16-720 Computer Vision: Homework 1 (Fall 2022)

Lucas-Kanade Tracking

Haejoon Lee

Q1.1

- What is $\frac{\partial W(x;p)}{\partial p^T}$ (should a 2x2 matrix)?

$$W(x;p) = x + p \rightarrow \frac{\partial W(x;p)}{\partial p^T} = I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- What is A and b?

$$A = \begin{bmatrix} \nabla I_{t+1}(x_1 + p) \\ \nabla I_{t+1}(x_2 + p) \\ \dots \\ \nabla I_{t+1}(x_D + p) \end{bmatrix} \qquad b = \begin{bmatrix} I_t(x_1) - I_{t+1}(x_1 + p) \\ I_t(x_2) - I_{t+1}(x_2 + p) \\ \dots \\ I_t(x_D) - I_{t+1}(x_D + p) \end{bmatrix}$$

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

$$\sum$$

$$\rightarrow I_{t+1}(x' + \Delta p_x, y' + \Delta p_y) = I_{t+1}(x', y') + \frac{\partial I_{t+1}}{\partial x} \cdot \Delta p_x + \frac{\partial I_{t+1}}{\partial y} \Delta p_y$$

$$= I_{t+1}(\bar{x}') + \nabla I_{t+1}(\bar{x}) \Delta p$$

$$= I_{t+1}(\bar{x} + p) + \nabla I_{t+1}(\bar{x}) \Delta p$$

We can solve eq (2) problem by

$$p \leftarrow p + \underset{\Delta p}{\arg\min} \sum_{x \in N} \| I_{t+1}(\bar{x} + p + \Delta p) - I_t(x) \|_2^2$$

$$\approx p + \underset{\Delta p}{\text{argmin}} \sum_{x \in N} \| I_{t+1}(\bar{x} + p) + \nabla I_{t+1}(\bar{x} + p) \Delta p - I_t(x) \|_2^2$$

$$\hookrightarrow \underset{\Delta p}{\text{argmin}} \sum_{x \in N} \| \nabla I_{t+1}(\bar{x} + p) \Delta p + I_{t+1}(\bar{x} + p) - I_t(x) \|_2^2$$

$$= \underset{\Delta p}{\text{argmin}} \left\| \underbrace{\begin{bmatrix} \nabla I_{t+1}(\bar{x}_1 + p) \\ \nabla I_{t+1}(\bar{x}_2 + p) \\ \vdots \\ \nabla I_{t+1}(\bar{x}_D + p) \end{bmatrix}}_{A} \Delta p - \underbrace{\begin{bmatrix} I_t(\bar{x}_1) - I_{t+1}(\bar{x}_1 + p) \\ \vdots \\ I_t(\bar{x}_D) - I_{t+1}(\bar{x}_D + p) \end{bmatrix}}_{b} \right\|_2^2$$

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

$A^T A$ should be invertible

if) $\frac{d}{d\nabla p}\|A\nabla p - b\|^2 = 2 \cdot A^T(A\nabla p - b) = 0$ → solution for argmin

→ $A^TA\nabla p = A^Tb$

$\nabla p = (A^TA)^{-1}A^Tb$

=> A^TA should be invertible.

Q1.2

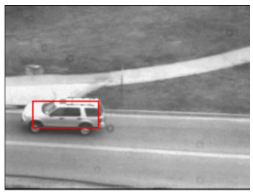
```python
def LucasKanade(It, It1, rect, threshold, num_iters, p0=np.zeros(2)):
    """
    :param It: template image
    :param It1: Current image
    :param rect: Current position of the car (top left, bot right coordinates)
    :param threshold: if the length of dp is smaller than the threshold, terminate 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
    # set up the threshold
    ################### TODO Implement Lucas Kanade ###################
    H_It, W_It = It.shape

    x1, y1, x2, y2 = rect[0], rect[1], rect[2], rect[3] #writen in image coordinates, not matrix
    H_rect, W_rect = y2 - y1, x2 - x1

    It1_y, It1_x = np.gradient(It1) #Gradient of current img

    x_axis_rect = np.linspace(x1, x2, int(W_rect))
    y_axis_rect = np.linspace(y1, y2, int(H_rect))
    X_grid_rect, Y_grid_rect = np.meshgrid(x_axis_rect, y_axis_rect)

    x = np.arange(0, W_It, 1)
    y = np.arange(0, H_It, 1)
    spline_It = RectBivariateSpline(y, x, It) #Spline interpolation over a rectangular mesh
    spline_It1 = RectBivariateSpline(y, x, It1)
    spline_It1_x = RectBivariateSpline(y, x, It1_x)
    spline_It1_y = RectBivariateSpline(y, x, It1_y)
    template_rect = spline_It.ev(Y_grid_rect, X_grid_rect) #Bring template_rect from It

    # Iterations for finding optimal dp
    p = p0
    dp = [[1000], [1000]] #Big dp for while loop
    counter = 1
    while np.square(dp).sum() > threshold and counter <= num_iters:

        # translate the rectangle
        x1_tr, y1_tr, x2_tr, y2_tr = x1+p[0], y1+p[1], x2+p[0], y2+p[1]
```

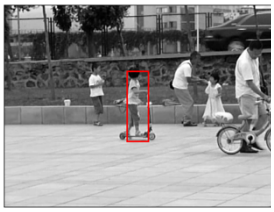
```
        x_axis_rect_tr = np.linspace(x1_tr, x2_tr, int(W_rect))
        y_axis_rect_tr = np.linspace(y1_tr, y2_tr, int(H_rect))
        X_grid_rect_tr, Y_grid_rect_tr = np.meshgrid(x_axis_rect_tr, y_axis_rect_tr)

        spline_It1_tr = spline_It1.ev(Y_grid_rect_tr, X_grid_rect_tr)

        #A, b
        It1_rect_tr_x = spline_It1_x.ev(Y_grid_rect_tr, X_grid_rect_tr)
        It1_rect_tr_y = spline_It1_y.ev(Y_grid_rect_tr, X_grid_rect_tr)
        A = np.vstack((It1_rect_tr_x.ravel(),It1_rect_tr_y.ravel())).T

        b = (template_rect - spline_It1_tr).reshape(-1, 1)
        b = b.reshape(-1,1) #Columnize

        #Solve argmin|Ax-b|^2 for finding dp
        dp = np.linalg.inv(A.T@A) @ (A.T) @ b

        #update parameters
        p[0] += dp[0,0]
        p[1] += dp[1,0]

        counter += 1

    return p
```

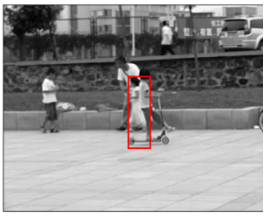
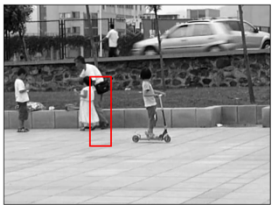
Q1.3







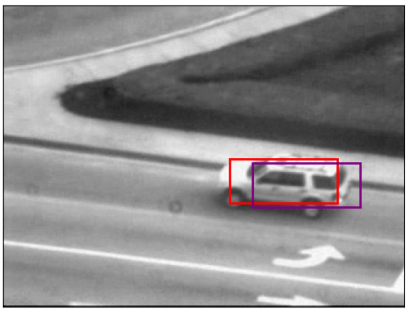
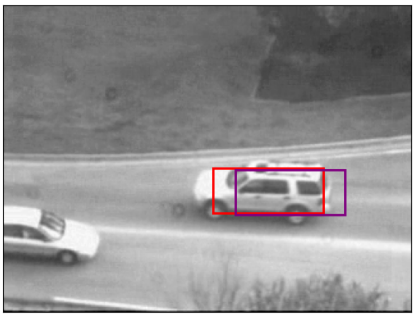
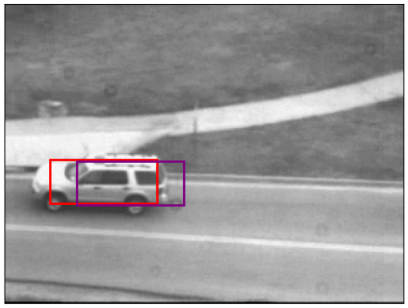
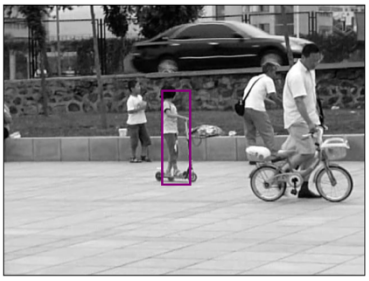
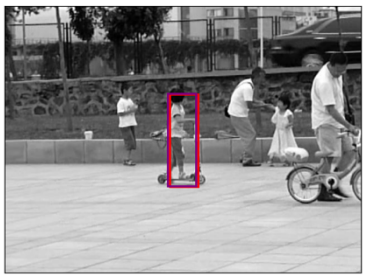



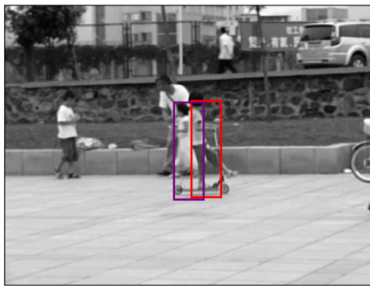
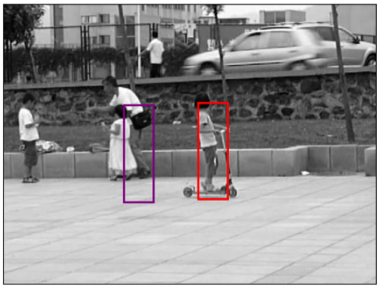



Q1.4

* Red: with template correction, Purple: original



```
def LucasKanade_difftemp(It, It1, rect_temp, rect_cur, threshold, num_iters, p0=np.zeros(2)):
    """
    :param It: template image
    :param It1: Current image
    :param rect_cur: Current position of the car (top left, bot right coordinates)
```

```
    :param rect_temp: Template rec position in the template image (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
    # set up the threshold
    ################### TODO Implement Lucas Kanade ###################
    H_It, W_It = It.shape
    H_It1, W_It1 = It1.shape

    x1, y1, x2, y2 = rect_cur[0], rect_cur[1], rect_cur[2], rect_cur[3] #writen in image
coordinates, not matrix
    xt1, yt1, xt2, yt2 = rect_temp[0], rect_temp[1], rect_temp[2], rect_temp[3] #writen in
image coordinates, not matrix

    H_rect, W_rect = y2 - y1, x2 - x1
    H_rect_temp, W_rect_temp = yt2 - yt1, xt2 - xt1

    It1_y, It1_x = np.gradient(It1) #Gradient of current img

#     x_axis_rect = np.linspace(x1, x2, int(W_rect))
#     y_axis_rect = np.linspace(y1, y2, int(H_rect))
#     X_grid_rect, Y_grid_rect = np.meshgrid(x_axis_rect, y_axis_rect)

    x_axis_rect_temp = np.linspace(xt1, xt2, int(round(W_rect_temp)))
    y_axis_rect_temp = np.linspace(yt1, yt2, int(round(H_rect_temp)))
    X_grid_rect_temp, Y_grid_rect_temp = np.meshgrid(x_axis_rect_temp, y_axis_rect_temp)



    xt = np.arange(0, W_It, 1)
    yt = np.arange(0, H_It, 1)
    x = np.arange(0, W_It1, 1)
    y = np.arange(0, H_It1, 1)
    spline_It = RectBivariateSpline(yt, xt, It) #Spline interpolation over a rectangular mesh
    spline_It1 = RectBivariateSpline(y, x, It1)
    spline_It1_x = RectBivariateSpline(y, x, It1_x)
    spline_It1_y = RectBivariateSpline(y, x, It1_y)
```

```
    template_rect = spline_It.ev(Y_grid_rect_temp, X_grid_rect_temp) #Bring template_rect
from It

    # Iterations for finding optimal dp
    p = p0
    dp = [[1000], [1000]] #Big dp for while loop
    counter = 1
    while np.square(dp).sum() > threshold and counter <= num_iters:

        # translate the rectangle
        x1_tr, y1_tr, x2_tr, y2_tr = x1+p[0], y1+p[1], x2+p[0], y2+p[1]

        x_axis_rect_tr = np.linspace(x1_tr, x2_tr, int(round(W_rect)))
        y_axis_rect_tr = np.linspace(y1_tr, y2_tr, int(round(H_rect)))
        X_grid_rect_tr, Y_grid_rect_tr = np.meshgrid(x_axis_rect_tr, y_axis_rect_tr)

        spline_It1_tr = spline_It1.ev(Y_grid_rect_tr, X_grid_rect_tr)

        #A, b
        It1_rect_tr_x = spline_It1_x.ev(Y_grid_rect_tr, X_grid_rect_tr)
        It1_rect_tr_y = spline_It1_y.ev(Y_grid_rect_tr, X_grid_rect_tr)
        A = np.vstack((It1_rect_tr_x.ravel(),It1_rect_tr_y.ravel())).T

        b = (template_rect - spline_It1_tr).reshape(-1, 1)
        b = b.reshape(-1,1) #Columnize

        #Solve argmin|Ax-b|^2 for finding dp
        dp = np.linalg.inv(A.T@A) @ (A.T) @ b

        #update parameters
        p[0] += dp[0,0]
        p[1] += dp[1,0]

        counter += 1

    return p
```

Q2.1

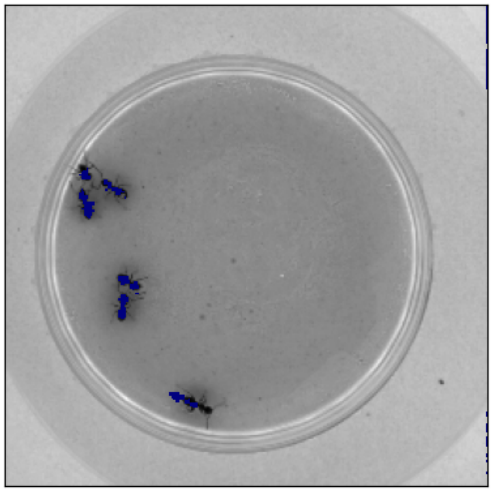
```python
def LucasKanadeAffine(It, It1, threshold, num_iters):
    """
    :param It: template image
    :param It1: Current image
    :param threshold: if the length of dp is smaller than the threshold, terminate the optimization
    :param num_iters: number of iterations of the optimization
    :return: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

    # put your implementation here
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
    ################### TODO Implement Lucas Kanade Affine ###################
    H_It, W_It = It.shape

    cor_leftop = np.array([[0, 0, 1]])
    cor_rightbot = np.array([[H_It, W_It, 1]])

    It1_y, It1_x = np.gradient(It1) #Gradient of current img

    x = np.arange(0, W_It, 1)
    y = np.arange(0, H_It, 1)
    spline_It = RectBivariateSpline(y, x, It) #Spline interpolation over a rectangular mesh
    spline_It1 = RectBivariateSpline(y, x, It1)
    spline_It1_x = RectBivariateSpline(y, x, It1_x)
    spline_It1_y = RectBivariateSpline(y, x, It1_y)

    x_axis = np.linspace(cor_leftop[0,0], cor_rightbot[0,0], int(round(W_It)))
    y_axis = np.linspace(cor_leftop[0,1], cor_rightbot[0,1], int(round(H_It)))
    X_grid, Y_grid = np.meshgrid(x_axis, y_axis)
    template = spline_It.ev(Y_grid, X_grid)

    # Iterations for finding optimal dp
    dp = [[1000], [1000]] #Big dp for while loop
    counter = 1
    while np.square(dp).sum() > threshold and counter <= num_iters:
        # Warp Current Image
        cor_leftop_wp = M@cor_leftop.T
        cor_rightbot_wp = M@cor_rightbot.T

        x_axis_wp = np.linspace(cor_leftop_wp[0,0], cor_rightbot_wp[0,0], int(round(W_It)))
        y_axis_wp = np.linspace(cor_leftop_wp[1,0], cor_rightbot_wp[1,0], int(round(H_It)))
        X_grid_wp, Y_grid_wp = np.meshgrid(x_axis_wp, y_axis_wp)
```

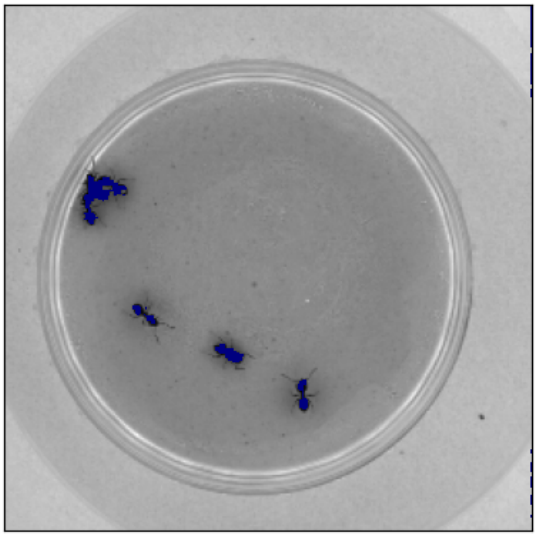
```
        It1_wp = spline_It1.ev(Y_grid_wp, X_grid_wp)

        #Get A and b
        It1_wp_x = spline_It1_x.ev(Y_grid_wp, X_grid_wp)
        It1_wp_y = spline_It1_y.ev(Y_grid_wp, X_grid_wp)
        Del_It1_wp = np.vstack((It1_wp_x.ravel(),It1_wp_y.ravel())).T

        A = np.zeros((H_It*W_It, 6))

        for i in range(H_It):
            for j in range(W_It):
                #I is (1,2) for each pixel
                #Jacobiani is (2,6)for each pixel
                Del_It1_wp_point = np.array([Del_It1_wp[i*W_It+j]]).reshape(1,2)
                jacob_point = np.array([[j, i, 1, 0, 0, 0], [0, 0, 0, j, i, 1]])
                A[i*W_It+j] = Del_It1_wp_point @ jacob_point

        b = (template - It1_wp).reshape(-1,1)

        #Solve argmin|Ax-b|^2 for finding dp
        dp = np.linalg.inv(A.T@A) @ (A.T) @ b

        #Updating
        M[0,0] += dp[0,0]
        M[0,1] += dp[1,0]
        M[0,2] += dp[2,0]
        M[1,0] += dp[3,0]
        M[1,1] += dp[4,0]
        M[1,2] += dp[5,0]

        counter += 1

    return M
```

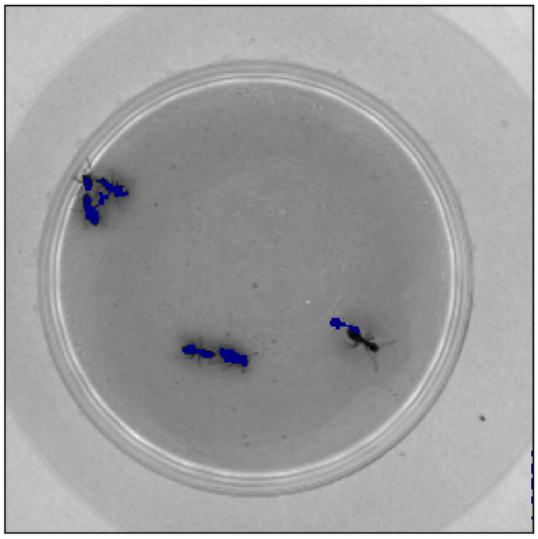
Q2.2

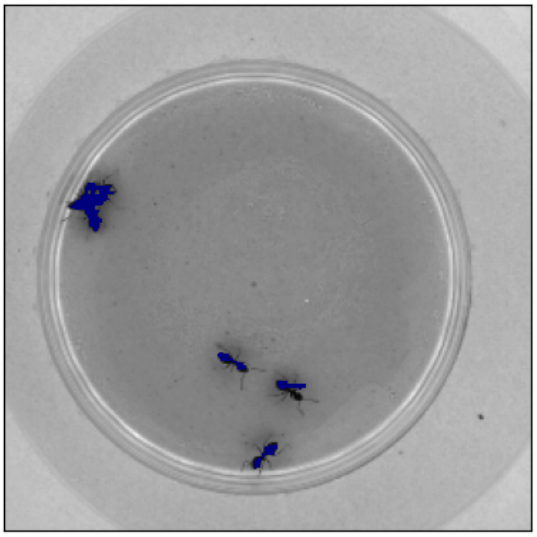
```python
def SubtractDominantMotion(image1, image2, threshold, num_iters, tolerance):
    """
    :param image1: Images at time t
    :param image2: Images at time t+1
    :param threshold: used for LucasKanadeAffine
    :param num_iters: used for LucasKanadeAffine
    :param tolerance: binary threshold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

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

    ################## TODO Implement Substract Dominent Motion ##################
    M = LucasKanadeAffine(image1, image2, threshold, num_iters)
    image2_wp = affine_transform(image2, M, output_shape = image1.shape)
    image2_wp = binary_dilation(binary_erosion(image2_wp))

    abs_diff = np.abs(image1 - image2_wp)
    mask = (abs_diff > tolerance)

    return mask.astype(bool)
```

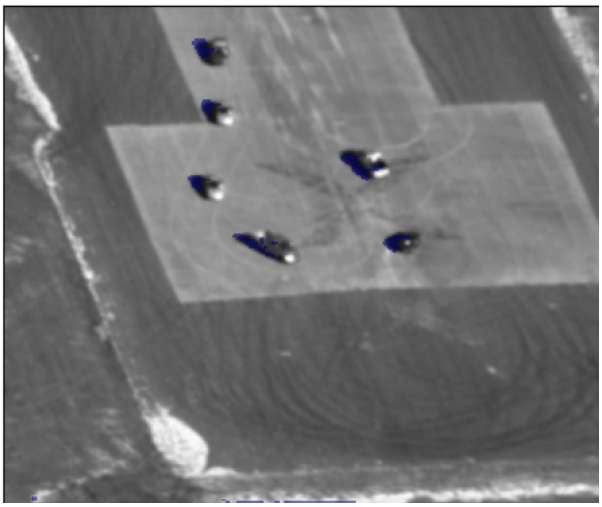
Q2.3

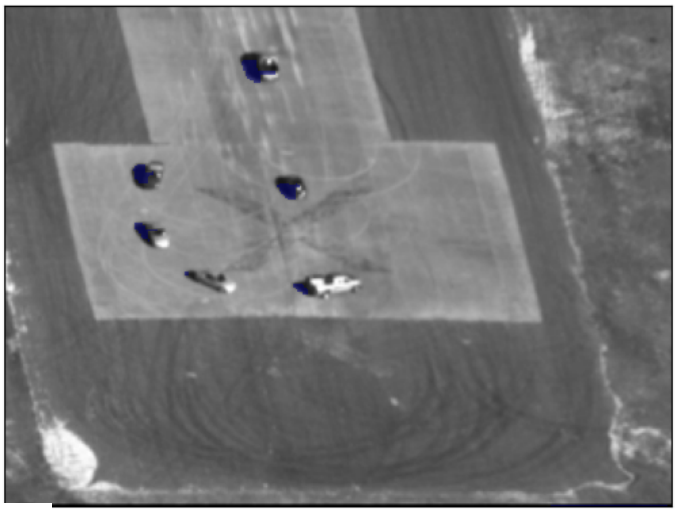


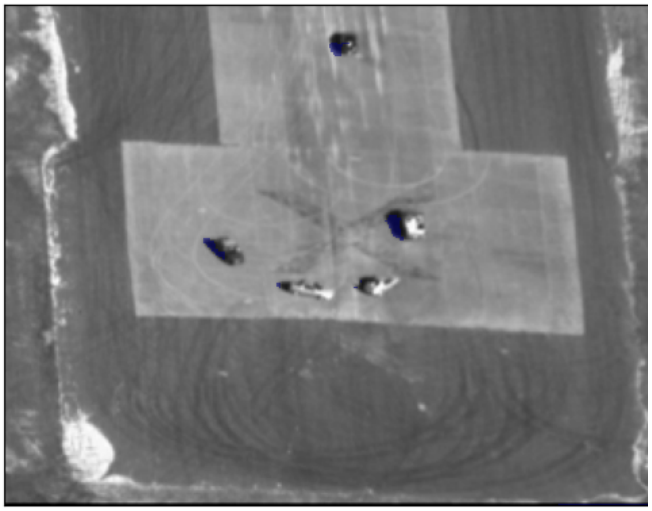


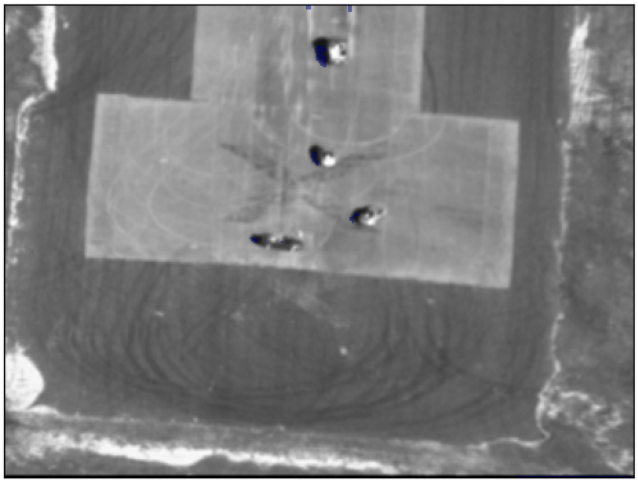








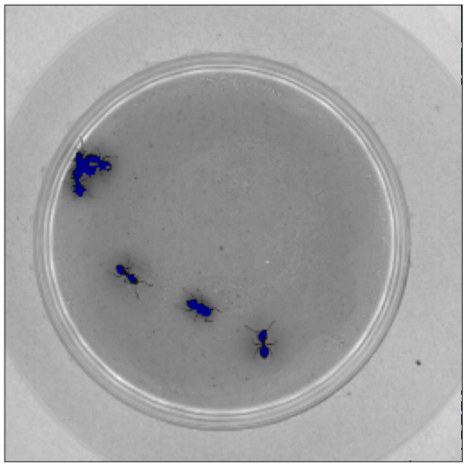


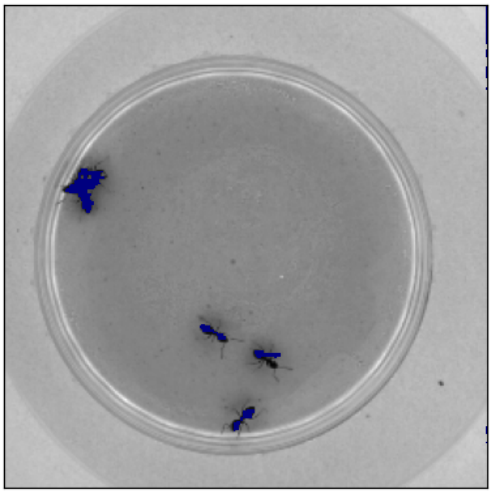


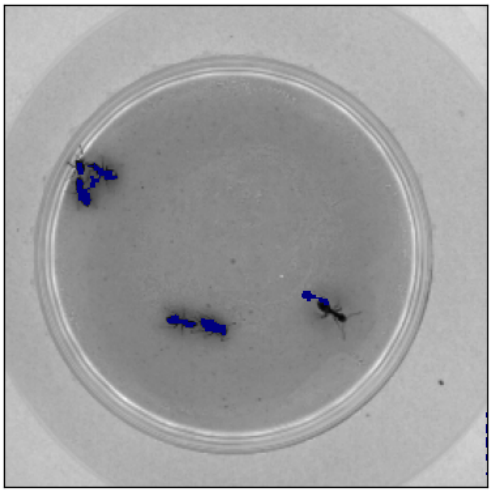
Masks here are very faint since of a small alpha value as bellow:

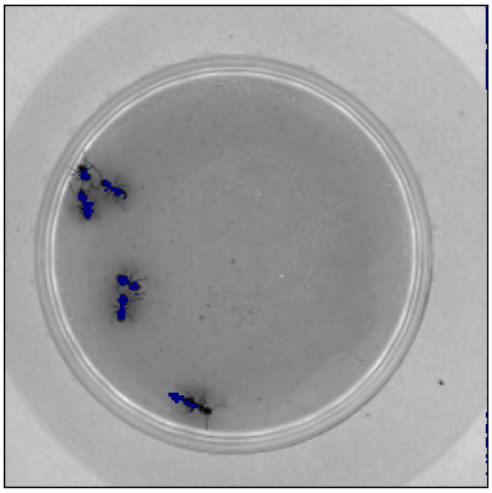
```
ax.imshow(np.ma.masked_where(np.invert(mask), mask), cmap='jet', alpha=1)
```

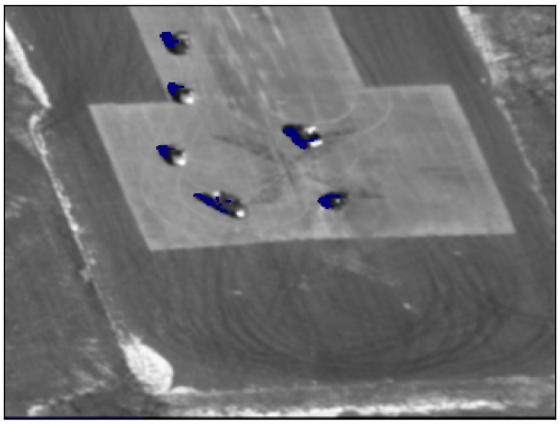
Q3.1

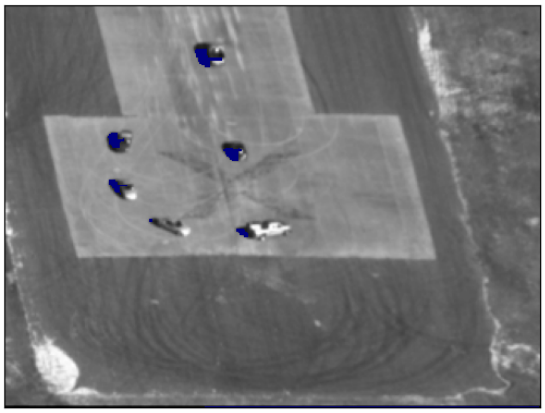


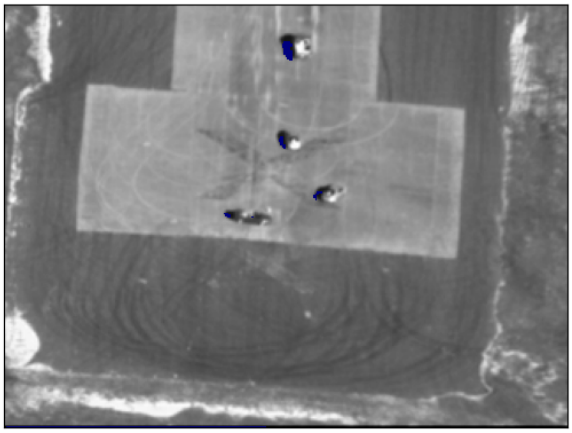


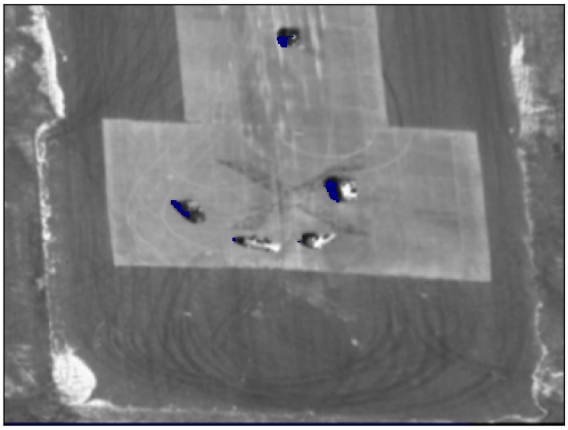












```
tAntSequence.py
100%|                                                                                | 124/124 [02:01<00:00,
tAntSequence_InvComp.py
100%|                                                                                | 124/124 [00:48<00:00,
```

After implementing Inverse Composition, the tracking time for ants reduced 2m

1s to 48s (x 2.52 faster)

```
tAerialSequence_InvComp.py
100%|                                                                                | 149/149 [05:16<00:00,
tAerialSequence.py
100%|                                                                                | 149/149 [40:50<00:00, 16.44s/it]
```

In terms of tracking vehicles in Aerial sequence, the tracking time reduced from 40m 50s to 5m 16s (x 7.75 faster).

Inverse compositional approach is more computationally efficient because the template doesn't change, so we can pre-compute $A^T A^{-1} A^T$ term at outside of every iteration.

Q4

I tried to track motions in 10s my squat clean video. To perform more robust tracking with the illumination effect of the gym, I normalized each frame individually to [0, 1] using frame.min(), and frame.max().

```
cap = cv2.VideoCapture('../data/Clean.mp4')
frameCount = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
frameWidth = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
frameHeight = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

buf = np.empty((frameHeight, frameWidth, int(frameCount/2)), np.dtype('float32'))

fc = 0
ret = True
i = 0

while (fc < frameCount  and ret):
    if (fc%2 == 0):
        ret, frame = cap.read()
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        frame_min = frame.min(axis=(0, 1), keepdims=True)
        frame_max = frame.max(axis=(0, 1), keepdims=True)
        frame = (frame - frame_min) / (frame_max - frame_min)
        buf[:, :, i] = frame
        i += 1
    fc += 1

cap.release()
np.save('../data/squatclean_norm.npy', buf)
```