16-720A — Spring 2021 — Homework 3

Jen-Hung Ho
jenhungh@andrew.cmu.edu

March 27, 2021

Question 1

Q1.1

We know that the pure translation warp function is

$$\mathcal{W}(\mathbf{x};\ \mathbf{p}) = \mathbf{x} + \mathbf{p} \tag{1}$$

$$\mathcal{W}(\mathbf{x};\ \mathbf{p}) = \begin{bmatrix} \mathcal{W}_x(x;\ p_x) \\ \mathcal{W}_x(y;\ p_y) \end{bmatrix} = \begin{bmatrix} x + p_x \\ y + p_y \end{bmatrix} \tag{2}$$

Take partial derivative with respect to $\mathbf{p}$, we can get

$$\frac{\partial \mathcal{W}(\mathbf{x};\ \mathbf{p})}{\partial \mathbf{p}^T} = \begin{bmatrix} \frac{\partial \mathcal{W}_x(x;\, p_x)}{\partial p_x} & \frac{\partial \mathcal{W}_x(x;\, p_x)}{\partial p_y} \\ \frac{\partial \mathcal{W}_y(y;\, p_y))}{\partial p_x} & \frac{\partial \mathcal{W}_y(y;\, p_y)}{\partial p_y} \end{bmatrix} = \begin{bmatrix} \frac{\partial (x+p_x)}{\partial p_x} & \frac{\partial (x+p_x)}{\partial p_y} \\ \frac{\partial (y+p_y)}{\partial p_x} & \frac{\partial (y+p_y)}{\partial p_y} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{3}$$

The Least Square problem we try to solve

$$\underset{\Delta\mathbf{p}}{\operatorname{argmin}} \sum_{x \in \mathbb{N}} ||\mathcal{I}_{t+1}(\mathbf{x} + \mathbf{p}) - \mathcal{I}_t(\mathbf{x})||_2^2 \tag{4}$$

$$= \underset{\Delta\mathbf{p}}{\operatorname{argmin}} \sum_{x \in \mathbb{N}} \left|\left| \mathcal{I}_{t+1}(\mathbf{x}') + \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial \mathcal{W}(\mathbf{x};\ \mathbf{p})}{\partial \mathbf{p}^T} \Delta\mathbf{p} - \mathcal{I}_t(\mathbf{x}) \right|\right|_2^2 \tag{5}$$

$$= \underset{\Delta\mathbf{p}}{\operatorname{argmin}} \sum_{x \in \mathbb{N}} \left|\left| \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial \mathcal{W}(\mathbf{x};\ \mathbf{p})}{\partial \mathbf{p}^T} \Delta\mathbf{p} - (\mathcal{I}_t(\mathbf{x}) - \mathcal{I}_{t+1}(\mathbf{x}')) \right|\right|_2^2 \tag{6}$$

$$= \underset{\Delta\mathbf{p}}{\operatorname{argmin}} ||\mathbf{A}\Delta\mathbf{p} - \mathbf{b}||_2^2 \tag{7}$$

From equation 6 and 7, we know that

$$\mathbf{A} = \sum_{x \in \mathbb{N}} \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial \mathcal{W}(\mathbf{x};\ \mathbf{p})}{\partial \mathbf{p}^T} = \begin{bmatrix} \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}'_1)}{\partial \mathbf{x}'^T_1} & \cdots & \mathbf{0}^T \\ \vdots & \ddots & \vdots \\ \mathbf{0}^T & \cdots & \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}'_N)}{\partial \mathbf{x}'^T_N} \end{bmatrix} \begin{bmatrix} \frac{\partial \mathcal{W}(\mathbf{x}_1;\, \mathbf{p})}{\partial \mathbf{p}^T} \\ \vdots \\ \frac{\partial \mathcal{W}(\mathbf{x}_N;\, \mathbf{p})}{\partial \mathbf{p}^T} \end{bmatrix} \tag{8}$$

$$\mathbf{b} = \sum_{x \in \mathbb{N}} \mathcal{I}_{t+1}(\mathbf{x}) - \mathcal{I}_{t+1}(\mathbf{x}'_1) = \begin{bmatrix} \mathcal{I}_t(\mathbf{x}_1) - \mathcal{I}_{t+1}(\mathbf{x}'_1) \\ \vdots \\ \mathcal{I}_t(\mathbf{x}_N) - \mathcal{I}_{t+1}(\mathbf{x}'_N) \end{bmatrix} \tag{9}$$

We can solve the Least Square problem by using the pseudo inverse.

$$\Delta \mathbf{p} = \sum_{x \in \mathbb{N}} (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T b \tag{10}$$

From equation 10, we know that $\mathbf{A}^T\mathbf{A}$ must be invertible so that an unique solution to $\Delta\mathbf{p}$ can be found.

Q1.3

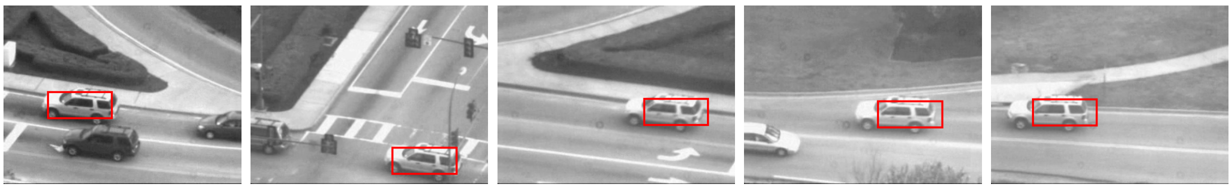

Figure 1: Car Sequence: Lucas-Kanade Tracking with One Single Template

Figure 2: Girl Sequence: Lucas-Kanade Tracking with One Single Template

Q1.4

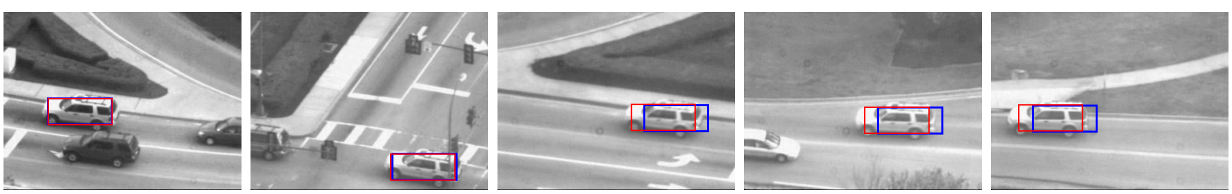

Figure 3: Car Sequence: Lucas-Kanade Tracking with Template Correction (blue: without correction, red: with correction)

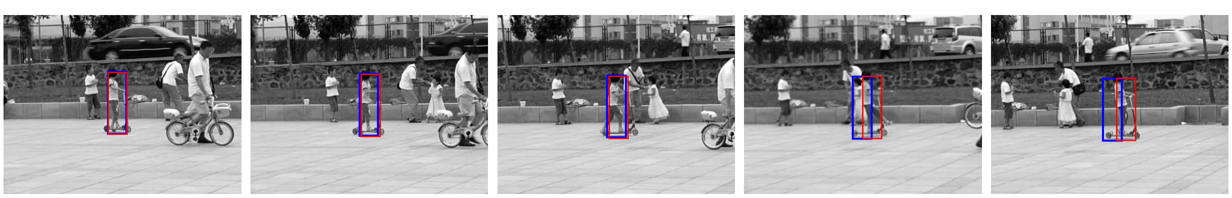

Figure 4: Girl Sequence: Lucas-Kanade Tracking with Template Correction (blue: without correction, red: with correction)

Question 2

Q2.3

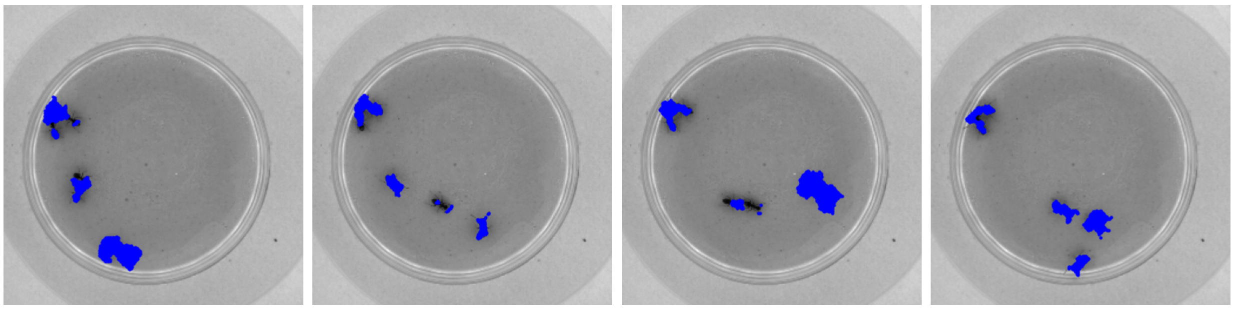

Figure 5: Ant Sequence: Lucas-Kanade Tracking with Motion Detection

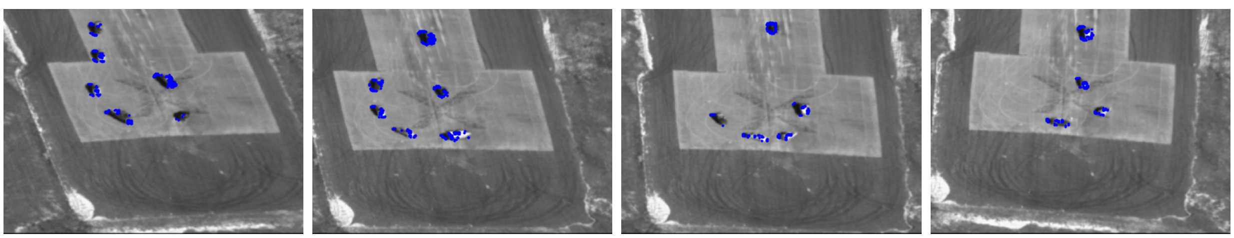

Figure 6: Aerial Sequence: Lucas-Kanade Tracking with Motion Detection

Question 3

Q3.2

The reason that inverse compositional approach is more conputationally efficient than the classical approach is beacuse of the benefits to precompute matrix $\mathbf{A}$ and the Hessian matrix ($\mathbf{H} = \mathbf{A}^T\mathbf{A}$). The calculation of the gradients and the jacobians are all moved out from the iteration loop, which saves a lot of time and thus increases the computational efficiency.

Code Appendix

0.1 LucasKanade.py

```
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]
    """

    # Set up the initial parameter guess and the corners of the rectangle
    p = p0
    # Print out rectangle to check
    x1, y1, x2, y2 = rect
    print(f"rect = {x1}, {y1}, {x2}, {y2}")

    # Compute the spline for sub-pixel interpolation
    h, w = It.shape
    h1, w1 = It1.shape
    spline_temp = RectBivariateSpline(np.arange(h), np.arange(w), It)
    spline_current = RectBivariateSpline(np.arange(h1), np.arange(w1), It1)

    # Compute I_temp
    xx, yy = np.meshgrid(np.arange(x1, x2 + 0.1), np.arange(y1, y2 + 0.1))
    total_size = xx.shape[0] * xx.shape[1]
    I_temp = spline_temp.ev(yy, xx)

    # Iterations
    for iters in range(int(num_iters)):
        # Compute I_current
        I_current = spline_current.ev(yy + p[1], xx + p[0])

        # Calculate the gradient: dI/dx and dI/dy
        dIdx = spline_current.ev(yy + p[1], xx + p[0], dy = 1).flatten()
        dIdy = spline_current.ev(yy + p[1], xx + p[0], dx = 1).flatten()

        # Calculate the jacobian: dW/dp
        dWdp = np.eye(2)

```

```
        # Compute A and b
        # Compute A = dI/dX * dW/dp
        A = np.zeros((total_size, 2))
        for index in range(total_size):
            A[index, 0] = dIdx[index] * dWdp[0, 0] + dIdy[index] * dWdp[1, 0]
            A[index, 1] = dIdx[index] * dWdp[0, 1] + dIdy[index] * dWdp[1, 1]
        # Compute b
        b = (I_temp - I_current).reshape(-1, 1)

        # Solve the Least Square problem using Pseudo Inverse
        dp = np.linalg.inv(A.T @ A) @ A.T @ b

        # Update p
        p = np.asarray([p[0] + dp[0, 0], p[1] + dp[1, 0]])

        # Check the norm of dp
        if (np.linalg.norm(dp, ord = 2) < threshold):
            break

    return p
```

0.2 LucasKanadeAffine.py

```
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
    """

    # Set up the initial affine matrix M
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])

    # Compute the spline for interpolation
    h, w = It.shape
    h1, w1 = It1.shape
    spline_temp = RectBivariateSpline(np.arange(h), np.arange(w), It)
    spline_current = RectBivariateSpline(np.arange(h1), np.arange(w1), It1)

    # Compute the meshgrid of the template
    XX, YY = np.meshgrid(np.arange(w), np.arange(h))

    # Iterations
    for iters in range(int(num_iters)):
        # Warp the image: compute X_bar
        xx, yy = XX, YY
        xx_bar = M[0, 0] * xx + M[0, 1] * yy + M[0, 2] * np.ones_like(xx)
        yy_bar = M[1, 0] * xx + M[1, 1] * yy + M[1, 2] * np.ones_like(xx)

        # Check the valid area
        valid_index = (xx_bar >= 0) & (xx_bar[0] < w) & (yy_bar >= 0) &
                                                                  (yy_bar < h)
        xx, yy = xx[valid_index], yy[valid_index]
        xx_bar, yy_bar = xx_bar[valid_index], yy_bar[valid_index]

        # Compute I_temp and I_current
        I_temp = spline_temp.ev(yy, xx).flatten()
        I_current = spline_current.ev(yy_bar, xx_bar).flatten()

        # Calculate the gradient: dI/dx and dI/dy
        dIdx = spline_current.ev(yy_bar, xx_bar, dy = 1).flatten()
        dIdy = spline_current.ev(yy_bar, xx_bar, dx = 1).flatten()
```

```python

        # Calculate the jacobian: dW/dp
        dWdp1 = xx.flatten()
        dWdp2 = yy.flatten()
        dWdp3 = np.ones_like(xx.flatten())
        dWdp4 = xx.flatten()
        dWdp5 = yy.flatten()
        dWdp6 = np.ones_like(xx.flatten())

        # Compute A and b
        # Compute A = dI/dX * dW/dp
        A = np.zeros((xx.flatten().shape[0], 6))
        A[:, 0] = dIdx * dWdp1
        A[:, 1] = dIdx * dWdp2
        A[:, 2] = dIdx * dWdp3
        A[:, 3] = dIdy * dWdp4
        A[:, 4] = dIdy * dWdp5
        A[:, 5] = dIdy * dWdp6
        # Compute b
        b = (I_temp - I_current).reshape(-1, 1)

        # Solve the Least Square problem using Pseudo Inverse
        dp = np.linalg.inv(A.T @ A) @ A.T @ b

        # Update M
        M[0, :] += dp[0:3, 0]
        M[1, :] += dp[3:6, 0]

        # Check the norm of dp
        if (np.linalg.norm(dp, ord = 2) < threshold):
            break

    return M
```

0.3 SubtractDominantMotion.py

```
import numpy as np
from scipy.interpolate import RectBivariateSpline
from LucasKanadeAffine import LucasKanadeAffine
from InverseCompositionAffine import InverseCompositionAffine

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]
    """

    # Initialize the binary mask
    mask = np.zeros(image1.shape, dtype=bool)

    # Compute the spline for interpolation
    h1, w1 = image1.shape
    h2, w2 = image2.shape
    spline1 = RectBivariateSpline(np.arange(h1), np.arange(w1), image1)
    spline2 = RectBivariateSpline(np.arange(h2), np.arange(w2), image2)

    # Affine Warp
    # Compute the meshgrid of image1
    xx, yy = np.meshgrid(np.arange(w1), np.arange(h1))
    # Compute transformation matrix M : LucasKanadeAffine or
    # InverseCompositionAffine
    M = LucasKanadeAffine(image1, image2, threshold, num_iters)
    # M = InverseCompositionAffine(image1, image2, threshold, num_iters)
    print(f"M = {M}")
    # Warp image1
    xx_bar = M[0, 0] * xx + M[0, 1] * yy + M[0, 2] * np.ones_like(xx)
    yy_bar = M[1, 0] * xx + M[1, 1] * yy + M[1, 2] * np.ones_like(xx)

    # Check the valid area
    valid_index = (xx_bar >= 0) & (xx_bar < w1) & (yy_bar >= 0) & (yy_bar < h1)
    valid_map = valid_index.astype(int)
    xx, yy = xx * valid_map, yy * valid_map
    xx_bar, yy_bar = xx_bar * valid_map, yy_bar * valid_map

    # Compute Image1_warp and Image2
    Image1_warp = spline1.ev(yy, xx)
    Image2 = spline2.ev(yy_bar, xx_bar)
```

```

    # Compute the subtraction
    diff = abs(Image2 - Image1_warp)
    mask[diff > tolerance] = 1

    return mask
```

0.4 InverseCompositionAffine.py

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

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]
    """

    # Set up the initial M
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])

    # Compute the spline for interpolation
    h, w = It.shape
    h1, w1 = It1.shape
    spline_temp = RectBivariateSpline(np.arange(h), np.arange(w), It)
    spline_current = RectBivariateSpline(np.arange(h1), np.arange(w1), It1)

    # Compute the meshgrid of the template
    xx, yy = np.meshgrid(np.arange(w), np.arange(h))

    # Compute A
    # Calculate the gradient: dI/dx and dI/dy
    dIdx = spline_temp.ev(yy, xx, dy = 1).flatten()
    dIdy = spline_temp.ev(yy, xx, dx = 1).flatten()
    # Calculate the jacobian: dW/dp
    dWdp1 = xx.flatten()
    dWdp2 = yy.flatten()
    dWdp3 = np.ones_like(xx.flatten())
    dWdp4 = xx.flatten()
    dWdp5 = yy.flatten()
    dWdp6 = np.ones_like(xx.flatten())
    # A = dI/dX * dW/dp
    A = np.zeros((xx.flatten().shape[0], 6))
    A[:, 0] = dIdx * dWdp1
    A[:, 1] = dIdx * dWdp2
    A[:, 2] = dIdx * dWdp3
    A[:, 3] = dIdy * dWdp4
    A[:, 4] = dIdy * dWdp5
    A[:, 5] = dIdy * dWdp6

    # Compute the Hessian = A.T @ A
```

```
    Hessian = A.T @ A

    # Iterations
    for iters in range(int(num_iters)):
        # Warp the image: compute X_bar
        xx_bar = M[0, 0] * xx + M[0, 1] * yy + M[0, 2] * np.ones_like(xx)
        yy_bar = M[1, 0] * xx + M[1, 1] * yy + M[1, 2] * np.ones_like(xx)

        # Check the valid area and set the error to 0 if invalid
        valid_index = (xx_bar >= 0) & (xx_bar[0] < w) & (yy_bar >= 0) &
                      (yy_bar < h)
        valid_map = valid_index.astype(int)
        xx, yy = xx * valid_map, yy * valid_map
        xx_bar, yy_bar = xx_bar * valid_map, yy_bar * valid_map

        # Compute I_temp and I_current
        I_temp = spline_temp.ev(yy, xx).flatten()
        I_current = spline_current.ev(yy_bar, xx_bar).flatten()

        # A and Hessian are precomputed
        # Compute b
        b = (I_current - I_temp).reshape(-1, 1)

        # Solve the Least Square problem using Pseudo Inverse
        # Compute dp and dM
        dp = np.linalg.inv(Hessian) @ A.T @ b
        dM = np.array([[1 + dp[0, 0], dp[1, 0], dp[2, 0]],
                       [dp[3, 0], 1 + dp[4, 0], dp[5, 0]],
                       [0, 0, 1]])

        # Update M
        M = np.concatenate((M, [[0, 0, 1]]), axis = 0)
        M = M @ np.linalg.inv(dM)
        M = M[0:2, :]

        # Check dp
        if (np.linalg.norm(dp, ord = 2) < threshold):
            break

    return M
```

0.5 testCarSequence.py

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

# write your script here, we recommend the above libraries
# Set up iterations and threshold
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

# Load the video frames and Set up the reported frames
seq = np.load("../data/carseq.npy")
reported_frames = [0, 99, 199, 299, 399]

# Set up the initial rectangle and the rectangle history
rect = [59, 116, 145, 151]
rect_history = []
rect_history.append(rect)

# Apply Lucas-Kanade tracker to every frames
num_frames = seq.shape[2]
for index in range(num_frames-1):
    # Print out the frame index
    print(f"frame = {index+1}")

    # Load template and current frames
    It = seq[:, :, index]
    It1 = seq[:, :, index+1]

    # Apply Lucas-Kanade tracker
    p = LucasKanade(It, It1, rect, threshold, num_iters)

    # Update and Append rectangle
    rect = [rect[0] + p[0], rect[1] + p[1], rect[2] + p[0], rect[3] + p[1]]
    rect_history.append(rect)

    # Plot the reported frames
    if index in reported_frames:
```

```python
        # Plot frames
        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        plt.imshow(It, cmap = 'gray')
        plt.axis('off')

        # Plot the rectangles
        x1, y1, x2, y2 = rect_history[index]
        width, height = (x2 - x1), (y2 - y1)
        rectangle = patches.Rectangle((x1, y1), width, height, edgecolor = 'r',
                                      facecolor = 'none', linewidth = 3)
        ax.add_patch(rectangle)

        # Save the figures
        plots_dir = '../plots/'
        if not os.path.exists(plots_dir):
            os.makedirs(plots_dir)
        fig.savefig('../plots/Car_f' + str(index + 1) + '.png',
                    bbox_inches = 'tight', pad_inches = 0)

# Save the rectangle history
results_dir = '../results/'
if not os.path.exists(results_dir):
    os.makedirs(results_dir)
np.save('../results/carseqrects.npy', np.asarray(rect_history))
```

0.6 testCarSequenceWithTemplateCorrection.py

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

# write your script here, we recommend the above libraries
# Set up iterations and thresholds
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

# Load the video frames and Set up the reported frames
seq = np.load("../data/carseq.npy")
reported_frames = [0, 99, 199, 299, 399]

# Set up initial guess, initial template, and first frame
# The first frame is used to correct the drift
p0 = np.zeros(2)
template = seq[:, :, 0]
template_first = seq[:, :, 0]

# Set up the initial rectangle and the rectangle history
rect_initial = [59, 116, 145, 151]
rect = rect_initial
rect_history = []
rect_history.append(rect)

# Apply Lucas-Kanade tracker with Template Correction to every frames
num_frames = seq.shape[2]
for index in range(num_frames-1):
    # Print out the frame index
    print(f"frame = {index+1}")

    # Load the current frames
    It1 = seq[:, :, index+1]

```

```
    # Apply Lucas-Kanade tracker with Template Correction
    # Compute p
    p = LucasKanade(template, It1, rect, threshold, num_iters, p0)
    # Compute pn
    pn_x = (rect[0] + p[0]) - rect_initial[0]
    pn_y = (rect[1] + p[1]) - rect_initial[1]
    pn = np.array([pn_x, pn_y])
    # Compute pn_star
    pn_star = LucasKanade(template_first, It1, rect_initial, threshold,
                          num_iters, pn)

    # Template Correction : Update the rectangle
    # Naive Update
    if (np.linalg.norm(pn_star - pn, ord = 2) <= template_threshold):
        # Update the rectangle : drift correction
        p_current_x = pn_star[0] - (rect[0] - rect_initial[0])
        p_current_y = pn_star[1] - (rect[1] - rect_initial[1])
        rect = [rect[0] + p_current_x, rect[1] + p_current_y,
                rect[2] + p_current_x, rect[3] + p_current_y]
        # Update template
        template = seq[:, :, index+1]
        # Reset p0
        p0 = np.zeros(2)
    # No Update
    else:
        p0 = p

    # Append the rectangle
    rect_history.append(rect)

    # Plot the reported frames
    rect_without_correction = np.load('../results/carseqrects.npy')
    if index in reported_frames:
        # Plot frames
        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        plt.imshow(seq[:, :, index], cmap = 'gray')
        plt.axis('off')
        # Plot the rectangles without correction
        x1_wo, y1_wo, x2_wo, y2_wo = rect_without_correction[index, :]
        width_wo, height_wo = (x2_wo - x1_wo), (y2_wo - y1_wo)
        rectangle_wo = patches.Rectangle((x1_wo, y1_wo), width_wo, height_wo,
                       edgecolor = 'b', facecolor = 'None', linewidth = 3)
        ax.add_patch(rectangle_wo)
        # Plot the rectangles with correction
        x1, y1, x2, y2 = rect_history[index]
        rectangle = patches.Rectangle((x1, y1), (x2 - x1), (y2 - y1),
                    edgecolor = 'r', facecolor = 'None', linewidth = 2)
```

```
        ax.add_patch(rectangle)

        # Save the figures
        plots_dir = '../plots/'
        if not os.path.exists(plots_dir):
            os.makedirs(plots_dir)
        fig.savefig('../plots/Car-wcrt_f' + str(index+1) + '.png',
                    bbox_inches = 'tight', pad_inches = 0)

# Save the rectangle history
results_dir = '../results/'
if not os.path.exists(results_dir):
    os.makedirs(results_dir)
np.save('../results/carseqrects-wcrt.npy', np.asarray(rect_history))
```

0.7 testGirlSequence.py

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

# write your script here, we recommend the above libraries
# Set up iterations and threshold
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

# Load the video frames and Set up the reported frames
seq = np.load("../data/girlseq.npy")
reported_frames = [0, 19, 39, 59, 79]

# Set up the initial rectangle and the rectangle history
rect = [280, 152, 330, 318]
rect_history = []
rect_history.append(rect)

# Apply Lucas-Kanade tracker to every frames
num_frames = seq.shape[2]
for index in range(num_frames-1):
    # Print out the frame index
    print(f"frame = {index+1}")

    # Load template and current frames
    It = seq[:, :, index]
    It1 = seq[:, :, index+1]

    # Apply Lucas-Kanade tracker
    p = LucasKanade(It, It1, rect, threshold, num_iters)

    # Update and Append the rectangles
    rect = [rect[0] + p[0], rect[1] + p[1], rect[2] + p[0], rect[3] + p[1]]
    rect_history.append(rect)

    # Plot the reported frames
    if index in reported_frames:
```

```
        # Plot the frames
        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        plt.imshow(It, cmap = 'gray')
        plt.axis('off')

        # Plot the rectangles
        x1, y1, x2, y2 = rect_history[index]
        width, height = (x2 - x1), (y2 - y1)
        rectangle = patches.Rectangle((x1, y1), width, height, edgecolor = 'r',
                        facecolor = 'None', linewidth = 3)
        ax.add_patch(rectangle)

        # Save the figures
        plots_dir = '../plots/'
        if not os.path.exists(plots_dir):
            os.makedirs(plots_dir)
        fig.savefig('../plots/Girl_f' + str(index+1) + '.png',
                        bbox_inches = 'tight', pad_inches = 0)

# Save the rectangle history
results_dir = '../results/'
if not os.path.exists(results_dir):
    os.makedirs(results_dir)
np.save('../results/girlseqrects.npy', np.asarray(rect_history))
```

0.8 testGirlSequenceWithTemplateCorrection.py

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

# write your script here, we recommend the above libraries
# Set up iterations and thresholds
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

# Load the video frames and Set up the reported frames
seq = np.load("../data/girlseq.npy")
reported_frames = [0, 19, 39, 59, 79]

# Set up initial guess, initial template, and first frame
# The first frame is used to correct the drift
p0 = np.zeros(2)
template = seq[:, :, 0]
template_first = seq[:, :, 0]

# Set up the initial rectangle and the rectangle history
rect_initial = [280, 152, 330, 318]
rect = rect_initial
rect_history = []
rect_history.append(rect)

# Apply Lucas-Kanade tracker with Template Correction to every frames
num_frames = seq.shape[2]
for index in range(num_frames-1):
    # Print out the frame index
    print(f"frame = {index+1}")

    # Load the current frames
    It1 = seq[:, :, index+1]

```

```
    # Apply Lucas-Kanade tracker with Template Correction
    # Compute p
    p = LucasKanade(template, It1, rect, threshold, num_iters, p0)
    # Compute pn
    pn_x = (rect[0] + p[0]) - rect_initial[0]
    pn_y = (rect[1] + p[1]) - rect_initial[1]
    pn = np.array([pn_x, pn_y])
    # Compute pn_star
    pn_star = LucasKanade(template_first, It1, rect_initial, threshold,
                          num_iters, pn)

    # Template Correction : Update the rectangle
    # Naive Update
    if (np.linalg.norm(pn_star - pn, ord = 2) <= template_threshold):
        # Update the rectangle : drift correction
        p_current_x = pn_star[0] - (rect[0] - rect_initial[0])
        p_current_y = pn_star[1] - (rect[1] - rect_initial[1])
        rect = [rect[0] + p_current_x, rect[1] + p_current_y,
                rect[2] + p_current_x, rect[3] + p_current_y]
        # Update template
        template = seq[:, :, index+1]
        # Reset p0
        p0 = np.zeros(2)
    # No Update
    else:
        p0 = p

    # Append the rectangle
    rect_history.append(rect)

    # Plot the reported frames
    rect_without_correction = np.load('../results/girlseqrects.npy')
    if index in reported_frames:
        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        # Plot frames
        plt.imshow(seq[:, :, index], cmap = 'gray')
        plt.axis('off')
        # Plot the rectangles without correction
        x1_wo, y1_wo, x2_wo, y2_wo = rect_without_correction[index, :]
        rectangle_wo = patches.Rectangle((x1_wo, y1_wo), (x2_wo - x1_wo),
            (y2_wo - y1_wo), edgecolor = 'b', facecolor = 'None', linewidth = 3)
        ax.add_patch(rectangle_wo)
        # Plot the rectangles with correction
        x1, y1, x2, y2 = rect
        rectangle = patches.Rectangle((x1, y1), (x2 - x1), (y2 - y1),
                    edgecolor = 'r', facecolor = 'None', linewidth = 2)
        ax.add_patch(rectangle)
```

```python

        # Save the figures
        plots_dir = '../plots/'
        if not os.path.exists(plots_dir):
            os.makedirs(plots_dir)
        fig.savefig('../plots/Girl-wcrt_f' + str(index+1) + '.png',
                    bbox_inches = 'tight', pad_inches = 0)

# Save the rectangle history
results_dir = '../results/'
if not os.path.exists(results_dir):
    os.makedirs(results_dir)
np.save('../results/girlseqrects-wcrt.npy', np.asarray(rect_history))
```

0.9 testAntSequence.py

```
import argparse
import numpy as np
import scipy.ndimage
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import os
from SubtractDominantMotion import SubtractDominantMotion


# write your script here, we recommend the above libraries
# Set up iterations, threshold, and tolerance
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=0.03, 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

# Load the video frames and Set up the reported frames
seq = np.load('../data/antseq.npy')
reported_frames = [29, 59, 89, 119]

# Apply Subtract Dominant Motion to every frames
frames = seq.shape[2]
for index in range(frames-1):
    # Print out the frame index
    print(f"frame = {index+1}")

    # Load images
    image1 = seq[:, :, index]
    image2 = seq[:, :, index+1]

    # Apply Subtract Dominant Motion
    mask = SubtractDominantMotion(image1, image2, threshold, num_iters,
                                                                    tolerance)

    # Improvement: binary dilation and erosion
    mask = scipy.ndimage.binary_dilation(mask, iterations = 3)
    mask = scipy.ndimage.binary_erosion(mask, iterations = 4)

    # Extarct the moving objects
```

```python
    target = np.where(mask == 1)

    # Plot the reported frames
    if index in reported_frames:
        # Plot frames
        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        plt.imshow(image1, cmap = 'gray')
        plt.axis('off')

        # Plot the moving objects
        plt.plot(target[1], target[0], 'bo', markersize = 2)

        # Save the reported figures
        plots_dir = '../plots/'
        if not os.path.exists(plots_dir):
            os.makedirs(plots_dir)
        fig.savefig('../plots/Ant_f' + str(index+1) + '.png',
                    bbox_inches = 'tight', pad_inches = 0)
```

0.10 testAerialSequence.py

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

# write your script here, we recommend the above libraries
# Set up iterations, threshold, and tolerance
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

# Load the video frames and Set up the reported frames
seq = np.load('../data/aerialseq.npy')
reported_frames = [29, 59, 89, 119]

# Apply Subtract Dominant Motion to every frames
frames = seq.shape[2]
for index in range(frames-1):
    # Print out the frame index
    print(f"frame = {index+1}")

    # Load images
    image1 = seq[:, :, index]
    image2 = seq[:, :, index+1]

    # Apply Subtract Dominant Motion
    mask = SubtractDominantMotion(image1, image2, threshold, num_iters,
                                                                            tolerance)

    # Extarct the moving objects
    target = np.where(mask == 1)

    # Plot the reported frames
    if index in reported_frames:
        # Plot frames
        fig = plt.figure()
```

```
        ax = fig.add_subplot(1, 1, 1)
        plt.imshow(image1, cmap = 'gray')
        plt.axis('off')

        # Plot the moving objects
        plt.plot(target[1], target[0], 'bo', markersize = 2)

        # Save the reported figures
        plots_dir = '../plots/'
        if not os.path.exists(plots_dir):
            os.makedirs(plots_dir)
        fig.savefig('../plots/Aerial_f' + str(index+1) + '.png',
                    bbox_inches = 'tight', pad_inches = 0)
```