Q1.1

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

$$W(x;p) = \begin{pmatrix} x + p_1 \\ y + p_2 \end{pmatrix}$$

b. A =

$$\begin{pmatrix} \overbrace{(\nabla I_{t+1})_{x_1'}}^{\text{Gradient}} * \overbrace{(J)_{x_1'}}^{\text{Jacobian}} \\ (\nabla I_{t+1})_{x_2'} * (J)_{x_2'} \\ \vdots \\ (\nabla I_{t+1})_{x_1} * (J)_{x_1} \end{pmatrix}$$

b = $[I_t(x,y) - I_{t+1}(x,y)]$; Error Term

c. $A^T A$ must be invertible and the det $(A^T A)$ should not equal 0.

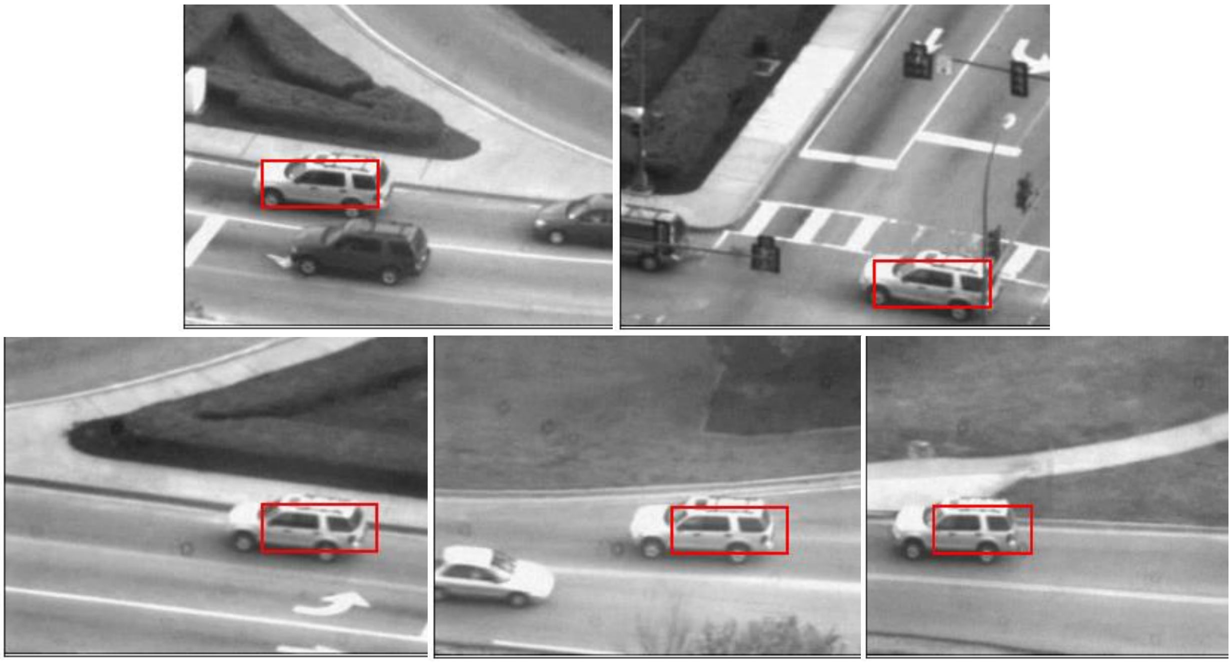
Q1.2

```python
import numpy as np
import scipy
from scipy.interpolate import RectBivariateSpline
import cv2


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 optimiz
    :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 threshold
    # Compute the optimal local motion from frame It to from It+1 that minimizes Equation 3
    # It is the image frame; rect is the 4x1 vector rep a rectangle with all pixel conditions

    #where to get x and y values? From the frame in It!

    #Current image
    x1 = rect[0]
    x2 = rect[2]
    y1 = rect[1]
    y2 = rect[3]

    # Ix is the x component of the gradient vector (di/dx)
    # Iy is the y component of the gradient vector

```

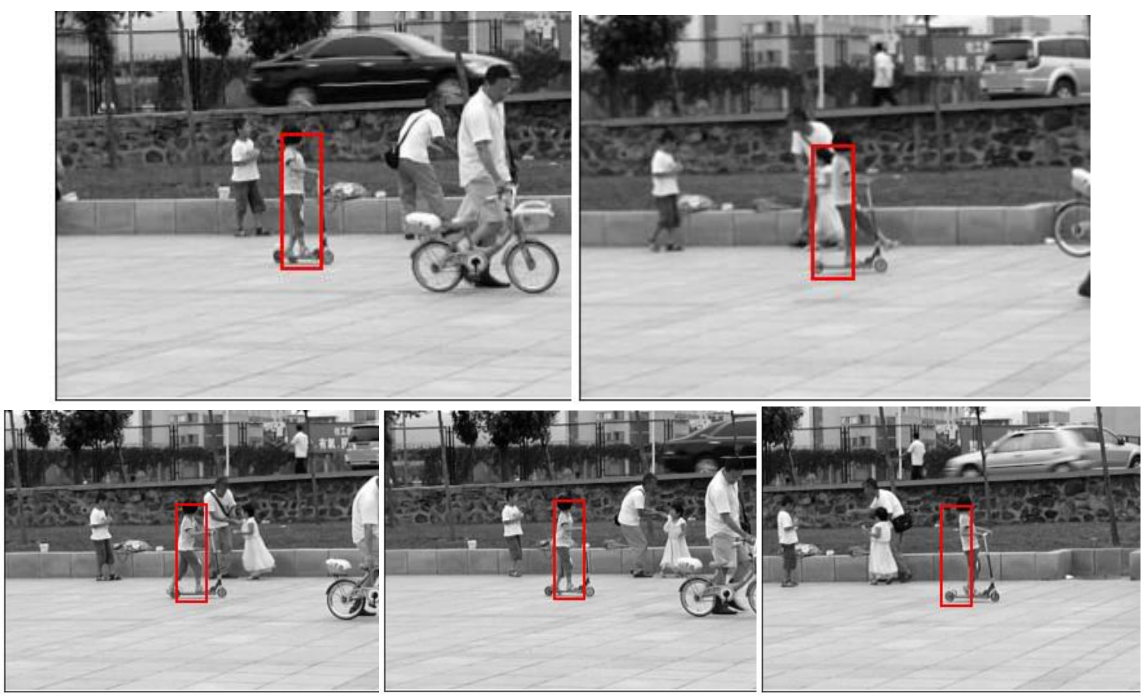
```python
    #Top left, Bottom Right
    x1x2 = np.arange(x1,x2)
    y1y2 = np.arange(y1,y2)
    y1_y2,x1_x2 = np.meshgrid(y1y2,x1x2)


    # x_warp,y_warp = np.meshgrid(x2,y2)

    # Spline allows us to extract the gradient at any point/pixel in the image
    # We are interpolating the derivative

    #Compute the spline for the image
    splineIt1 = RectBivariateSpline(np.arange(0,It1.shape[0]),np.arange(0,It1.shape[1]),It1) #
    splineIt = RectBivariateSpline(np.arange(0,It.shape[0]),np.arange(0,It.shape[1]),It)
    # Compute delta_p in order to shift the bounding box (Reiew lecture slides)
    p = np.copy(p0)
    t_x = splineIt.ev(y1_y2,x1_x2) #template image for error
    jacobian = [[1,0],[0,1]]

    for i in range(int(num_iters)):
        #Warping I to compute I(W(x;p)) requires interpolating the image I at the sub-pixel lc
        #W(x:p)
        xp = np.arange(x1,x2)+p[0]
        yp = np.arange(y1,y2)+p[1]
        y_p,x_p = np.meshgrid(yp,xp)

        #1. Warp I with (x;p)
        warped_I_rect = splineIt1.ev(y_p,x_p)

        #2. Compute the error
        error = (t_x-warped_I_rect).flatten()

        #3. warp the gradient
```

```python

        #3. warp the gradient
        grad_x = splineIt1.ev(y_p,x_p, dx=0,dy=1)
        grad_y = splineIt1.ev(y_p,x_p, dx=1,dy=0)

        #Substep: Concatenate gradients
        grad_x_y = np.stack((grad_x,grad_y),axis=2)
        grad_x_y = np.reshape(grad_x_y,(-1,2)) #Nx2

        #4. Jacobian in translation is identity matrix so no need to evaluate at W(x;p)

        #5. Compute steepest descent images
        steepest_descent = np.dot(grad_x_y,jacobian) #Nx2
        #6. Compute Hessian Matrix
        #6a. evaluate steepest_descent.T *steepest_descent
        hessian = np.dot(steepest_descent.T,steepest_descent)

        #7. Compute the sum
        A = np.dot(steepest_descent.T,error)

        #8. Compute deltap
        hessian_inv = np.linalg.inv(hessian)

        deltap = np.dot(hessian_inv,A)

        p += deltap

        if(np.sum(np.square(deltap)) < threshold):
            break

    return p
```

Q1.3



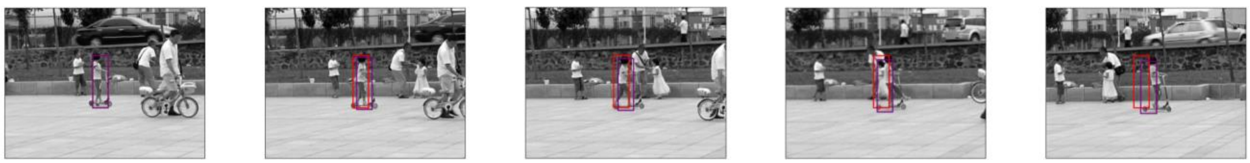
Car Tracking



Girl Tracking

Q1.4

Car with Template Correction



Girl with Template Correction

Q2.1

```python
def LucasKanadeAffine(It, It1, threshold, num_iters):
    """
    :param It: template image (It)
    :param It1: Current image (It+1)
    :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
    """

    #Current image
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])

    x1x2 = np.arange(0,(It.shape[1]-1)) #Instead of rectangle we are now looking at the entire image
    y1y2 = np.arange(0,(It.shape[0]-1))


    splineIt1 = RectBivariateSpline(np.arange(0,It1.shape[0]),np.arange(0,It1.shape[1]),It1) # Rectangular Spline for current
    splineIt = RectBivariateSpline(np.arange(0,It.shape[0]),np.arange(0,It.shape[1]),It)


    x1_x2,y1_y2 = np.meshgrid(x1x2, y1y2)

    # put your implementation here
    p = M.flatten()

    deltap = [[1], [0], [0], [0], [1], [0]]
    for i in range(int(num_iters)):


        xp = x1_x2*(M[0,0])+y1_y2*M[0,1]+M[0,2]
        yp = x1_x2*(M[1,0])+y1_y2*(M[1,1])+M[1,2]

        M = np.array([[(1+p[0]),p[1],p[2]],[p[3],(1+p[4]),p[5]],[0,0,1]])

        #We only need to consider the common points between the current image and the warped current image
```

```python
        x_comparison = np.logical_and(0<xp,xp<It.shape[1])
        y_comparison = np.logical_and(0<yp,yp<It.shape[0])
        valid_pts = np.logical_and(x_comparison,y_comparison)

        # Check deltap against threshold

        xp,yp = xp[valid_pts],yp[valid_pts]

        #y_p,x_p = np.meshgrid(yp,xp)

        #1. Warp I with (x;p)

        # warped_I = affine_transform(It1,M)
        warped_I = splineIt1.ev(yp, xp)
        #2. Compute error
        error =(It[valid_pts].flatten()-warped_I.flatten())

        #3. warp the gradient
        grad_x = splineIt1.ev(yp,xp, dx=0,dy=1)
        grad_y = splineIt1.ev(yp,xp, dx=1,dy=0)

        #4.Evaluate the jacobian at (x;p); #What is x and y in the jacobian? - pixel values
        A=np.zeros((It.shape[0]*It.shape[1],6)) #Nx6, with N being number of pixels

        #for y in range(It.shape[0]):
           # for x in range(It.shape[1]):
            #    jacobian = np.array([[x, 0, y, 0, 1, 0],[0, x, 0, y, 0, 1]])
                #5. Compute steepest_descent
              #  steepest_descent = np.dot(grad_x_y,jacobian)
               # sd_save[x*y] = steepest_descent[x*y]
                #print(x)

        A = np.vstack((grad_x*xp,grad_x*yp,grad_x,grad_y*xp,grad_y*yp,grad_y)).T

        # import pdb; pdb.set_trace()
        #A = np.reshape(A,(It.shape[1]*It.shape[0],6))


        #6. Compute Hessian Matrix
        #6a. evaluate steepest_descent.T *steepest_descent
        hessian = np.dot(A.T,A)

        #7. Compute the sum
        x = np.dot(A.T,error)
```

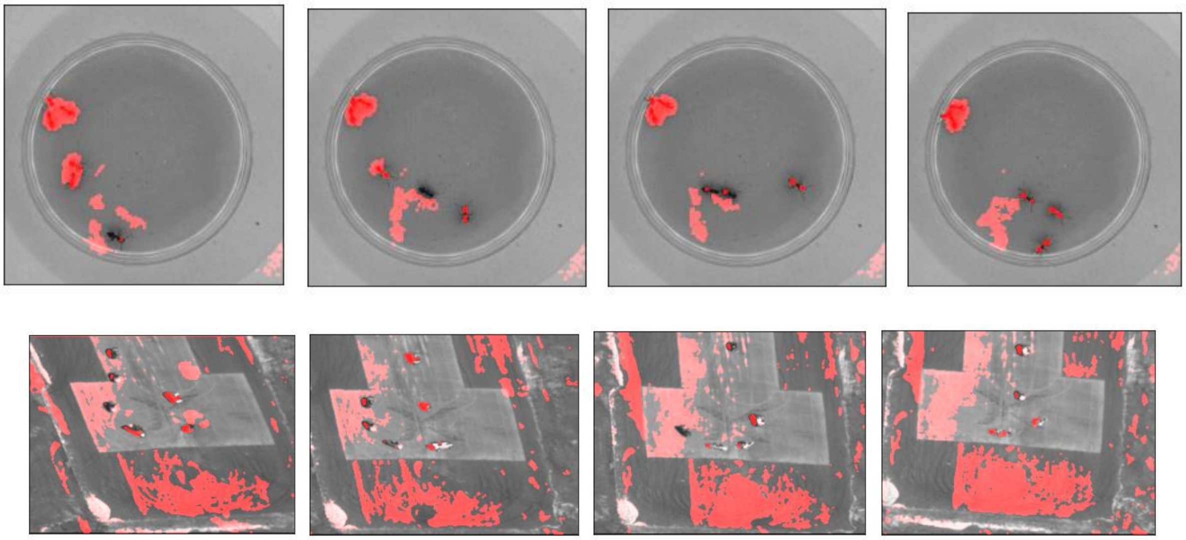
```python
        #8. Compute deltap
        hessian_inv = np.linalg.inv(hessian)

        deltap = np.dot(hessian_inv,x)

        p+=deltap

        #9. Update M
        deltaM = np.array([[p[0],p[1],p[2]],[p[3],p[4],p[5]],[0,0,0]])

        # print(np.linalg.norm(deltap),threshold)
        # threshold = .75
        if(np.sum(np.square(deltap)) < threshold):
            break

        M += deltaM
    ################### TODO Implement Lucas Kanade Affine ###################

    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)

    # t = tolerance #what is the tolerance for?
    # #Binary_erosion (uses a structuring element for shrinking the shapes in an image) and dilatio
    # # (a structuring element used for expanding the shapes in an image
    # # The binary dilation of an image by a structuring element is the locus of the points covered
    # # the structuring element, when its center lies within the non-zero points of the image.)

    #M = np.linalg.inv(LucasKanadeAffine(image1, image2, threshold, num_iters))
    M = np.linalg.inv(InverseCompositionAffine(image1, image2, threshold, num_iters))

    # #Warp the image using M
    warp_image1 = affine_transform(image1,M)

    array_structure = np.array(([0,1,0],[1,1,1],[0,1,0]))

    #Getting the difference
    abs_diff = np.abs(image2 - warp_image1)
    mask = abs_diff > tolerance

    #Erosion and dilation

```

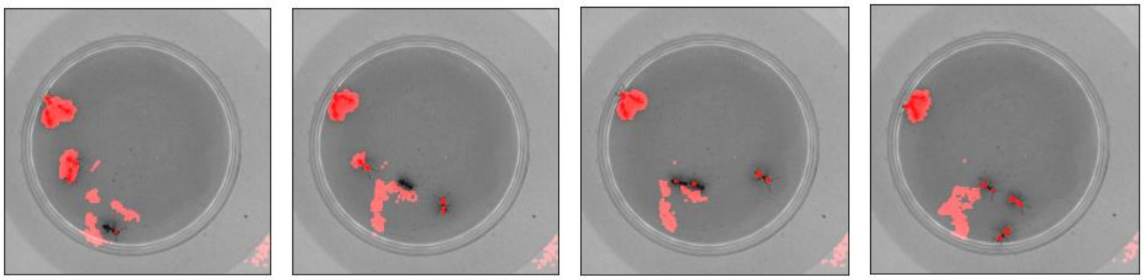
```python
    #Erosion and dilation

    mask = binary_erosion(mask, array_structure)
    mask = binary_dilation(mask, array_structure)

    mask = abs_diff > tolerance

    return mask.astype(bool)

```

Q2.3

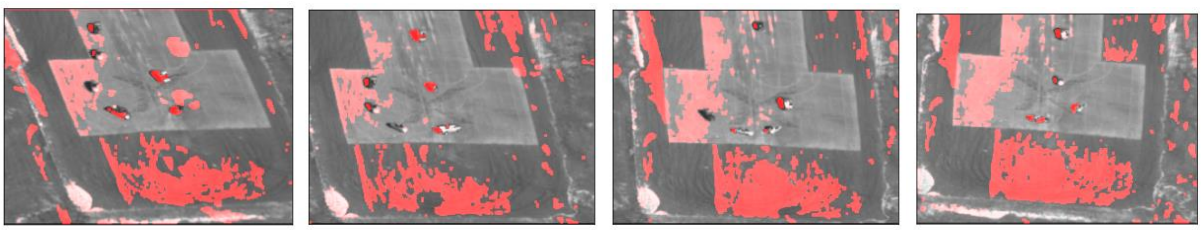


Q3.1

Runtime Performance: 23 seconds (LKAffine); 12 seconds (Inverse) – Ant Sequence



Runtime Performance: 56 seconds (LKAffine); 36 seconds (Inverse)



When comparing the results between the Lucas Kanade Affine and the Inverse composition affine there is no noticeable difference in the results from the image processing but there is a big difference in how long the runtime is. This change in runtime performance could potentially be because of the pre-computations that happen outside of the iteration in the inverse composition affine. We are not computing the hessian for each and every update to p (via deltap) and the hessian remains constant. This reduces effort and time on the computation, when looking at the ant sequence the results are relatively similar so this means that either method will yield good results but going with the composition affine will require less computational effort (potentially a better method). (Note: The results for aerial sequence would be a lot better if the masking was done correctly).