Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)
Image Processing and Computer Vision I (DJ19DSL603)

JHANVI PAREKH

60009210033

CSE(DATA SCIENCE)

Experiment 10:

AIM: Lucas-Kanade optical flow estimation Algorithm in image sequences

Objective: The objective of this laboratory session is to understand and implement the Lucas-Kanade algorithm for optical flow estimation.

Theory: The Lucas-Kanade algorithm is a widely used method for estimating optical flow, which represents the apparent motion of objects between consecutive frames in a video sequence. It works by assuming that the motion between frames is approximately constant within a local neighborhood of each pixel. By solving a system of linear equations, the algorithm estimates the motion vectors for each pixel, providing a dense optical flow field.

Procedure:

1. **Understanding Optical Flow:**
   - Optical flow represents the movement of objects in an image sequence over time.
   - It is typically represented by a 2D vector field, where each vector corresponds to the motion of a point in the image.
2. **Lucas-Kanade Algorithm Overview:**
   - The Lucas-Kanade algorithm estimates optical flow by considering a small window of pixels around each point.

- It assumes that the motion within this window is approximately constant.
- By minimizing the difference between the intensities of corresponding pixels in consecutive frames, the algorithm computes the motion vectors.

3. Implementing Lucas-Kanade Algorithm:

- Choose a suitable programming language and environment for implementation (e.g., Python with OpenCV).
- Load the image sequence or video file.
- Preprocess the images if necessary (e.g., convert to grayscale).
- Define the window size and other parameters for Lucas-Kanade algorithm.
- Implement the Lucas-Kanade algorithm using the following steps:
    - Compute image gradients using Sobel or other edge detection methods.
    - Compute the spatial gradients of the image and the temporal gradient between consecutive frames.
    - Estimate the motion vectors using least squares or other optimization techniques.
- Visualize the resulting optical flow field.

4. Testing and Evaluation:

- Test the implemented Lucas-Kanade algorithm on sample image sequences or video files.
- Evaluate the accuracy and performance of the optical flow estimation.
- Compare the results with ground truth or other optical flow estimation methods if available.
- Analyze any limitations or challenges encountered during implementation.

5. Extensions:

- Experiment with different parameter settings (e.g., window size, pyramid levels) to observe their effects on optical flow estimation.

- Explore advanced variants of the Lucas-Kanade algorithm, such as the pyramidal Lucas-Kanade method or its extensions for robust motion estimation.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)
Image Processing and Computer Vision I (DJ19DSL603)

Lab 5: Image Enhancement in Spatial Domain using Neighbourhood Processing Techniques

Lab Assignments to complete in this session

Problem Statement: Develop a Python program utilizing the OpenCV library to manipulate images from the Fashion MNIST digits dataset. The program should address the following tasks:

1. Importing libraries
2. Read random image(s) from the given Gif image derived dataset.
3. Get intensity change in X-Direction
4. Get intensity change in Y-Direction
5. Get intensity change in T-Direction
6. Track Region
7. Estimate optical flow in image sequences
8. Visualize optical flow

The solution to the operations performed must be produced by scratch coding without the use of built in OpenCV methods.

Department of Computer Science and Engineering (Data Science)
Image Processing and Computer Vision I (DJ19DSL603)

About Optical Flow

Optical flow refers to the pattern of apparent motion of objects in an image or sequence of images. It is a crucial concept in computer vision and image processing, providing information about how objects move within a scene over time. The primary goal of optical flow analysis is to track the movement of pixels from one frame to the next, estimating the velocity or displacement of objects in an image sequence.

Key points about optical flow include:

- **Motion Estimation**: Optical flow is used to estimate the motion of objects by analyzing the changes in intensity or color of pixels between consecutive frames in a video sequence. It can be applied to both 2D and 3D motion analysis.
- **Assumptions**: Optical flow methods are based on certain assumptions, such as brightness constancy (the intensity of a pixel remains constant between frames) and smoothness of motion (neighboring pixels exhibit similar motion). These assumptions help simplify the complex task of motion estimation.
- **Mathematical Representation**: Optical flow is often represented as a vector field, where each pixel in an image corresponds to a vector indicating the direction and magnitude of motion. This representation allows for a quantitative analysis of how objects move within a scene.
- **Applications**: Optical flow finds applications in various fields, including computer vision, robotics, and video processing. It is utilized in tasks such as object tracking, video stabilization, action recognition, and autonomous navigation for vehicles and drones.
- **Challenges**: Optical flow estimation can be challenging in the presence of occlusions, fast motion, or changes in lighting conditions. Robust algorithms and techniques are required to address these challenges and provide accurate motion analysis.
- **Methods**: Different algorithms are employed for optical flow computation, ranging from traditional techniques like Lucas-Kanade method to more advanced approaches using deep learning. Dense optical flow methods estimate motion for every pixel in the image, while sparse methods focus on specific points of interest.
- **Limitations**: Optical flow has its limitations, especially in scenarios where the assumptions do not hold or when dealing with complex scenes. Ambiguities and inaccuracies may arise, and integrating additional information or combining optical flow with other computer vision techniques can enhance its reliability.

In summary, optical flow is a fundamental concept in computer vision, enabling the analysis of motion in images and videos. Its applications are diverse, contributing to advancements in fields such as robotics, video processing, and artificial intelligence. Researchers continue to develop and refine optical flow methods to address challenges and improve the accuracy of motion estimation in various real-world scenarios. In this experiment we will work on **Lucas-Kanade** algorithm to perform optical flow.

+ Code + Text

Diving into Lucas-Kanade Algorithm

Lucas Kanade algorithm is an optical flow based feature tracking technique for video. Starting with an assumption that the tracked feature's movement between two sequential frames is very small, this algorithm attempts to calculate the velocity vector (u, v) or every tracked feature. The features are usually represented by $n \times n$ pixel blocks. At the first frame, initial positions of these features are given as algorithm inputs. For every next frame the algorithm updates those positions using a velocity vector calculated at every frame.

The algorithm assumes that the following equation is valid for any pixel in any tracked feature:

$$\Delta B_x(x,y) \cdot v(x) + \Delta B_y(x,y) \cdot v_(y) = -\Delta B_t(x,y)$$

where:

- $\Delta B_x(x,y)$ is the increase in brightness in x direction at pixel position (x,y).
- $\Delta B_y(x,y)$ is the increase in brightness in y direction at pixel position (x,y).
- $\Delta B_t(x,y)$ is the increase in brightness between the pixel at position (x,y) in the next frame t_2 and the current frame t_1

The equation above refers to a single pixel inside the tracked region. For all pixels inside the tracked feature, a set of such equations is needed. These equations can be written in a matrix form:

$$\begin{bmatrix} \Delta B_{x,1}(x,y) & \Delta B_{y,1}(x,y) \\ \Delta B_{x,2}(x,y) & \Delta B_{y,2}(x,y) \\ \dots & \dots \\ \Delta B_{x,3}(x,y) & \Delta B_{y,3}(x,y) \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} \Delta B_{t,1}(x,y) \\ \Delta B_{t,2}(x,y) \\ \dots \\ \Delta B_{t,3}(x,y) \end{bmatrix}$$

The equation above can be also written as:

$$Sv = t$$

The velocity vector v can be calculated from this expression. Since this system of linear equations is overdetermined, v can be calculated using the least-squares method:

$$v = (SS^T)^{-1} S^T t$$

Once v for a tracked feature is known, its position can be updated for the next frame.

Assumptions of Lucas Kanade Method

The algorithm is based on certain assumptions to simplify the problem of motion estimation. Here are the key assumptions made by the Lucas-Kanade algorithm:

- **Brightness Constancy**: The algorithm assumes that the intensity or brightness of a pixel remains constant over time. In other words, for a pixel corresponding to a specific point in a scene, the change in intensity between frames is primarily due to motion, not changes in illumination.
- **Spatial Coherence**: Lucas-Kanade assumes that the motion of neighboring pixels is similar. This assumption implies that the motion in a local neighborhood is relatively smooth and consistent. It allows the algorithm to estimate the motion at a particular point by considering the motion in the surrounding area.
- **Temporal Coherence**: The algorithm assumes that the motion between consecutive frames is small. This assumption is based on the idea that the motion in a short time interval is approximately constant. Consequently, the algorithm performs well for small displacements and may not be as accurate for large motion.
- **Small Motion Displacements**: Lucas-Kanade is designed for scenarios where the motion between frames is small. It may not perform well when dealing with large displacements, as the linear approximation used by the algorithm becomes less accurate in such cases. Local Feature Tracking: The algorithm assumes that it is sufficient to track a single feature point (pixel) or a small set of feature points in a local neighborhood to estimate the overall optical flow. This localized approach simplifies the computation and allows for real-time performance.
- **Linear Motion Model**: The algorithm assumes a linear relationship between the optical flow parameters (motion in the x and y directions) and the image intensity gradients. This linear model is used to solve a system of linear equations for estimating the motion parameters.

It's important to note that while these assumptions simplify the optical flow estimation problem, they also impose limitations on the applicability of the Lucas-Kanade algorithm. In real-world scenarios where these assumptions may not hold, alternative optical flow methods or improvements to the Lucas-Kanade algorithm, such as incorporating pyramids or using robust techniques, may be employed to enhance accuracy and robustness.

Implementation

Importing required modules

```
[ ] import numpy as np
    import matplotlib.pyplot as plt
    import matplotlib.animation as animation
    from PIL import Image, ImageDraw
    from tqdm.auto import tqdm
```

Helper class to work with images and create results

```
[ ] class ImageHelper:
        def load_image(self, file_path):
            '''
            Load image from a file path into memory. Get the pixel array.
            '''
            im = Image.open(file_path)
            pixels = im.load()
            return (im, pixels)

        def draw_rectangle(self, img, xy,fill, outline=None):
            d = ImageDraw.Draw(img)
            d.rectangle(xy,fill,outline)
            return img

        def draw_line(self, img, coord, fill_color):
            draw = ImageDraw.Draw(img)
            draw.line(coord, fill=fill_color)
```

Implementing Lucas Kanade method

```
class LucasKanadeTracker:

    # Get intensity change in X direction
    def get_intensity_changes_for_box_x(self, grid, row, col, box_size):
        output = np.zeros((box_size, box_size))

        for r in range(box_size):
            for c in range(box_size):
                next_value = 0
                prev_value = 0

                if (c + 1 > box_size - 1):
                    next_value = grid[row + r][col + c]
                else:
                    next_value = grid[row + r][col + c + 1]

                if (c - 1 < 0):
                    prev_value = grid[row + r][col + c]
                else:
                    prev_value = grid[row + r][col + c - 1]

                # output[r][c] = next_value - grid[row + r][col + c]
                output[r][c] = (next_value * 1.00 - prev_value) / 2.0

        return output
```

Department of Computer Science and Engineering (Data Science) Image Processing and Computer Vision I (DJ19DSL603)

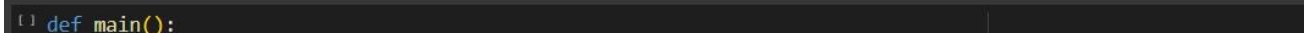
Lab

Department of Computer Science and Engineering (Data Science) Image Processing and Computer Vision I (DJ19DSL603)

```
    # Get intensity change in Y direction
    def get_intensity_changes_for_box_y(self, grid, row, col, box_size):
        output = np.zeros((box_size, box_size))
        for r in range(box_size):
            for c in range(box_size):
                next_value = 0
                prev_value = 0
                cur_value = grid[row + r][col + c]

                if (r + 1 > box_size - 1):
                    next_value = cur_value
                else:
                    next_value = grid[row + r + 1][col + c]

                if (r - 1 < 0):
                    prev_value = cur_value
                else:
                    prev_value = grid[row + r - 1][col + c]

                # output[r][c] = next_value - grid[row + r][col + c]
                output[r][c] = (next_value - prev_value) / 2.0
        return output

    # Get intensity change between time t1 and t2
    def get_intensity_changes_for_box_time(self, grid_t1, grid_t2, row, col, box_size):
        output = np.zeros((box_size, box_size))
        for r in range(box_size):
            for c in range(box_size):
                output[r][c] = -(grid_t2[row + r][col + c] - grid_t1[row + r][col + c])
        return output
```

Working with an example

The video presented below has been generated using a series of images located within the designated "images" directory.



```
[ ] def main():
        image_width, image_height = 400, 400

        # tracked regions (features):
        # starting point (top left coordinate)
        x0_arr = [178, 295, 250]
        y0_arr = [96, 175, 310]

        original_x0_arr, original_y0_arr = x0_arr[:], y0_arr[:]
        # tracked box width / height in pixels:
        box_size = 30

        # image dataset size:
        start_image_index = 1
        end_image_index = 23

        image_name_base = "sample1-"
        velocity_vector_arr = [None] * len(x0_arr)
        resulting_images = []
```

Department of Computer Science and Engineering (Data Science) Image Processing and Computer Vision I (DJ19DSL603)

Department of Computer Science and Engineering (Data Science) Image Processing and Computer Vision I (DJ19DSL603)

```
# for any image:
for img_index in tqdm(range(start_image_index, end_image_index), "Applying Lucas Kanade"):

    img_helper = ImageHelper()
    # full color images:
    (loaded_image_1, pixels_1) = img_helper.load_image("images/sample1-" + str(img_index) + ".png")
    (loaded_image_2, pixels_2) = img_helper.load_image("images/sample1-" + str(img_index + 1) + ".png")

    reduced_map_t1 = []  # map of grey pixels for image t1
    for i in range(image_height):
        reduced_map_t1.append([])
        for j in range(image_width):
            reduced_map_t1[i].append(None)

    for x in range(image_width):
        for y in range(image_height):
            avg = (pixels_1[x, y][0] + pixels_1[x, y][1] + pixels_1[x, y][2]) / 3.0
            reduced_map_t1[y][x] = avg

    reduced_map_t2 = []  # map of grey pixels for image t2
    for i in range(image_height):
        reduced_map_t2.append([])
        for j in range(image_width):
            reduced_map_t2[i].append(None)

    for x in range(image_width):
        for y in range(image_height):
            avg = (pixels_2[x, y][0] + pixels_2[x, y][1] + pixels_2[x, y][2]) / 3.0
            reduced_map_t2[y][x] = avg

   # for any tracked rectange:
   for point_index in range(len(x0_arr)):
       if velocity_vector_arr[point_index] is not None:
           x0_arr[point_index] += velocity_vector_arr[point_index][0][0]
           y0_arr[point_index] += velocity_vector_arr[point_index][1][0]

           x0_arr[point_index] = round(x0_arr[point_index])
           y0_arr[point_index] = round(y0_arr[point_index])
           x0_arr[point_index] = int(x0_arr[point_index])
           y0_arr[point_index] = int(y0_arr[point_index])

       lucas_kanade = LucasKanadeTracker()
       changes_x = lucas_kanade.get_intensity_changes_for_box_x(reduced_map_t1,
                                                                y0_arr[point_index], x0_arr[point_index], box_size)
       changes_y = lucas_kanade.get_intensity_changes_for_box_y(reduced_map_t1, y0_arr[point_index],
                                                                x0_arr[point_index],
                                                                box_size)
       changes_t = lucas_kanade.get_intensity_changes_for_box_time(reduced_map_t1, reduced_map_t2,
                                                                   y0_arr[point_index], x0_arr[point_index],
                                                                   box_size)

       cropped_t1 = loaded_image_1.crop((x0_arr[point_index], y0_arr[point_index], x0_arr[point_index] + box_size,
                                         y0_arr[point_index] + box_size))
       cropped_t2 = loaded_image_2.crop((x0_arr[point_index], y0_arr[point_index], x0_arr[point_index] + box_size,
                                         y0_arr[point_index] + box_size))

       # if we want to display the tracked region only:
       # cropped_t1.show()
       # cropped_t2.show()

       # compute velocity vector:

       # form S matrix now:
       flatten_x = changes_x.flatten()
       transpose_changes_x = np.array([flatten_x]).transpose()
```

Department of Computer Science and Engineering (Data Science) Image Processing and Computer Vision I (DJ19DSL603)

```
            flatten_y = changes_y.flatten()
            transpose_changes_y = np.array([flatten_y]).transpose()

            flatten_t = changes_t.flatten()
            transpose_changes_t = np.array([flatten_t]).transpose()

            s_matrix = np.concatenate(np.array([transpose_changes_x, transpose_changes_y]), axis=1)
            s_matrix_transpose = s_matrix.transpose()
            t_matrix = transpose_changes_t
            st_s = np.matmul(s_matrix_transpose, s_matrix)

            w, v = np.linalg.eig(st_s)
            cond = abs(w[0]) / abs(w[1])

            st_s_inv = np.linalg.inv(st_s)
            temp_matrix = np.matmul(st_s_inv, s_matrix_transpose)
            velocity_vector_arr[point_index] = np.matmul(temp_matrix, t_matrix)

            # draw  scaled velocity vector:
            line_scale_factor = 30

            line_x0 = x0_arr[point_index] + box_size / 2
            line_y0 = y0_arr[point_index] + box_size / 2

            img_helper.draw_line(loaded_image_1,
                                 (
                                     line_x0, line_y0,
                                     line_x0 + velocity_vector_arr[point_index][0][0] * line_scale_factor,
                                     line_y0 + velocity_vector_arr[point_index][1][0] * line_scale_factor),
                                 (255, 0, 0))

            # draw the original position:
            img_helper.draw_rectangle(loaded_image_1,
                                      [original_x0_arr[point_index], original_y0_arr[point_index],
                                       original_x0_arr[point_index] + box_size,
                                       original_y0_arr[point_index] + box_size], fill=None,
                                      outline=(100, 100, 100))

            # draw the tracked box:
            img_helper.draw_rectangle(loaded_image_1,
                                      [x0_arr[point_index], y0_arr[point_index], x0_arr[point_index] + box_size,
                                       y0_arr[point_index] + box_size], fill=None,
                                      outline=(255, 255, 255))
        resulting_images.append(loaded_image_1)
    return resulting_images
```

```
[ ] images = main()
    gif = []
    for image in images:
        gif.append(image.convert("P",palette=Image.ADAPTIVE))
    gif[0].save('result.gif', save_all=True,optimize=False, append_images=gif[1:], loop=0)
```

```
Applying Lucas Kanade: 100% 22/22 [00:08<00:00,  2.56it/s]
```

```
[ ] %%HTML
    <h2> Results </h2>
    <div style="text-align:center;">
        <img src="result.gif" />
    </div>
```

Department of Computer Science and Engineering (Data Science)
Image Processing and Computer Vision I (DJ19DSL603)

Results

Limitations

The Lucas-Kanade algorithm, while effective in certain scenarios, has limitations that can impact its performance in various situations. Here are some of the limitations of the Lucas-Kanade algorithm

- The algorithm assumes that pixel intensity remains constant between frames, which may not hold true in the presence of changing lighting conditions or non-rigid motion.
- The algorithm is designed for small displacements, and its accuracy may degrade in the presence of large motion, occlusions, or rapid changes in the scene.
- Lucas-Kanade is not well-suited for handling global motion, as it assumes that motion is locally smooth. This limitation can result in inaccuracies when dealing with complex scenes or significant camera movements.
- The algorithm estimates the apparent motion based on intensity changes, making it sensitive to texture patterns. It may not perform well in regions with low-texture or uniform areas.
- The accuracy of Lucas-Kanade is influenced by the availability and reliability of image gradients. Noisy or inaccurate gradients can lead to errors in motion estimation.
- The algorithm relies on parameters such as the size of the spatial neighborhood and the temporal window. Selecting appropriate parameter values can be challenging and may require manual tuning.
- Lucas-Kanade does not explicitly handle feature correspondence across frames. In cases where features change or disappear, the algorithm may struggle to provide accurate motion estimates.
- The algorithm assumes a stationary scene, and variations in the scene structure can lead to errors. It may not perform well in dynamic environments with evolving objects or scene changes.
- Lucas-Kanade is not designed to handle non-rigid motion well. It assumes that motion is well-described by an affine transformation, limiting its applicability in scenarios involving deformable objects.
- The algorithm may be sensitive to outliers, such as erroneous correspondences or occluded regions, leading to inaccurate motion estimates.