Explanation

Optical Flow using Lukas-Kanade Sparse Optical Flow Model

To perform sparse optical flow with Lucas Kanade optical flow model for the pixels of interest which are derived from Shi Tomashi corner detection algorithm, the following steps were done.

1. **Importing the necessary libraries:**

Program starts with importation of the required libraries which are NumPy and OpenCV. These libraries are shown below:

```
# Import the required libraries

import cv2

import NumPy as np
```

2. **Reading the video from its location and highlighting the corners with colors:**

The main script starts by reading the video file into an instance with the help of the Video capture function in the open CV. Then we use NumPy array for random color creation which helps highlight the corners. Furthermore , this captured video from source is then read across the first instance of the video. This is captured below:

```
#driver script

capture = cv2.VideoCapture("traffic.mp4")

#lk_params = dict(winSize=(15,15),maxLevel=2,criteria=(cv2.TERM_CRITERIA_EPS|cv2.TERM_CRITERIA_COUNT,10,0.03))

color = np.random.randint(0,255,(100,3))

ret,prev_frame = capture.read()
```

3. **Converting the first captured frame in gray scale**

The first frame is converted to the gray scale image with the help of the cvt color function in the OpenCV. Features from the first gray image are extracted with the help of the user defined function. The Shi-Tomasi function is applied via a created function call with help find the good corners from the input image. The function to implement the Shi-Tomasi function is shown below:

tracking features. Identifying corners through Shi-Tomasi corner detector method

cv2.goodFeaturesToTrack OpenCV function is used find the features using Shi-Tomasi corner detector algorithm

```
def find_features(gray_images):

    points = cv2.goodFeaturesToTrack(gray_images,mask = None, maxCorners = 100,qualityLevel = 0.2,minDistance = 10,blockSize = 7)

    return points
```

Some parameters used in the Shi-Tomasi function implementation are extracted from the reference manual of OpenCV as described below:

gray_image - Input image in which features has to be detected
maxCorners - If there are more than 100 corners detected, then only strongest 100 is returned
qualityLevel - To determine the minimum accepted quality of a corner i.e., weak corner
minDistance - Minimum distance between two corners that is returned
points - Returns the feature points using Shi-Tomasi corner detection algorithm
blockSize - computes the covariance matrix over each pixel neighborhood

We create the output video instance with the help of the in-built video writer function in the OpenCV which takes the output file name, file format, frames per second, frame dimensions which is taken same as the input video.

```
#driver script

capture = cv2.VideoCapture("traffic.mp4")

#lk_params = dict(winSize=(15,15),maxLevel=2,criteria=(cv2.TERM_CRITERIA_EPS|cv2.TERM_CRITERIA_COUNT,10,0.03))

color = np.random.randint(0,255,(100,3))

ret,prev_frame = capture.read()
```

The output of this function returns the points which are considered as corner in the input image. An image is created by the same size as the first frame of the video with zeros as input using a masked image to populate it with zeros as seen below:

```
prev_gray= cv2.cvtColor(prev_frame,cv2.COLOR_BGR2GRAY)

prev_points = find_features(prev_gray)
```

mask_image = np.zeros_like(prev_frame)

output_frames = []

4. Reading Further Frames

A while loop is created to read the further frames of the video file where each of next frame of the video is read for every repetition of the while loop.

If no frame is detected, the while loop breaks using this line of code

```
ret,frame = capture.read()

#Terminate when no frame is detected

if not ret:

    break
```

otherwise, that frame is converted to grayscale with the help of

```
current_gray = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
```

This is akin to Converting the current frame to gray scale. The position of the points from the current image relative to the features obtained from the detector function are passed into the Lukas-Kanade function which utilize the sparse optical flow model to find the points in the current image:

```
#Lucas Kanade sparse optical flow model
def LK_model(gray_t_1,gray_t,prev_points):
    points,status,error = cv2.calcOpticalFlowPyrLK(gray_t_1,gray_t,prev_points,None,winSize = (20,20),maxLevel=2,criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 0.04))

    return (points,status,error)
```

The following argument are used in this function:

gray_t_1 - Previous frame image in gray scale
gray_t - Current frame image in gray scale
winSize - Size of search window
maxLevel - Maximum pyramid level. Since set to 2, three levels of pyramid are used
criteria – The termination criteria for algorithm when the maximum number of repetitions is reached, or search window moves less than criteria

```
current_gray = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
current_points,status,error = LK_model(prev_gray,current_gray,prev_points)
```

Function returns the feature points in the current image along with the respective status & errors or we can simply say that The function returns the next points, status whether correspondence is found and the error

```
#extracting successful points
if current_points is not None:
    current_points = current_points[status ==1]
    old_points = prev_points[status==1]

#performing mapping
for i, (current,prev) in enumerate(zip(current_points,old_points)):
    a,b = current.ravel()
    c,d = prev.ravel()
```

Next, there is the creation of a line in the mask image between the feature points of the previous image & current image with the help of the line function in Open CV. This will create black screen where lines are drawn from a-d to track the flow of the pixels

```
mask_image = cv2.line(mask_image,(int(a),int(b)),(int(c),int(d)),color[i].tolist(),2)
```

We fill out the original version of the current images with circle to highlight the feature points of the current as well as the previous images and then we add the masked image on top of the original image as shown below;

```
# Draw a circle for each current point

    frame = cv2.circle(frame,(int(a),int(b)),5,color[i].tolist(),-1) img = cv2.add(frame,mask_image)

  #image = frame



  cv2.imshow('frame',img)

  output_frames.append(img)

  #cv2.imshow('mask image',mask_image)

  #cv2.imshow('mask image',image)
```

We now write this image to an output video and replace the previous images and points with the current feature points and images so that the while loop repeats the procedure for the next frame

```
#set the current frame as the t-1 frame and update the current points at the prvious points (t-1). Basically updating the frame
    prev_gray = current_gray.copy()
    prev_points = current_points.reshape(-1,1,2)


capture.release()


cv2.destroyAllWindows()


height,width,_ = img.shape
size = (width,height)


# Creating object for appending to the video
output = cv2.VideoWriter('Traffic_Optical_Flow.mp4',cv2.VideoWriter_fourcc('F','M','P','4'), 30, size)
# Writing the video by repeating through each frame
for image in output_frames:
    output.write(image)
```