCHUNK
    image1=importImage("traffic0.png", False)
    image2=importImage("traffic1.png", False)
    lbda=0.1
    epochs=8
    U, V = HornSchunckFlow(image1, image2, lbda, epochs)
    plt.imshow(U, cmap='gray')
    plt.imshow(V, cmap='gray')
    plt.title('Optical Flow Magnitude for lambda = 10, epoches =
500'), plt.xticks([]), plt.yticks([])
if __name__=="__main__":
    image_path=["sphere0.png", "sphere1.png"] #change the image names
to toggle between problems
    main(image_path, True)
```