Homework 3: Lucas - Kanade Tracking

Srividya Gandikota
sgandiko
sgandiko@andrew.cmu.edu

1 Lucas-Kanade Tracking

1.1 Starting with an initial guess of p (for instance, $p = [0, 0]^T$), we can compute the optimal p^* iteratively. In each iteration, the objective function is locally linearized by first-order Taylor expansion,

$$I_{t+1}(x' + \Delta p) \approx I_{t+1}(x') + \frac{\partial I_{t+1}(x')}{\partial x'^T} \frac{\partial W(x;p)}{\partial p^T} \Delta p \tag{1}$$

where $\Delta p = [\Delta p_x, \Delta p_y]^T$ is the template offset. Further, $x' = W(x;p) = x + p$ and $\frac{\partial I(x')}{\partial x'^T}$ is a vector of the x and y image gradients at pixel coordinate x'. In a similar manner to Equation 3 one can incorporate these linearized approximations into a vectorized form such that,

$$\arg\max_{\Delta p} ||A\Delta p - b||_2^2 \tag{2}$$

such that $p \leftarrow p + \Delta p$ at each iteration.

1. What is $\frac{\partial W(x;p)}{\partial p^T}$? Write out the mathematical expression
   - Derivation Below:

     $$W(x;p) = \begin{bmatrix} 1 & 0 & p_1 \\ 0 & 1 & p_2 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + p_1 \\ y + p_2 \end{bmatrix}$$

     The derivative of this matrix is:

     $$\frac{\partial W(x;p)}{\partial p^T} = \begin{bmatrix} \frac{\partial W_x(x;p)}{\partial p_1^T} & \frac{\partial W_x(x;p)}{\partial p_2^T} \\ \frac{\partial W_y(x;p)}{\partial p_1^T} & \frac{\partial W_y(x;p)}{\partial p_2^T} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

2. What is A and b? Write out the mathematical expression
   - A (m x n) will become matrix of: $\frac{\partial I_{t+1}(x')}{\partial x'^T} \frac{\partial W(x;p)}{\partial p^T}$

     $$A = \begin{bmatrix} \frac{\partial I_{t+1}(x'_1)}{\partial x_1'^T} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{\partial I_{t+1}(x'_n)}{\partial x_n'^T} \end{bmatrix} \begin{bmatrix} \frac{\partial W(x_1;p)}{\partial p^T} \\ \vdots \\ \frac{\partial W(x_n;p)}{\partial p^T} \end{bmatrix}$$

   - b will become matrix of: $I_t(x) - I_{t+1}(x')$

     $$b = \begin{bmatrix} I_t(x_1) - I_{t+1}(x'_1) \\ \vdots \\ I_t(x_n) - I_{t+1}(x'_n) \end{bmatrix}$$

3. What conditions must $A^T A$ meet so that a unique solution to Δp can be found?

   - $A^T A$ must be both invertible and have a non zero determinant for a unique solution to be found. These requirements come from the need to find the inverse of the Hessian, which demands that the Hessian is invertible.

1.2 Implement a function with the following signature LucasKanade(It, It1, rect, p0 = np.zeros(2)) that computes the optimal local motion from frame It to frame It+1 that minimizes Equation 3.

My implementation for this function can be found in Figure 1 below.

```python
import numpy as np
from scipy.interpolate import RectBivariateSpline

def LucasKanade(It, It1, rect, threshold, num_iters, p0=np.zeros(2)):
    """
    :param It: template image
    :param It1: Current image
    :param rect: Current position of the car (top left, bot right coordinates)
    :param threshold: if the length of dp is smaller than the threshold, terminate the optimization
    :param num_iters: number of iterations of the optimization
    :param p0: Initial movement vector [dp_x0, dp_y0]
    :return: p: movement vector [dp_x, dp_y]
    """

    # Put your implementation here
    p = p0
    magnitude = 1
    count = 1

    #Get Image Shapes
    template_frame_height, template_frame_width = It.shape
    init_frame_height, init_frame_width = It1.shape

    #Set Up Rect Info
    x_1, y_1, x_2, y_2 = rect
    rect_height = int(x_2 - x_1)
    rect_width = int(y_2 - y_1)

    #Calculate Splines
    template_x = np.linspace(0, template_frame_height, template_frame_height, False)
    template_y = np.linspace(0, template_frame_width, template_frame_width, False)
    template_spline = RectBivariateSpline(template_x, template_y, It)

    init_x = np.linspace(0, init_frame_height, init_frame_height, False)
    init_y = np.linspace(0, init_frame_width, init_frame_width, False)
    init_spline = RectBivariateSpline(init_x, init_y, It1)

    # Creating a 2D grid of coordinates (x, y)
    complex_num = 1j
    x, y = np.mgrid[x_1:(x_2+1):(rect_width * complex_num), y_1:(y_2+1):(rect_height * complex_num)]

    while (magnitude > threshold) and (count < num_iters):
        #Partial Dirivatives
        movement_x = x + p[0]
        movement_y = y + p[1]
        dp_x = init_spline.ev(movement_y, movement_x, dy = 1).flatten()
        dp_y = init_spline.ev(movement_y, movement_x, dx = 1).flatten()

        It1_p = init_spline.ev(movement_y, movement_x).flatten()
        It_p = template_spline.ev(y, x).flatten()

        #Make Matrix A
        A = np.zeros((rect_width * rect_height, 2 * rect_width * rect_height))
        for i in range(rect_width * rect_height):
            A[i, (2 * i)] = dp_x[i]
            A[i, (2 * i + 1)] = dp_y[i]
        A = np.dot(A, (np.matlib.repmat(np.eye(2), rect_width * rect_height, 1)))

        #Make Matrix b
        b = np.reshape(It_p - It1_p, (rect_width * rect_height, 1))

        #Pseudoinverse Solve and Update
        magnitude = np.linalg.norm(np.linalg.pinv(A).dot(b))

        #New p Update
        p = (p + np.linalg.pinv(A).dot(b).T).ravel()
        count += 1

    return p
```

Figure 1: LucasKanade Function Implementation

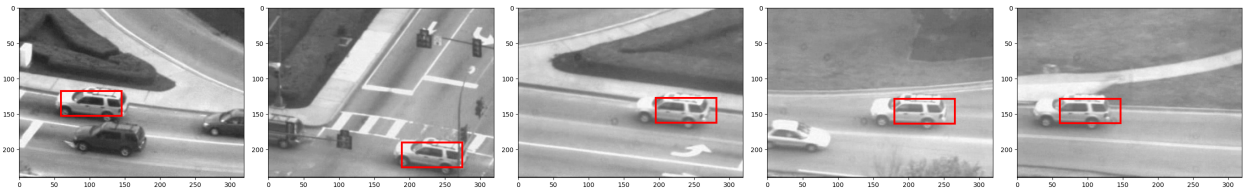
1.3 Write a script testCarSequence.py that loads the video frames from carseq.npy, and runs the Lucas-Kanade tracker that you have implemented in the previous task to track the car. Similarly, write a script testGirlSequence.py that loads the video from girlseq.npy and runs your Lucas-Kanade tracker on it.

My implementation for the testcarsequence function can be found in Figure 2 below. The results of the

```python
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# write your script here, we recommend the above libraries for making your animation
import LucasKanade

parser = argparse.ArgumentParser()
parser.add_argument('--num_iters', type=int, default=1e4, help='number of iterations of Lucas-Kanade')
parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-Kanade for terminating optimization')
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold

seq = np.load("../data/carseq.npy")
rect = [59, 116, 145, 151]

#Set up array for all boxes
_, _, frame = seq.shape
box_array = rect.copy()
for i in range(frame - 1):
    print(f'Frame ',(i+1), 'of ', (frame))

    #Get frames for analysis
    template_frame = seq[:, :, i]
    curr_frame = seq[:, :, i + 1]

    #Calculate Lucas Kanade Update
    movement_vec = LucasKanade.LucasKanade(template_frame, curr_frame, rect, threshold, num_iters)
    #print(movement_vec)
    for j in range(4):
        rect[j] += movement_vec[j % 2]
    box_array = np.vstack((box_array, rect))

    #Create Boxed Image
    if i == 0 or i == 99 or i == 199 or i == 299 or i == 399:
        print("True")
        plt.figure()
        plt.imshow(curr_frame, cmap='gray')
        patch = patches.Rectangle((rect[0], rect[1]), (rect[2] - rect[0]),
                                  (rect[3] - rect[1]), edgecolor='r', linewidth=3, facecolor='none')
        ax = plt.gca()
        ax.add_patch(patch)
        plt.savefig('../results/Frame_'+str(i + 1)+'.png', bbox_inches='tight')

np.save('carseqrects.npy', box_array)
```

Figure 2: testCarSequence Function Implementation

Lucas-Kanade tracker on the car can be found in Figure 3 below. Here we can see that the bounding box rectangle follows the car fairly well but as the car moves more, the box hangs off the edge of the car instead of accurately bounding against the entire car.



(a) Frame 1 (b) Frame 100 (c) Frame 200 (d) Frame 300 (e) Frame 400

Figure 3: Lucas-Kanade Tracking on Car with One Single Template

My implementation for the testgirlsequence function can be found in Figure 4 below.

```python
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# write your script here, we recommend the above libraries for making your animation
import LucasKanade

parser = argparse.ArgumentParser()
parser.add_argument('--num_iters', type=int, default=1e4, help='number of iterations of Lucas-Kanade')
parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-Kanade for terminating optimization')
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold

seq = np.load("../data/girlseq.npy")
rect = [280, 152, 330, 318]

#Set up array for all boxes
_, _, frame = seq.shape
box_array = rect.copy()
for i in range(frame - 1):
    print(f'Frame ',(i+1), 'of ', (frame))

    #Get frames for analysis
    template_frame = seq[:, :, i]
    curr_frame = seq[:, :, i + 1]

    #Calculate Lucas Kanade Update
    movement_vec = LucasKanade.LucasKanade(template_frame, curr_frame, rect, threshold, num_iters)
    #print(movement_vec)
    for j in range(4):
        rect[j] += movement_vec[j % 2]
    box_array = np.vstack((box_array, rect))

    #Create Boxed Image
    if i == 0 or i == 19 or i == 39 or i == 59 or i == 79:
        print("True")
        plt.figure()
        plt.imshow(curr_frame, cmap='gray')
        patch = patches.Rectangle((rect[0], rect[1]), (rect[2] - rect[0]),
                                  (rect[3] - rect[1]), edgecolor='r', linewidth=3, facecolor='none')
        ax = plt.gca()
        ax.add_patch(patch)
        plt.savefig('../results/GirlFrame_'+str(i + 1)+'.png', bbox_inches='tight')

np.save('girlseqrects.npy', box_array)
```

Figure 4: testGirlSequence Function Implementation

The results of the Lucas-Kanade tracker on the girl can be found in Figure 5 below. Here we can see that the bounding box rectangle follows the girl okay but as the girl moves more in front of more people, the box struggles to properly find the girl with the gradient difference. Once the girl leaves the region of other people, we notice that the box gets better at identifying her but still is not perfect.

(a) Frame 1 (b) Frame 20 (c) Frame 40 (d) Frame 60 (e) Frame 80

Figure 5: Lucas-Kanade Tracking on Girl with One Single Template

1.4 Write two scripts with a similar functionality to Q1.3 but with a template correction routine incorporated: testCarSequenceWithTemplateCorrection.py and testGirlSequenceWithTemplateCorrection.py. Save the rects as carseqrects-wcrt.npy and girlseqrects-wcrt.npy, and also report the performance at those frames.

My implementation for the testCarSequenceWithTemplateCorrection function can be found in Figure 6 below.

```python
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import LucasKanade

# write your script here, we recommend the above libraries for making your animation

parser = argparse.ArgumentParser()
parser.add_argument('--num_iters', type=int, default=1e4, help='number of iterations of Lucas-Kanade')
parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-Kanade for terminating optimization')
parser.add_argument('--template_threshold', type=float, default=5, help='threshold for determining whether to update template')
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold
template_threshold = args.template_threshold

seq = np.load("../data/carseq.npy")
rect = [59, 116, 145, 151]

#Set up array for all boxes
_, _, frame = seq.shape
box_array = rect.copy()
template_array = rect.copy()
template_frame = seq[:, :, 0]
move_vec_orig = np.zeros(2)

for i in range(frame-1):
    print(f'Frame ',(i+1), 'of ', (frame))
    #Get frames for analysis
    curr_frame = seq[:, :, i+1]

    #Calculate Lucas Kanade Update with Drift Correction
    movement_vec = LucasKanade.LucasKanade(template_frame, curr_frame, template_array, threshold, num_iters, move_vec_orig)
    adjusted_vec = movement_vec + [template_array[0] - rect[0], template_array[1]-rect[1]]
    updated_vec = LucasKanade.LucasKanade(seq[:, :, 0], curr_frame, rect, threshold, num_iters, adjusted_vec)
    delta_movement = np.linalg.norm(adjusted_vec - updated_vec)
    #print(delta_movement)

    if delta_movement < template_threshold:
        change_vec = (updated_vec - [template_array[0]-rect[0], template_array[1] - rect[1]])
        for j in range(4):
            template_array[j] += change_vec[j % 2]
        template_frame = seq[:, :, i+1]
        box_array = np.vstack((box_array, template_array))
        move_vec_orig = np.zeros(2)
    else:
        box_array = np.vstack(
            (box_array, [template_array[0]+movement_vec[0], template_array[1]+movement_vec[1], template_array[2]+movement_vec[0], template_array[3]+movement_vec[1]]))
        move_vec_orig = movement_vec
    #print(box_array)

    #Create Boxed Image
    carseq = np.load('carseqrects.npy')
    if i == 0 or i == 99 or i == 199 or i == 299 or i == 399:
        print("True")
        plt.figure()
        plt.imshow(curr_frame, cmap='gray')
        no_update = carseq[i + 1, :]
        updated = box_array[-1, :]
        patch_no_update = patches.Rectangle((no_update[0], no_update[1]), (no_update[2] - no_update[0]),
                                            (no_update[3] - no_update[1]), edgecolor='b', linewidth=3, facecolor='none')
        patch_update = patches.Rectangle((updated[0], updated[1]), (updated[2] - updated[0]),
                                         (updated[3] - updated[1]), edgecolor='r', linewidth=3, facecolor='none')
        ax = plt.gca()
        ax.add_patch(patch_no_update)
        ax.add_patch(patch_update)
        plt.savefig('../results/CarCorrectionFrame_' + str(i + 1)+'.png', bbox_inches='tight')

np.save('carseqrects-wcrt.npy', box_array)
```

Figure 6: testCarSequenceWithTemplateCorrection Function Implementation

The results of the Lucas-Kanade tracker on the car can be found in Figure 7 below. Here we can see that the blue no update template bounding box follows the car fairly well but as the car moves more, it lags behind and doesn't track the car well. The red bounding box uses the template correction method and we can see here that the box follows the car very well and follows the car much better than the no update version.

(a) Frame 1 (b) Frame 100 (c) Frame 200 (d) Frame 300 (e) Frame 400

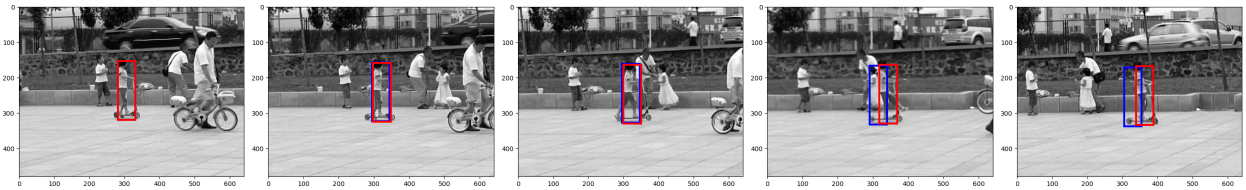
Figure 7: Lucas-Kanade Tracking on Car with Template Correction

My implementation for the testGirlSequenceWithTemplateCorrection function can be found in Figure 8 below.

```python
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import LucasKanade

# write your script here, we recommend the above libraries for making your animation

parser = argparse.ArgumentParser()
parser.add_argument('--num_iters', type=int, default=1e4, help='number of iterations of Lucas-Kanade')
parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-Kanade for terminating optimization')
parser.add_argument('--template_threshold', type=float, default=5, help='threshold for determining whether to update template')
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold
template_threshold = args.template_threshold

seq = np.load("../data/girlseq.npy")
rect = [280, 152, 330, 318]

#Set up array for all boxes
_, _, frame = seq.shape
print(seq.shape)
box_array = rect.copy()
template_array = rect.copy()
template_frame = seq[:, :, 0]
move_vec_orig = np.zeros(2)

for i in range(frame - 1):
    print(f'Frame ',(i+1), 'of ', (frame))
    #Get frames for analysis
    curr_frame = seq[:, :, i+1]

    #Calculate Lucas Kanade Update with Drift Correction
    movement_vec = LucasKanade.LucasKanade(template_frame, curr_frame, template_array, threshold, num_iters, move_vec_orig)
    adjusted_vec = movement_vec + [template_array[0] - rect[0], template_array[1]-rect[1]]
    updated_vec = LucasKanade.LucasKanade(seq[:, :, 0], curr_frame, rect, threshold, num_iters, adjusted_vec)
    delta_movement = np.linalg.norm(adjusted_vec - updated_vec)
    #print(delta_movement)

    if delta_movement < template_threshold:
        change_vec = (updated_vec - [template_array[0]-rect[0], template_array[1] - rect[1]])
        for j in range(4):
            template_array[j] += change_vec[j % 2]
        template_frame = seq[:, :, i+1]
        box_array = np.vstack((box_array, template_array))
        move_vec_orig = np.zeros(2)
    else:
        box_array = np.vstack(
            (box_array, [template_array[0]+movement_vec[0], template_array[1]+movement_vec[1], template_array[2]+movement_vec[0], template_array[3]+movement_vec[1]]))
        move_vec_orig = movement_vec
    #print(box_array)

    #Create Boxed Image
    girlseq = np.load('girlseqrects.npy')
    if i == 0 or i == 19 or i == 39 or i == 59 or i == 79:
        print("True")
        plt.figure()
        plt.imshow(curr_frame, cmap='gray')
        no_update = girlseq[i + 1, :]
        updated = box_array[-1, :]
        patch_no_update = patches.Rectangle((no_update[0], no_update[1]), (no_update[2] - no_update[0]),
                                            (no_update[3] - no_update[1]), edgecolor='b', linewidth=3, facecolor='none')
        patch_update = patches.Rectangle((updated[0], updated[1]), (updated[2] - updated[0]),
                                         (updated[3] - updated[1]), edgecolor='r', linewidth=3, facecolor='none')
        ax = plt.gca()
        ax.add_patch(patch_no_update)
        ax.add_patch(patch_update)
        plt.savefig('../results/GirlCorrectionFrame_' + str(i + 1)+'.png', bbox_inches='tight')

np.save('girlseqrects-wcrt.npy', box_array)
```

Figure 8: testGirlSequenceWithTemplateCorrection Function Implementation

The results of the Lucas-Kanade tracker on the girl can be found in Figure 9 below. Here we can see that the blue no update template bounding box follows the girl okay but as the girl moves more in front of more people, the box struggles to properly find the girl with the gradient difference. Once the girl leaves the region of other people, we notice that the box gets better at identifying her but still is not perfect. On the other hand, the red template correction bounding box can be seen to follow the child perfectly even while it crosses the other background people.



(a) Frame 1 (b) Frame 20 (c) Frame 40 (d) Frame 60 (e) Frame 80

Figure 9: Lucas-Kanade Tracking on Girl with Template Correction

2 Affine Motion Subtraction

2.1 Write a function with the following signature LucasKanadeAffine(It, It1) which returns the affine transformation matrix M (shape 3 × 3), and It and It1 are It and It+1 respectively. LucasKanadeAffine should be relatively similar to LucasKanade from the first section (you will probably also find scipy.ndimage.affine transform helpful).

My implementation for the LucasKanadeAffine function can be found in Figure 10 below.

```python
import numpy as np
from scipy.interpolate import RectBivariateSpline

def LucasKanadeAffine(It, It1, threshold, num_iters):
    """
    :param It: template image
    :param It1: Current image
    :param threshold: if the length of dp is smaller than the threshold, terminate the optimization
    :param num_iters: number of iterations of the optimization
    :return: M: the Affine warp matrix [2x3 numpy array] put your implementation here
    """
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
    p = np.zeros(6)
    error = 1
    count = 1

    #Get Image Shapes
    template_frame_height, template_frame_width = It.shape
    init_frame_height, init_frame_width = It1.shape

    #Calculate Splines
    template_x = np.linspace(0, template_frame_height, template_frame_height, False)
    template_y = np.linspace(0, template_frame_width, template_frame_width, False)
    template_spline = RectBivariateSpline(template_x, template_y, It)

    init_x = np.linspace(0, init_frame_height, init_frame_height, False)
    init_y = np.linspace(0, init_frame_width, init_frame_width, False)
    init_spline = RectBivariateSpline(init_x, init_y, It1)

    #Creating a mesh grid of coordinates (x, y)
    grid_x, grid_y = np.meshgrid(range(template_frame_width), range(template_frame_height))
    x = grid_x.flatten()
    y = grid_y.flatten()
    coords = np.vstack((x, y, np.ones(template_frame_height * template_frame_width)))

    while error > threshold and count < num_iters:
        count += 1
        #Update transformation matrix M with p
        M[0, 0] = 1 + p[0]
        M[0, 1] = p[1]
        M[0, 2] = p[2]
        M[1, 0] = p[3]
        M[1, 1] = p[4] + 1
        M[1, 2] = p[5]

        #Tranform and remove out of bounds coordinates
        warped_x = np.dot(M, coords)[0]
        warped_y = np.dot(M, coords)[1]
        bad_coords = find_bad_coordinates(warped_x, warped_y, template_frame_width, template_frame_height)
        valid_coords = np.logical_not(bad_coords)
        x_new = x[valid_coords]
        x_new = x_new[:, np.newaxis]
        y_new = y[valid_coords]
        y_new = y_new[:, np.newaxis]
        warped_x = warped_x[valid_coords]
        warped_x = warped_x[:, np.newaxis]
        warped_y = warped_y[valid_coords]
        warped_y = warped_y[:, np.newaxis]

        #Find partial derivatives
        dp_x = init_spline.ev(warped_y, warped_x, dy = 1).flatten()
        dp_x = dp_x[:, np.newaxis]
        dp_y = init_spline.ev(warped_y, warped_x, dx = 1).flatten()
        dp_y = dp_y[:, np.newaxis]
        It_p = template_spline.ev(y_new.flatten(), x_new.flatten()).flatten()
        It1_p = init_spline.ev(warped_y.flatten(), warped_x.flatten()).flatten()

        #Calculate matrices
        A = np.hstack((x_new * dp_x, y_new * dp_x, dp_x, x_new * dp_y, y_new * dp_y, dp_y))
        b = (It_p - It1_p).reshape(-1, 1)

        #Pseudoinverse Solve and Update
        magnitude = np.linalg.norm(np.linalg.pinv(A).dot(b))
        p = (p + np.linalg.pinv(A).dot(b).T).ravel()
```

Figure 10: LucasKanadeAffine Function Implementation

2.2 Using the function you have developed for dominant motion estimation, write a function with the following signature SubtractDominantMotion(image1, image2) where image1 and image2 form the input image pair, and the return value mask is a binary image of the same size that dictates which pixels are considered to be corresponding to moving objects.

My implementation for the SubtractDominantMotion function can be found in Figure 11 below.

```python
import numpy as np
import LucasKanadeAffine
import scipy

def SubtractDominantMotion(image1, image2, threshold, num_iters, tolerance):
    """
    :param image1: Images at time t
    :param image2: Images at time t+1
    :param threshold: used for LucasKanadeAffine
    :param num_iters: used for LucasKanadeAffine
    :param tolerance: binary threshold of intensity difference when computing the mask
    :return: mask: [nxm]
    """

    # put your implementation here
    mask = np.ones(image1.shape, dtype=bool)

    #Find Matrix M and Warp Image
    matrix_M = LucasKanadeAffine.LucasKanadeAffine(image1, image2, threshold, num_iters, tolerance)
    warped_image = scipy.ndimage.affine_transform(image1, -matrix_M, offset=0.0, output_shape=None)

    #Generate Image Mask
    mask = (np.abs(warped_image - image2) > tolerance)
    mask = scipy.ndimage.morphology.binary_erosion(mask)
    mask = scipy.ndimage.morphology.binary_dilation(mask)

    return mask
```

Figure 11: SubtractDominantMotion Function Implementation

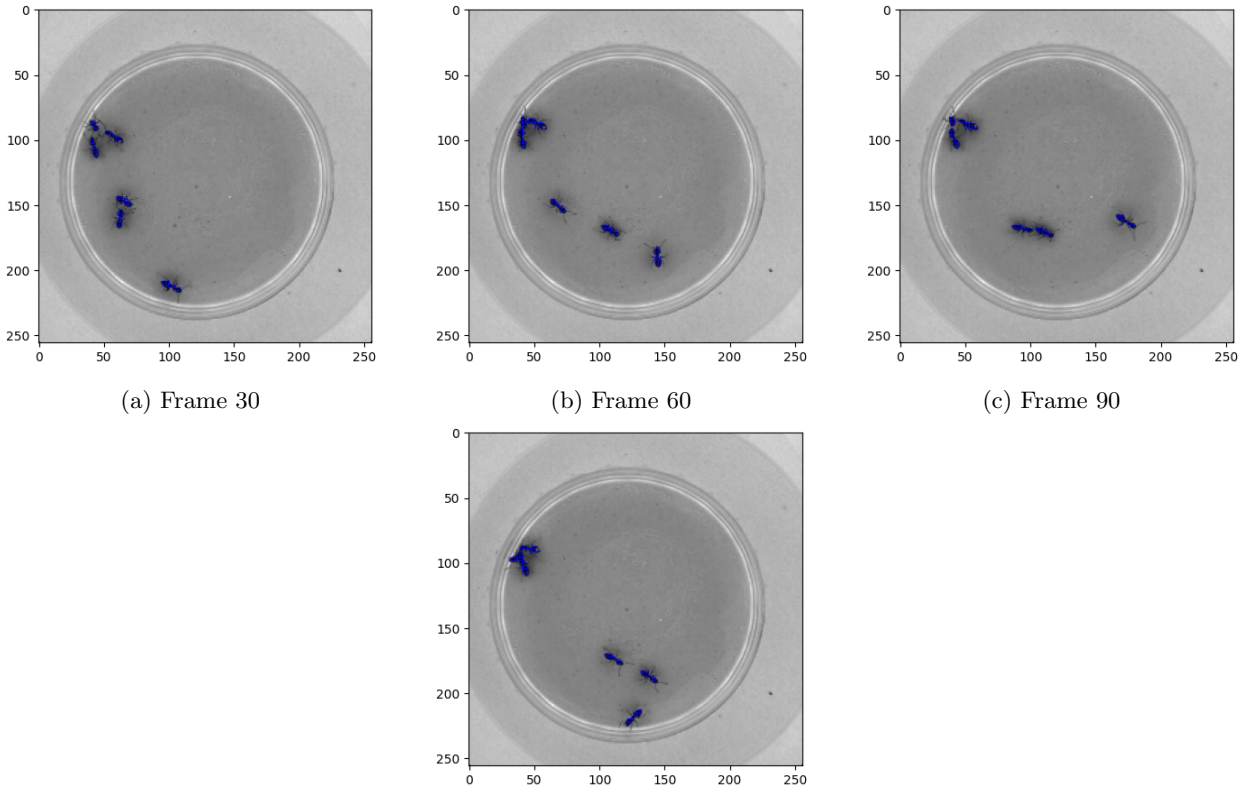
2.3 Write two scripts testAntSequence.py and testAerialSequence.py that load the image sequence from antseq.npy and aerialseq.npy and run the motion detection routine you have developed to detect the moving objects. Try to implement testAntSequence.py first as it involves little camera movement and can help you debug your mask generation procedure.

My implementation for the testAntSequence function can be found in Figure 12 below.

```python
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import SubtractDominantMotion

# write your script here, we recommend the above libraries for making your animation

parser = argparse.ArgumentParser()
parser.add_argument('--num_iters', type=int, default=1e3, help='number of iterations of Lucas-Kanade')
parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-Kanade for terminating optimization')
parser.add_argument('--tolerance', type=float, default=0.2, help='binary threshold of intensity difference when computing the mask')
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold
tolerance = args.tolerance

seq = np.load('../data/antseq.npy')
_, _, frame = seq.shape
for i in range(frame - 1):
    print(f'Frame ',(i+1), 'of ', (frame))

    #Get Frames for Analysis
    template_frame = seq[:, :, i]
    curr_frame = seq[:, :, i + 1]

    #Get Image for Mask
    mask = SubtractDominantMotion.SubtractDominantMotion(template_frame, curr_frame, threshold, num_iters, tolerance)
    positions_x = []
    positions_y = []
    for j in range(mask.shape[0]):
        for k in range(mask.shape[1]):
            if mask[j, k] == 0:
                positions_y.append(j)
                positions_x.append(k)

    #Create Boxed Image
    if i == 29 or i == 59 or i == 89 or i == 119:
        print("True")
        plt.figure()
        plt.imshow(curr_frame, cmap='gray')
        plt.plot(positions_x, positions_y, ',', markerfacecolor='blue')
        plt.savefig('../results/AntFrame_'+str(i + 1)+'.png', bbox_inches='tight')
```

Figure 12: testAntSequence Function Implementation

The results of the Lucas-Kanade Affine tracker on the ant sequence can be found in Figure 13 below. Here we can see that the blue plots follows the the ants around fairly well even though it may be a little difficult to see. The affine transformation visibly helps the Lucas Kanade method follow the optical flow of the ants much better.



(a) Frame 30

(b) Frame 60

(c) Frame 90

(d) Frame 120

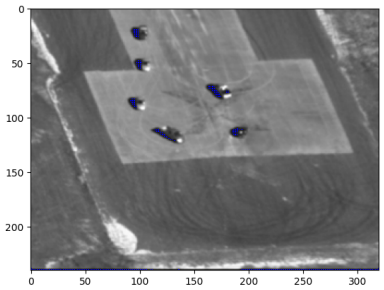
Figure 13: Lucas-Kanade Affine Tracking on Ants

My implementation for the testAerialSequence function can be found in Figure 14 below.

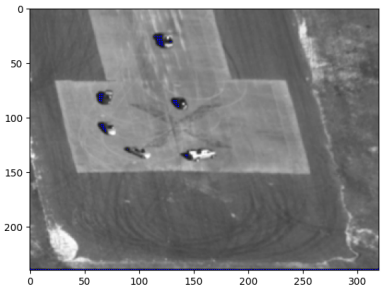
```python
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import SubtractDominantMotion

# write your script here, we recommend the above libraries for making your animation

parser = argparse.ArgumentParser()
parser.add_argument('--num_iters', type=int, default=1e3, help='number of iterations of Lucas-Kanade')
parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-Kanade for terminating optimization')
parser.add_argument('--tolerance', type=float, default=0.2, help='binary threshold of intensity difference when computing the mask')
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold
tolerance = args.tolerance

seq = np.load('../data/aerialseq.npy')
_, _, frame = seq.shape
for i in range(frame - 1):
    print(f'Frame ',(i+1), 'of ', (frame))

    #Get Frames for Analysis
    template_frame = seq[:, :, i]
    curr_frame = seq[:, :, i + 1]

    #Get Image for Mask
    mask = SubtractDominantMotion.SubtractDominantMotion(template_frame, curr_frame, threshold, num_iters, tolerance)
    positions_x = []
    positions_y = []
    for j in range(mask.shape[0]):
        for k in range(mask.shape[1]):
            if mask[j, k] == 0:
                positions_y.append(j)
                positions_x.append(k)

    #Create Boxed Image
    if i == 29 or i == 59 or i == 89 or i == 119:
        print("True")
        plt.figure()
        plt.imshow(curr_frame, cmap='gray')
        plt.plot(positions_x, positions_y, ',', markerfacecolor='blue')
        plt.savefig('../results/AerialFrame_'+str(i + 1)+'.png', bbox_inches='tight')
```

Figure 14: testAerialSequence Function Implementation

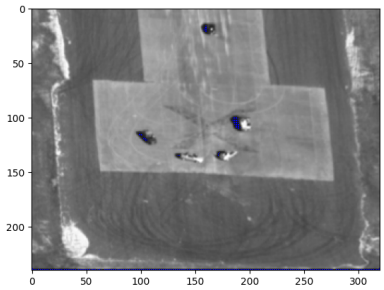
The results of the Lucas-Kanade Affine tracker on the aerial car sequence can be found in Figure 15 below. The blue outline in this image is much more difficult to see in these images but if we zoom in, we see that the blue tracks the car shadow to identify the locations by using the optical flow of the darker shadow regions instead.



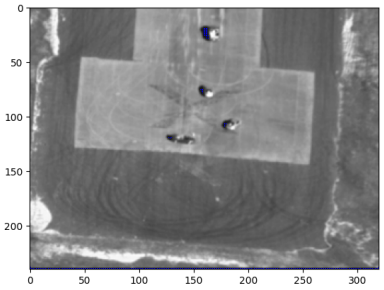
(a) Frame 30



(b) Frame 60



(c) Frame 90



(d) Frame 120

Figure 15: Lucas-Kanade Affine Tracking on Aerial Car Motion

3 Efficient Tracking

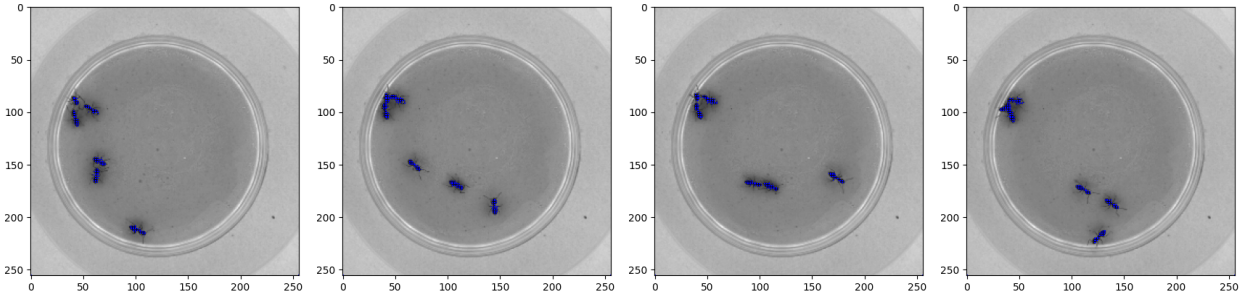
3.1 Reimplement the function LucasKanadeAffine(It,It1) as InverseCompositionAffine(It,It1) using the inverse compositional method. Report the results of the query frames (see Q2.3) for both ant and aerial sequences, as well as the runtime performance gain you observe using the inverse composition method. In your own words, please describe why the inverse compositional approach is more computationally efficient than the classical approach.

My implementation for the InverseCompositionAffine function can be found in Figure 16 below.

```python
import numpy as np
from scipy.interpolate import RectBivariateSpline
import scipy

def InverseCompositionAffine(It, It1, threshold, num_iters):
    """
    :param It: template image
    :param It1: Current image
    :param threshold: if the length of dp is smaller than the threshold, terminate the optimization
    :param num_iters: number of iterations of the optimization
    :return: M: the Affine warp matrix [2x3 numpy array]
    """

    # put your implementation here
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
    Hessian = np.zeros((6, 6))
    p = np.zeros(6)
    error = 1
    count = 1

    #Get Gradients and Calculate Hessian
    x_grad, y_grad = np.gradient(It)
    for i in range(It.shape[0]):
        for j in range(It.shape[1]):
            A_matrix = np.dot(np.array([x_grad[i, j], y_grad[i, j]]), np.array([[j, 0, i, 0, 1, 0], [0, j, 0, i, 0, 1]]))[np.newaxis, :]
            Hessian = Hessian + np.dot(A_matrix.T, A_matrix)
    print("Hessian Found")

    while error > threshold and count < num_iters:
        print("Iteration: ", count)
        count += 1
        b_matrix = np.zeros((6, 1))
        warped_curr_frame = scipy.ndimage.affine_transform(It1, M)

        #Compute gradient of b
        for i in range(It.shape[0]):
            for j in range(It.shape[1]):
                A_matrix = np.dot(np.array([x_grad[i, j], y_grad[i, j]]), np.array([[j, 0, i, 0, 1, 0], [0, j, 0, i, 0, 1]]))[np.newaxis, :]
                Hessian = Hessian + np.dot(A_matrix.T, A_matrix)
                b_matrix = b_matrix + (np.transpose(A_matrix) * (warped_curr_frame - It))

        #Pseudoinverse Solve and Update
        magnitude = np.dot(np.linalg.pinv(Hessian), (b_matrix))

        #Update M matrix using derivative
        mag_derivative = [
            [1.0 + magnitude[0][0], magnitude[2][0], magnitude[4][0]],
            [magnitude[1][0], 1.0 + magnitude[3][0], magnitude[5][0]],
            [0, 0, 1]]
        M = np.dot(np.concatenate((M, np.array([[0, 0, 1]])), axis=0), np.linalg.pinv(mag_derivative))
        M = M[0:2, :]

        #New p Update
        p = (p + magnitude.T).ravel()
        error = np.linalg.norm(magnitude)

    return M
```

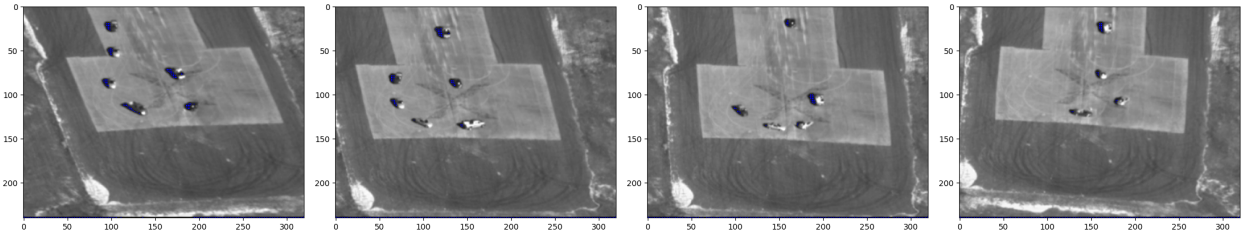
Figure 16: InverseCompositionAffine Function Implementation

The results of the Lucas-Kanade Inverse tracker can be found in Figures 17 and 18 below. Here we can see that the blue follows the car and ant just as well as the affine method we saw before. The difference between the two is much more noticeable when considering the runtime between the two. For the original Lucas Kanade Affine method, the runtime for the ant and car were found to be 324 seconds (5.4 minutes) and 371 seconds (6.18 minutes) correspondingly. On the other hand, the inverse Lucas Kanade method was able to achieve the same results with a much faster runtime, 119 seconds (1.983 minutes) for ant and 158 seconds (2.63 minutes) for aerial car. This disparity is most likely due to the fact that the inverse method allows us to precompute the large terms like the hessian and gradients. Using this, we can track the template to the current frame and then use that to find the correct warp from the current frame to the template with inverted warping.



(a) Frame 30 (b) Frame 60 (c) Frame 90 (d) Frame 120

Figure 17: Lucas-Kanade Inverse Tracking on Ant Motion



(a) Frame 30 (b) Frame 60 (c) Frame 90 (d) Frame 120

Figure 18: Lucas-Kanade Inverse Tracking on Aerial Car Motion