Q1.1

(1) [[1, 0] [0, 1]]

(2)$A = \text{diagonal}[\partial \tau_{t+1}(x_1')/\ \partial x_1'^{T} \cdots \partial \tau_{t+1}(x_N')/\ \partial x_N'^{T}]$

$$*[\partial W(x_1;p)/\ \partial p^{T} \cdots \partial W(x_N;p)/\ \partial p^{T}]^{T}$$

$$b = [\ \tau_{t+1}(x_1') - \tau_t(x_1) \cdots \tau_{t+1}(x_N') - \tau_t(x_N)]^{T}$$

(3) $A^{T}A$ should be invertible

Q1.2

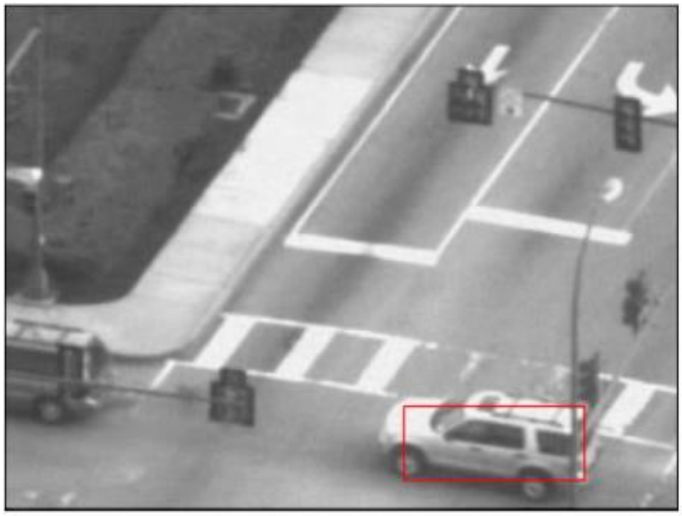
```
def LucasKanade(It, It1, rect, threshold, num_iters, p0=np.zeros(2)):
    """
    :param It: template image
    :param It1: Current image
    :param rect: Current position of the car (top left, bot right
coordinates)
    :param threshold: if the length of dp is smaller than the
threshold, terminate 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
    x_min, y_min, x_max, y_max = rect[0], rect[1], rect[2], rect[3]
    It_spline = RectBivariateSpline(np.arange(It.shape[0]),
np.arange(It.shape[1]), It)
    It1_spline = RectBivariateSpline(np.arange(It1.shape[0]),
np.arange(It1.shape[1]), It1)

    x_temp = np.arange(x_min, x_max + 0.1)
    y_temp = np.arange(y_min, y_max + 0.1)
    x_meshIt, y_meshIt = np.meshgrid(x_temp, y_temp)
    It_interp = It_spline.ev(y_meshIt, x_meshIt)

    for i in range(num_iters):
        x_cur = np.arange(x_min + p[0], x_max + p[0] + 0.1)
        y_cur = np.arange(y_min + p[1], y_max + p[1] + 0.1)
        x_meshIt1, y_meshIt1 = np.meshgrid(x_cur, y_cur)
        It1_interp = It1_spline.ev(y_meshIt1, x_meshIt1)

        It1_interpX = It1_spline.ev(y_meshIt1, x_meshIt1, dx=0, dy=1)
        It1_interpY = It1_spline.ev(y_meshIt1, x_meshIt1, dx=1, dy=0)
        A = np.vstack((It1_interpX.flatten(),
It1_interpY.flatten())).T
        b = It_interp.flatten() - It1_interp.flatten()
        H = A.T @ A
        delta_p = np.linalg.inv(H) @ (A.T @ b)
        p[0], p[1] = p[0]+delta_p[0], p[1]+delta_p[1]
        if np.sum(delta_p**2) < threshold:
            break

    return p
```

Q1.3

Car Sequence:

Frame 1:

Frame 100:



Frame 200:

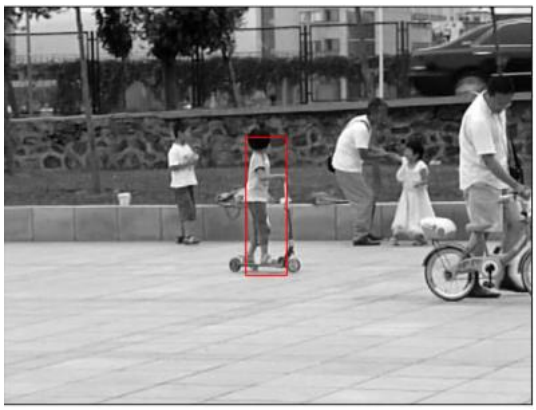
Frame 300:

Frame 400:

Girl Sequence:

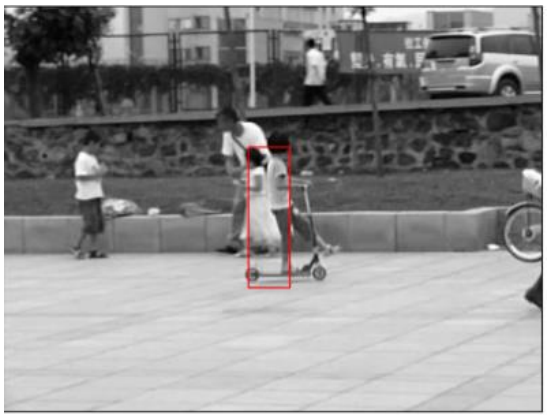
Frame 1:

Frame 20:

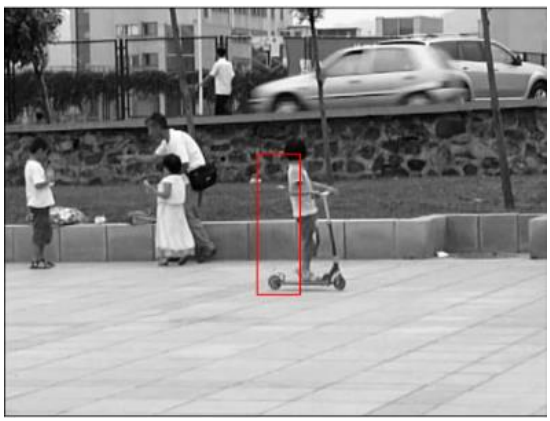


Frame 40:

Frame 60:



Frame 80:



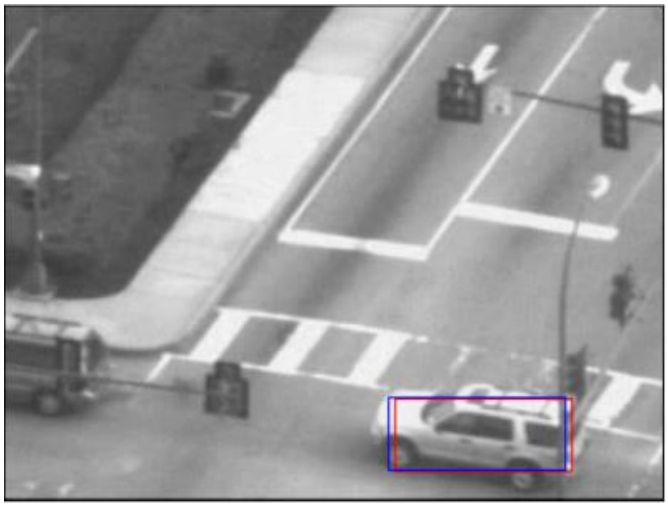
Q1.4

Here the red rectangles are created with the baseline tracker in Q1.3, the blue ones with the tracker in Q1.4.

Car Sequence:

Frame 1:

Frame 100:



Frame 200:

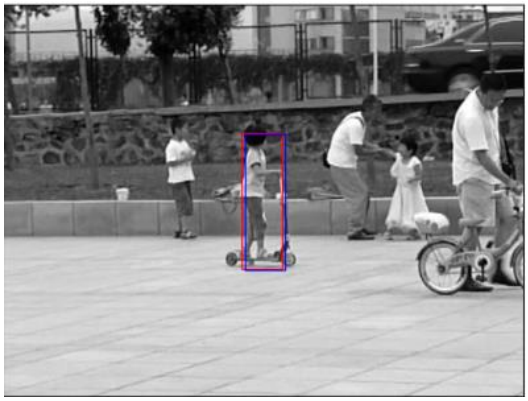
Frame 300:

Frame 400:

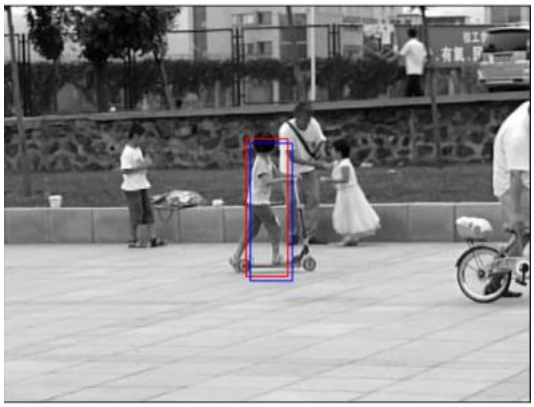
Girl Sequence:

Frame 1:

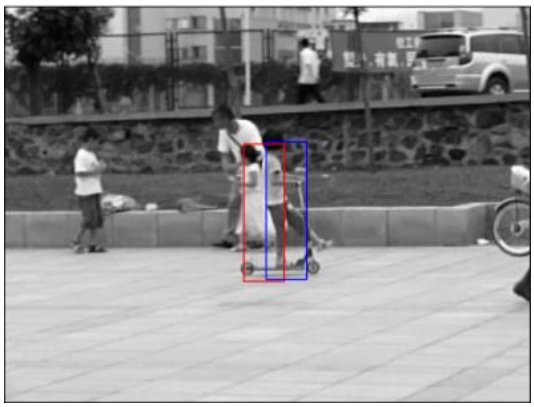
Frame 20:



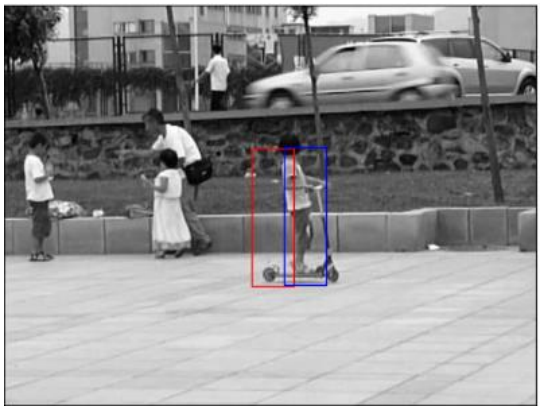
Frame 40:



Frame 60:



Frame 80:



Q2.1

```
def LucasKanadeAffine(It, It1, threshold, num_iters):
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
    p = M.flatten()
    It1_spline = RectBivariateSpline(np.arange(It1.shape[0]),
np.arange(It1.shape[1]), It1)
    x_min, y_min, x_max, y_max = 0, 0, It.shape[1]-1, It.shape[0]-1
    x_temp = np.arange(x_min, x_max + 0.1)
    y_temp = np.arange(y_min, y_max + 0.1)
    x_meshIt1, y_meshIt1 = np.meshgrid(x_temp, y_temp)

    for i in range(num_iters):
        x_cur = p[0]*x_meshIt1 + p[1]*y_meshIt1 + p[2]
        y_cur = p[3]*x_meshIt1 + p[4]*y_meshIt1 + p[5]
        # find valid points
        points = (x_cur > 0) & (x_cur < It.shape[1]) & (y_cur > 0) &
(y_cur < It.shape[0])
        x_cur, y_cur = x_cur[points], y_cur[points]
        It1_interp = It1_spline.ev(y_cur, x_cur)
        It1_interpX = It1_spline.ev(y_cur, x_cur, dx=0,
dy=1).flatten()
        It1_interpY = It1_spline.ev(y_cur, x_cur, dx=1,
dy=0).flatten()
        # calculate Affine matrix
        A = np.zeros((It1_interpX.shape[0], 6))
        A[:, 0] = np.multiply(It1_interpX,
x_meshIt1[points].flatten())
        A[:, 1] = np.multiply(It1_interpX,
y_meshIt1[points].flatten())
        A[:, 2] = It1_interpX
        A[:, 3] = np.multiply(It1_interpY,
x_meshIt1[points].flatten())
        A[:, 4] = np.multiply(It1_interpY,
y_meshIt1[points].flatten())
        A[:, 5] = It1_interpY
        # calculate matrix b
        b = It[points].flatten() - It1_interp.flatten()
        delta_p = np.linalg.inv(A.T @ A) @ (A.T @ b)
        p += delta_p.flatten()

        if np.sum(delta_p ** 2) <= threshold:
            break
    M = np.reshape(p, (2, 3))
    return M
```

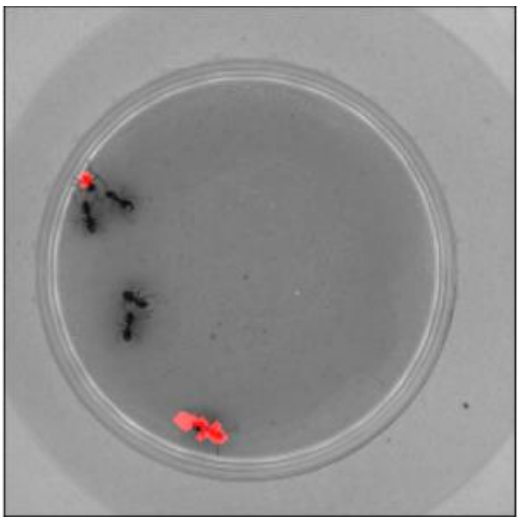
Q2.2

```
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)

    M = LucasKanadeAffine(image1, image2, threshold, num_iters)

    image1_warp = affine_transform(image1, M, image1.shape)
    diff = np.abs(image2 - image1_warp)
    mask[diff < tolerance] = 0

    mask = binary_dilation(mask, np.ones((3, 3)))
    mask = binary_erosion(mask, np.ones((3, 3)))
    return mask
```

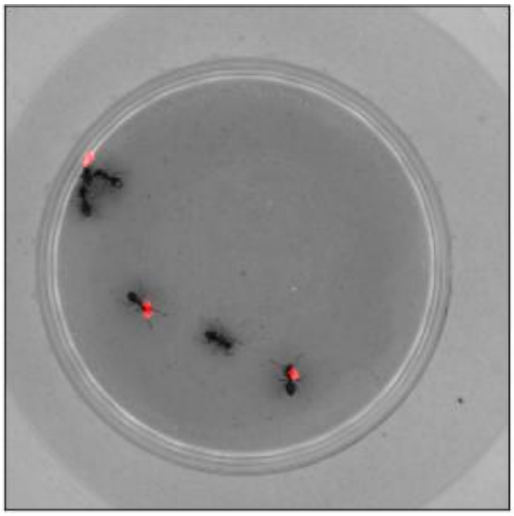
Q2.3

Ant Sequence:

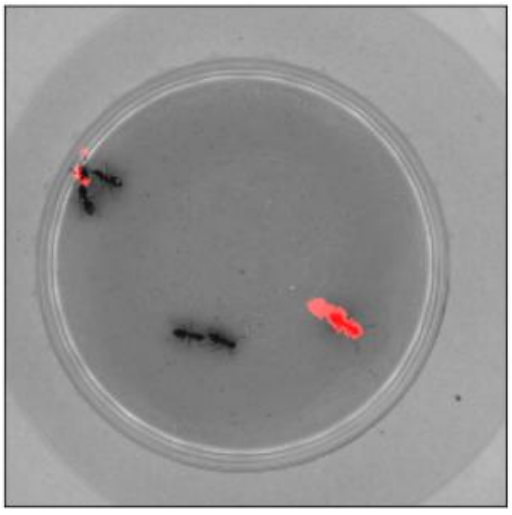
Frame 30:



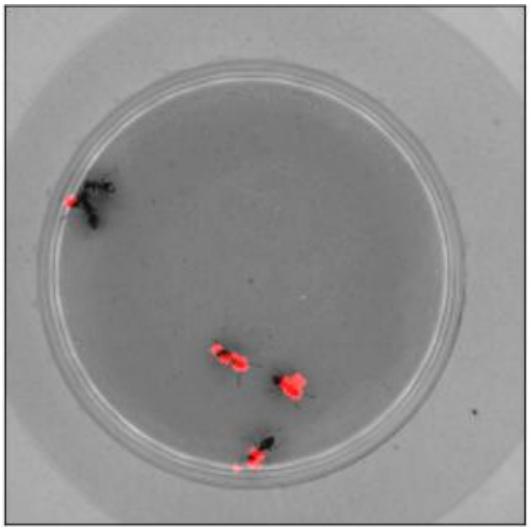
Frame 60:



Frame 90:

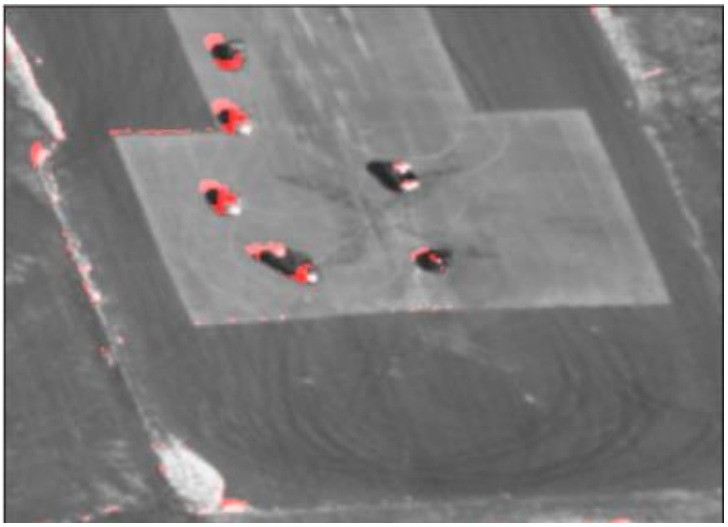


Frame 120:

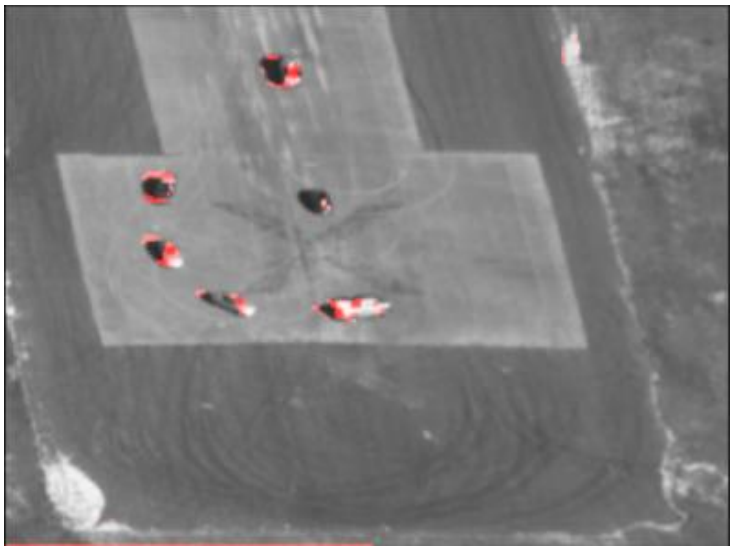


Aerial Sequence:

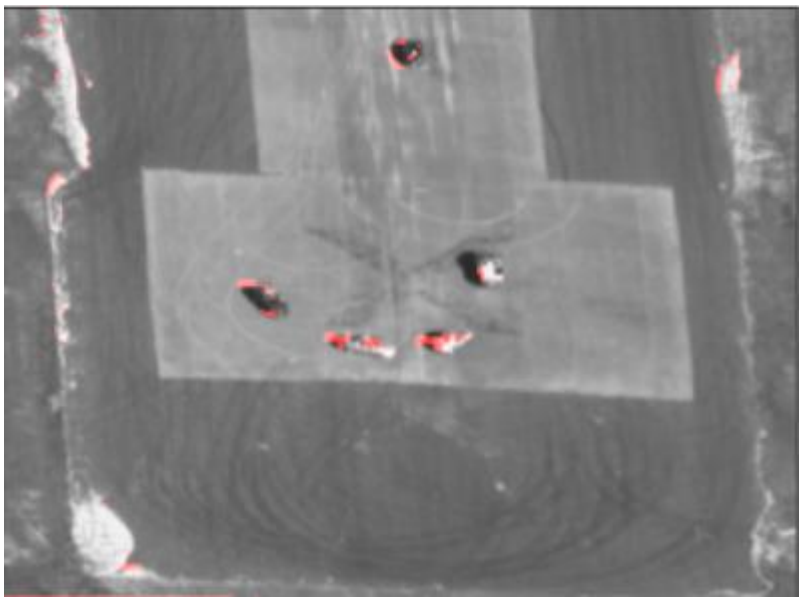
Frame 30:



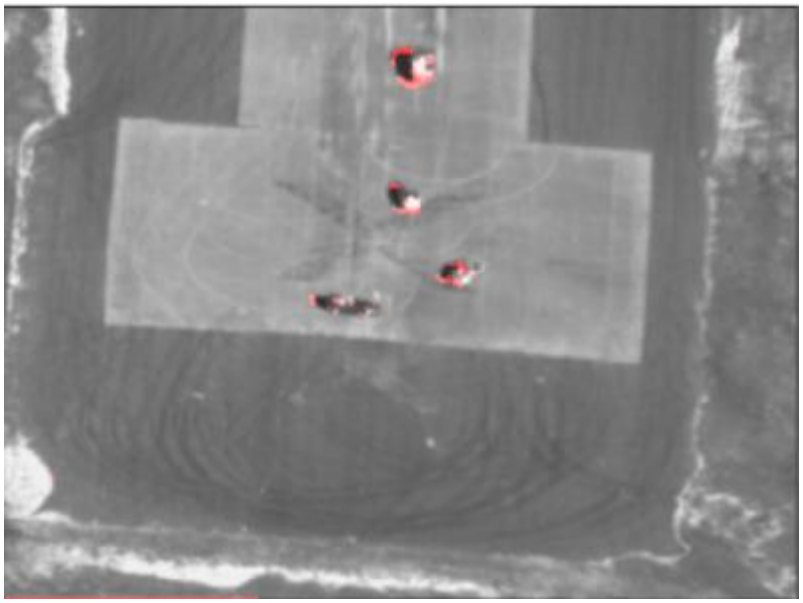
Frame 60:



Frame 90:



Frame 120:

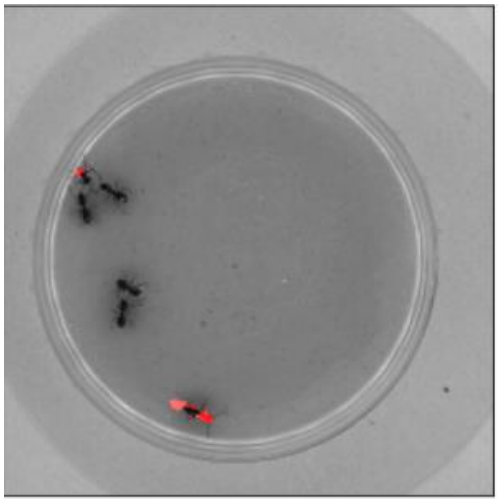


Q3.1

Ant Sequence by Inverse Composition: 13s

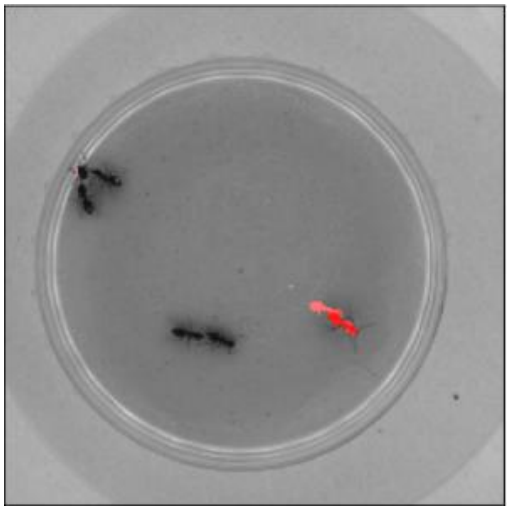
Ant Sequence by Lucas-Kanade Affine: 15s

Frame 30:

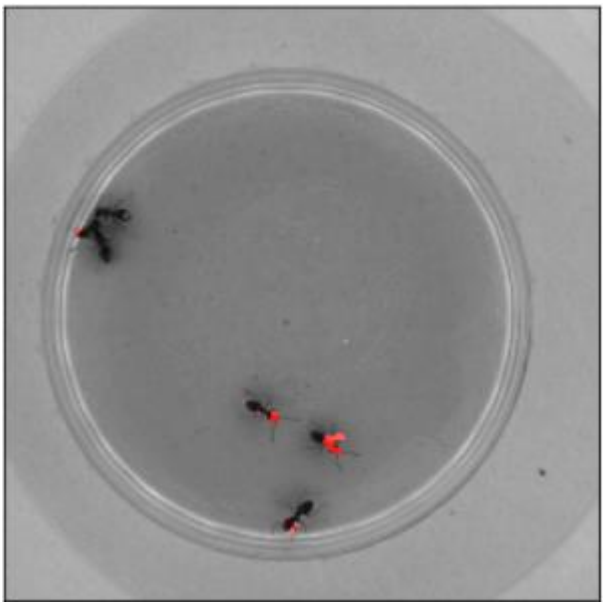


Frame 60:

Frame 90:



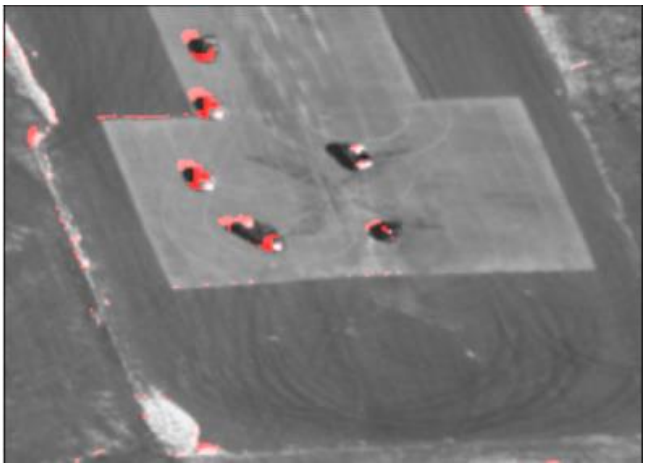
Frame 120:



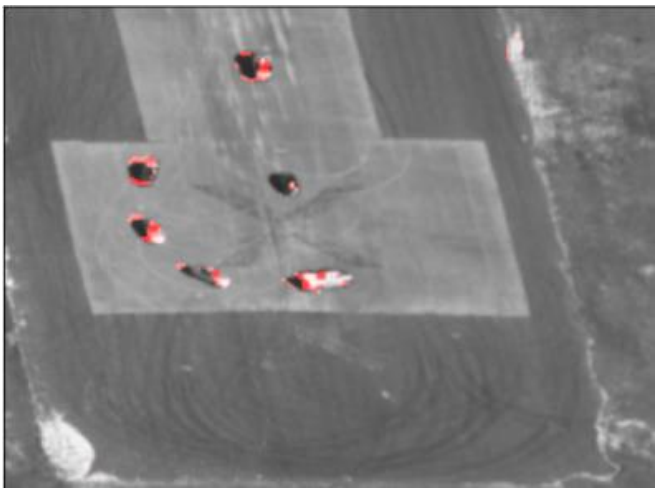
Aerial Sequence by Inverse Composition: 30s

Aerial Sequence by Lucas-Kanade Affine: 52s

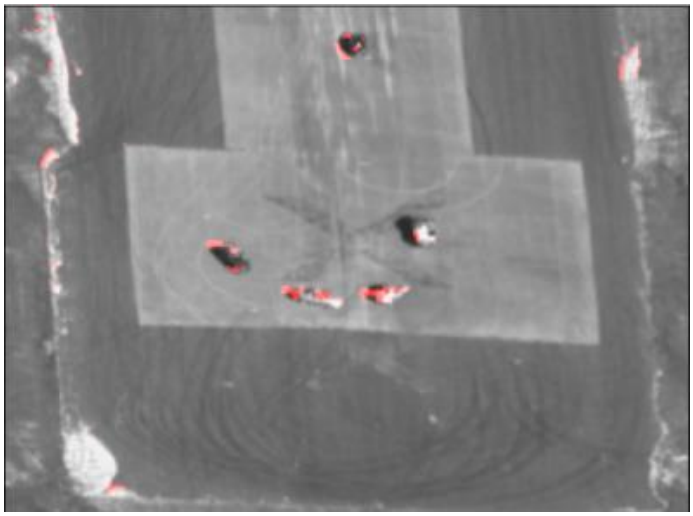
Frame 30:



Frame 60:



Frame 90:



Frame 120:

The results show that the inverse composition approach is much faster than classical Lucas-Kanade approach, especially for videos with a great amount of frames. Furthermore, inverse composition approach still shows good objects tracking performance compared to classical algorithm. I think this tradeoff is acceptable.

One important reason why inverse composition approach is more computationally efficient than the classical approach is the computation complexity. In classical approach, we need to update matrix A in each iteration according to different p. On the contrary, matrix A is the same in every iteration of inverse composition approach which only needs to be computed once before start iterating. Therefore, the computation times are much smaller in inverse composition approach.