16720 – hw3
mrohitth

Problem 1.1.1

Given: $I_{t+1}(x' + \Delta p) \approx I_{t+1}(x') + \partial I_{t+1}(x') / \partial x' \,.\, \partial W(x;p) / \partial p^T \,.\, \Delta p$

Here, $p = [px, py]^T$

$X' = W(x;p) = x+p$

Hence, W(x;p) has two parameters p1 and p2

$$W(x;p) = \begin{bmatrix} 1 & 0 & p1 \\ 0 & 1 & p2 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + p1 + 0 \\ 0 + p2 + y \end{bmatrix} = \begin{bmatrix} x + p1 \\ y + p2 \end{bmatrix}$$

$$\text{So, } \partial W/\partial p^T = \begin{bmatrix} \frac{\partial Wx}{\partial p1} & \frac{\partial Wx}{\partial p2} \\ \frac{\partial Wy}{\partial p1} & \frac{\partial Wy}{\partial p2} \end{bmatrix} = \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} \end{bmatrix}$$

Problem 1.1.2

Using first-order Taylor expansion we can linearize the objective function locally and on rearranging the equation to minimize we get as below.

$$\arg\min_{\Delta \mathbf{p}} \sum_{\mathbf{x}\in\mathbb{N}} \|\mathcal{I}_{t+1}(\mathbf{x}' + \Delta\mathbf{p}) - \mathcal{I}_t(\mathbf{x})\|_2^2$$

$$where \;\; \mathcal{I}_{t+1}(\mathbf{x}' + \Delta\mathbf{p}) \approx \mathcal{I}_{t+1}(\mathbf{x}') + \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial \mathcal{W}(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T} \Delta\mathbf{p}$$

$$\arg\min_{\Delta \mathbf{p}} \sum_{\mathbf{x}\in\mathbb{N}} \left\| \mathcal{I}_{t+1}(\mathbf{x}') + \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial \mathcal{W}(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T} \Delta\mathbf{p} - \mathcal{I}_t(\mathbf{x}) \right\|_2^2 \qquad (2)$$

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

$$\arg\min_{\Delta \mathbf{p}} \|\mathbf{A}\Delta\mathbf{p} - \mathbf{b}\|_2^2$$

On comparing the last two above forms we get that:

$$\mathbf{A} = \sum_{\mathbf{x}\in\mathbb{N}} \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial \mathcal{W}(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T}$$

$$\mathbf{b} = \sum_{\mathbf{x}\in\mathbb{N}} \mathcal{I}_t(\mathbf{x}) - \mathcal{I}_{t+1}(\mathbf{x}')$$

A represents the steepest descent images and **b** represents the error image.

Problem 1.1.3

For minimizing Δp in equation 2, we take its first derivative and equate it to zero and we get the following:

$$\sum_{\mathbf{x}\in\mathbb{N}} 2\mathbf{A}^\top(\mathbf{A}\Delta\mathbf{p}-\mathbf{b})=0$$

$$\sum_{\mathbf{x}\in\mathbb{N}} \mathbf{A}^\top\mathbf{A}\Delta\mathbf{p}=\sum_{\mathbf{x}\in\mathbb{N}} \mathbf{A}^\top\mathbf{b}$$

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

In order to solve for Δp, $\mathbf{A}^T\mathbf{A}$ must be **invertible** or **non-singular matrix** or must **have a non-zero determinant** to obtain a unique solution for Δp.

Problem 1.2

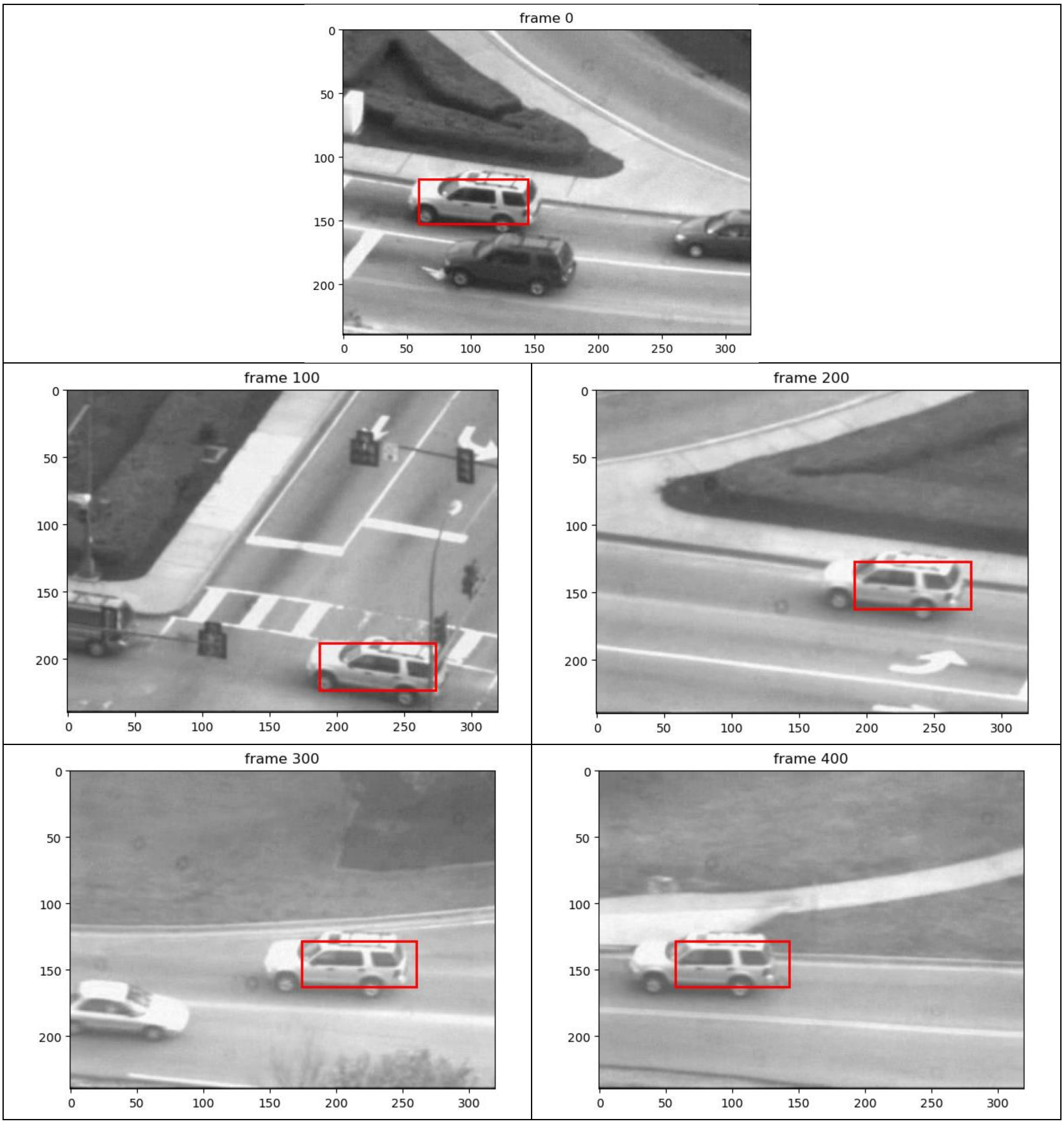
```python
def LucasKanade(It, It1, rect, threshold, num_iters, p0=np.zeros(2)):
    """
    :param It: template image
    :param It1: Current image
    :param rect: Current position of the car (top left, bot right coordinates)
    :param threshold: if the length of dp is smaller than the threshold, terminate 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
    p = p0

    h0, w0 = np.shape(It)
    h1, w1 = np.shape(It1)

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

    st0    = np.linspace(0, h0, num=h0, endpoint=False)
    stop0  = np.linspace(0, w0, num=w0, endpoint=False)
    st1    = np.linspace(0, h1, num=h1, endpoint=False)
    stop1  = np.linspace(0, w1, num=w1, endpoint=False)

    s0 = RectBivariateSpline(st0, stop0, It)
    s1 = RectBivariateSpline(st1, stop1, It1)

    w, h = int(x2-x1), int(y2-y1)

    c = 1
    k = 1
    x, y = np.mgrid[x1:x2+1:w*1j, y1:y2+1:h*1j]
    # print(x, y)
    while (c > threshold and k < num_iters):
        # print(k, c)
        dxp  = s1.ev(y+p[1], x+p[0], dy = 1).flatten()
        dyp  = s1.ev(y+p[1], x+p[0], dx = 1).flatten()
        It1p = s1.ev(y+p[1], x+p[0]).flatten()
        Itp  = s0.ev(y, x).flatten()
        A = np.zeros((w*h, 2*w*h))
        # print(A)
        for i in range(w*h):
            A[i, 2*i]   = dxp[i]
            A[i, 2*i+1] = dyp[i]
        Rs = m.repmat(np.eye(2), w*h, 1)

        A = np.matmul(A, Rs)
        b = np.reshape(Itp - It1p,(w*h, 1))
        # print(b)
        deltap = np.linalg.pinv(A).dot(b)
        c = np.linalg.norm(deltap)
        p = (p + deltap.T).ravel()
    return p
```

16720 – hw3
mrohitth

Problem 1.3

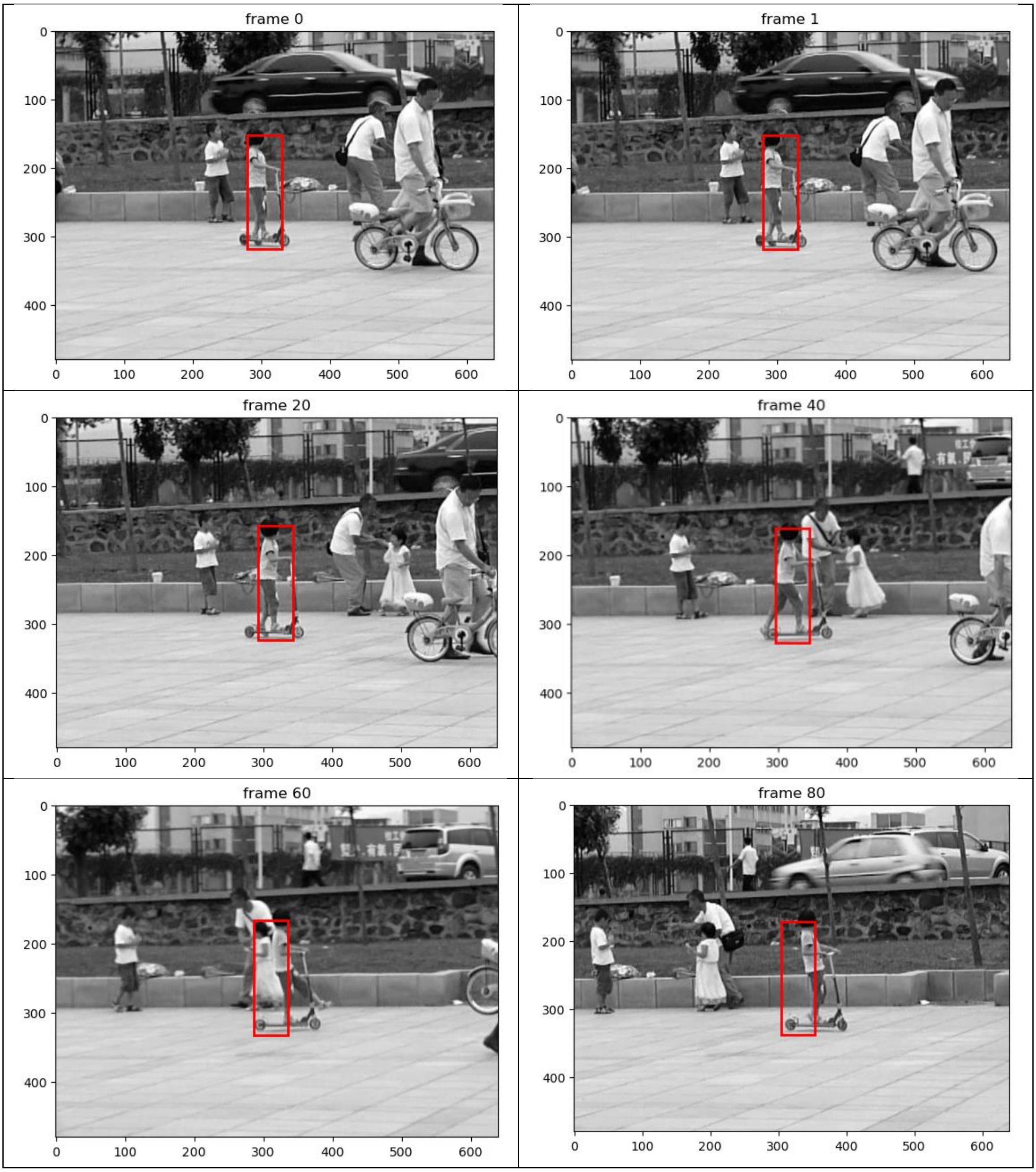
testCarSequence.py

```
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]
r_list = np.zeros((seq.shape[2]-1,4))

h,w,f = np.shape(seq)
for frame in range(f-1):
        It  = seq[:, :, frame]
        It1 = seq[:, :, frame+1]
        l = LucasKanade(It, It1, rect, threshold, num_iters)
        rect[0] += l[0] #x1
        rect[1] += l[1] #y1
        rect[2] += l[0] #x2
        rect[3] += l[1] #y2

        r_list[frame] = rect

        if (frame % 100 == 0 or frame == 0):

            plt.figure()
            plt.imshow(seq[:,:,frame], cmap='gray')
            rectangle = patches.Rectangle((int(rect[0]), int(rect[1])), (rect[2]-rect[0]),
                                    (rect[3]-rect[1]), fill=False, edgecolor='r', linewidth=2)

            plt.gca().add_patch(rectangle)
            plt.title('frame %d'%frame)
            plt.savefig('carseqframe' + str(frame) + '.png', bbox_inches='tight')
            plt.show()

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


frame 0
frame 100
frame 200
frame 300
frame 400

testGirlSequence.py

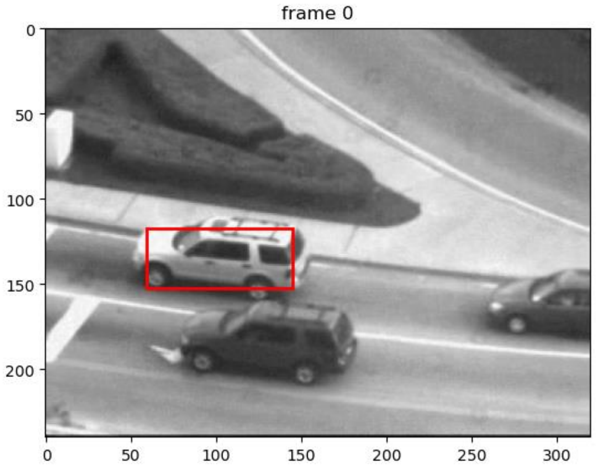
```
parser = argparse.ArgumentParser()
parser.add_argument('--num_iters', type=int, default=1e4,
                    help='number of iterations of Lucas-Kanade')
parser.add_argument('--threshold', type=float, default=1e-2,
                    help='dp threshold of Lucas-Kanade for terminating optimization')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

seq = np.load("../data/girlseq.npy")
rect = [280, 152, 330, 318]
r_list = np.zeros((seq.shape[2]-1,4))

h,w,f = np.shape(seq)
print(h, w, f)
for frame in range(f-1):
        It  = seq[:, :, frame]
        It1 = seq[:, :, frame+1]
        l = LucasKanade(It, It1, rect, threshold, num_iters)
        rect[0] += l[0] #x1
        rect[1] += l[1] #y1
        rect[2] += l[0] #x2
        rect[3] += l[1] #y2

        r_list[frame] = rect
        if (frame % 20 == 0 or frame == 1):

            plt.figure()
            plt.imshow(seq[:,:,frame], cmap='gray')
            rectangle = patches.Rectangle((int(rect[0]), int(rect[1])), (rect[2]-rect[0]),
                                    (rect[3]-rect[1]), fill=False, edgecolor='r', linewidth=2)

            plt.gca().add_patch(rectangle)
            plt.title('frame %d'%frame)
            plt.savefig('girlseqframe' + str(frame) + '.png', bbox_inches='tight')
            plt.show()

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


frame 0
frame 1
frame 20
frame 40
frame 60
frame 80

Problem 1.4

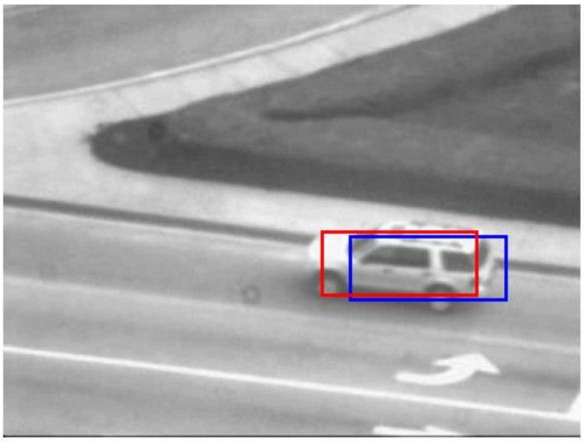
testCarSequenceWithTemplateCorrection.py

```
                    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]
r = rect[:]
rects = rect[:]
h, w, frames = np.shape(seq)
update = True
It = seq[:, :, 0]
p0 = np.zeros(2)
print(frames)
for f in range(frames-1):
    print(f)
    It1 = seq[:,:,f+1]
    p = LucasKanade(It, It1, r, threshold, num_iters, p0)

    pdp = p + [r[0] - rect[0], r[1] - rect[1]] #shifting the p
    p_star = LucasKanade(seq[:, :, 0], It1, rect, threshold, num_iters, pdp)

    change = np.linalg.norm(pdp-p_star)
    if change<threshold:
        p_2= (p_star - [r[0] - rect[0], r[1] - rect[1]])
        r[0] += p_2[0]
        r[2] += p_2[0]
        r[1] += p_2[1]
        r[3] += p_2[1]
        It = seq[:, :, f+1]
        rects = np.vstack((rects, r))
        p0 = np.zeros(2)
    else:
        rects = np.vstack((rects, [r[0]+p[0], r[1]+p[1], r[2]+p[0], r[3]+p[1]]))
        p0 = p

np.save('carseqrects-wcrt.npy', rects)
carseqrects = np.load('carseqrects.npy')
carseqrects_ct = np.load('carseqrects-wcrt.npy')
frame_req= [1, 100, 200, 300, 400]

for index in range(len(frame_req)):
    i = frame_req[index]
    fig = plt.figure()
    frame = seq[:,:,i]
    rect_nc = carseqrects[i,:]
    rect_ct = carseqrects_ct[i,:]
    plt.imshow(frame, cmap='gray')
    plt.axis('off')
    patch1 = patches.Rectangle((rect_nc[0],rect_nc[1]), (rect_nc[2]-rect_nc[0]),
                            (rect_nc[3]-rect_nc[1]), edgecolor = 'b', facecolor='none', linewidth=2)
    patch2 = patches.Rectangle((rect_ct[0],rect_ct[1]), (rect_ct[2]-rect_ct[0]),
                            (rect_ct[3]-rect_ct[1]), edgecolor = 'r', facecolor='none', linewidth=2)
    ax = plt.gca()
    ax.add_patch(patch1)
    ax.add_patch(patch2)
    fig.savefig('carseq-wcrtframe' + str(i) + '.png', bbox_inches='tight')
```

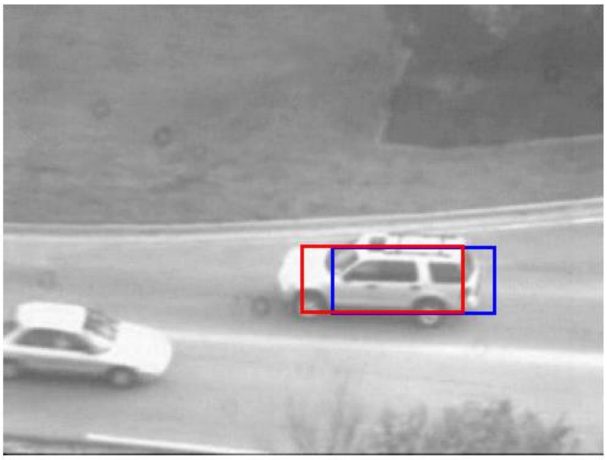


Frame 1

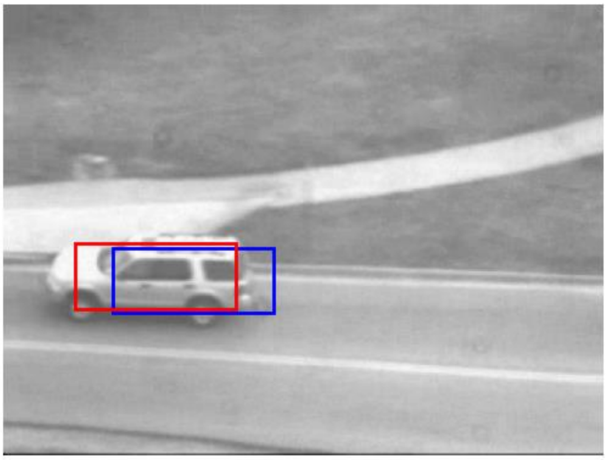
Frame 100



Frame 200



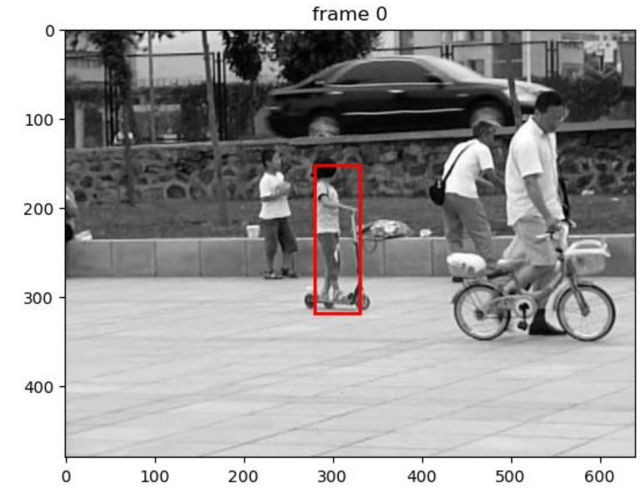
Frame 300

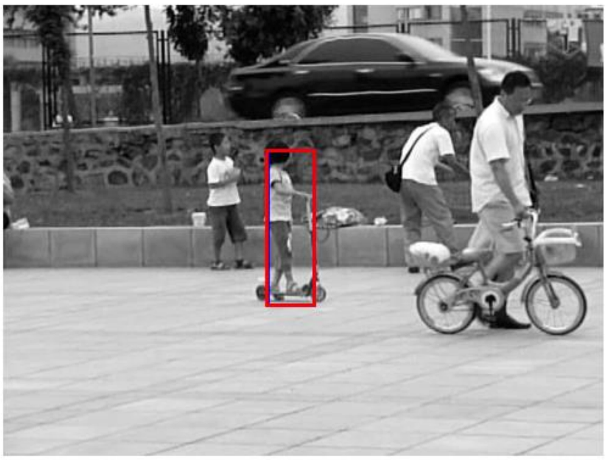


Frame 400

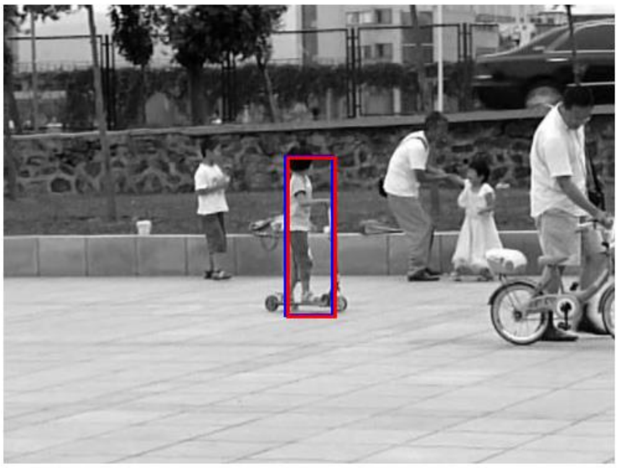
testGirlSequenceWithTemplateCorrection.py

```
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]
r = rect[:]
rects = rect[:]
h, w, frames = np.shape(seq)
update = True
It = seq[:, :, 0]
p0 = np.zeros(2)
print(frames)
for f in range(frames-1):
    print(f)
    It1 = seq[:,:,f+1]
    p = LucasKanade(It, It1, r, threshold, num_iters, p0)
    pdp = p + [r[0] - rect[0], r[1] - rect[1]] #shifting the p
    p_star = LucasKanade(seq[:, :, 0], It1, rect, threshold, num_iters, pdp)
    change = np.linalg.norm(pdp-p_star)
    if change<threshold:
        p_2= (p_star - [r[0] - rect[0], r[1] - rect[1]])
        r[0] += p_2[0]
        r[2] += p_2[0]
        r[1] += p_2[1]
        r[3] += p_2[1]
        It = seq[:, :, f+1]
        rects = np.vstack((rects, r))
        p0 = np.zeros(2)
    else:
        rects = np.vstack((rects, [r[0]+p[0], r[1]+p[1], r[2]+p[0], r[3]+p[1]]))
        p0 = p

np.save('girlseqrects-wcrt.npy', rects)
carseqrects = np.load('girlseqrects.npy')
carseqrects_ct = np.load('girlseqrects-wcrt.npy')
frame_req= [1, 20, 40, 60, 80]

for index in range(len(frame_req)):
    i = frame_req[index]
    fig = plt.figure()
    frame = seq[:,:,i]
    rect_nc = carseqrects[i,:]
    rect_ct = carseqrects_ct[i,:]
    plt.imshow(frame, cmap='gray')
    plt.axis('off')
    patch1 = patches.Rectangle((rect_nc[0],rect_nc[1]), (rect_nc[2]-rect_nc[0]),
                               (rect_nc[3]-rect_nc[1]), edgecolor = 'b', facecolor='none', linewidth=2)
    patch2 = patches.Rectangle((rect_ct[0],rect_ct[1]), (rect_ct[2]-rect_ct[0]),
                               (rect_ct[3]-rect_ct[1]), edgecolor = 'r', facecolor='none', linewidth=2)
    ax = plt.gca()
    ax.add_patch(patch1)
    ax.add_patch(patch2)
    fig.savefig('girlseq-wcrtframe' + str(i) + '.png', bbox_inches='tight')
```

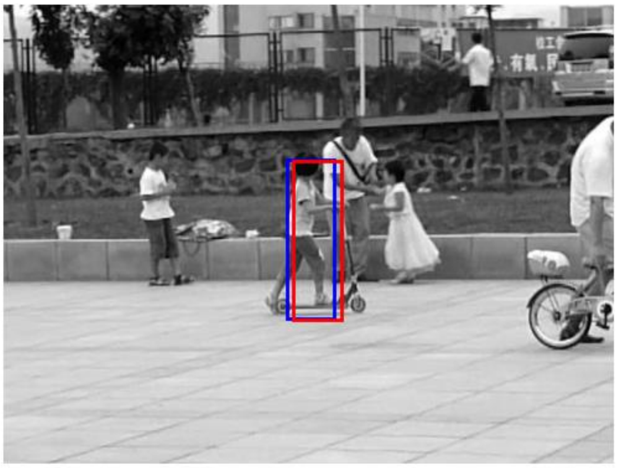




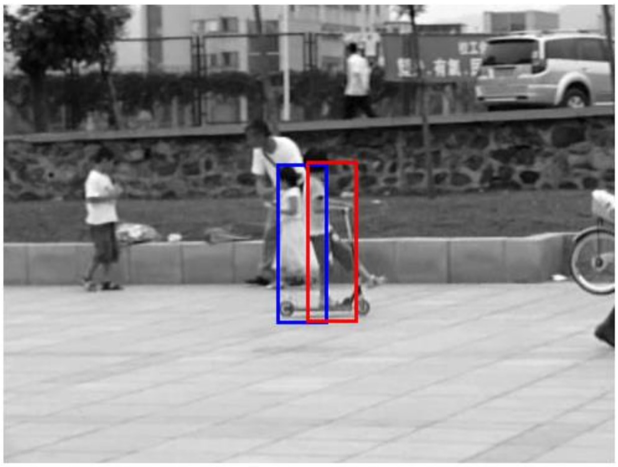
Frame 1



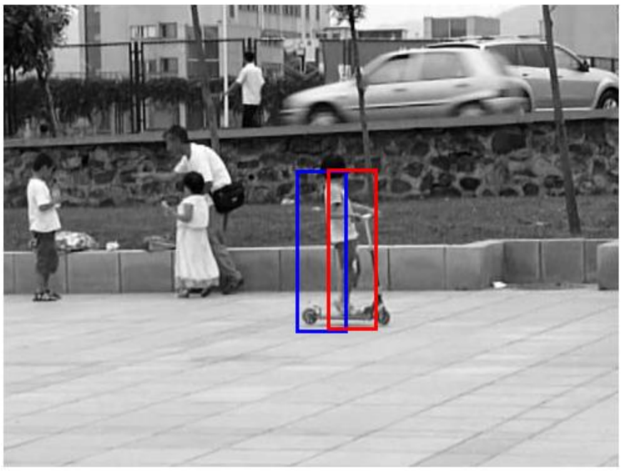
Frame 20



Frame 40



Frame 60



Frame 80

16720 – hw3
mrohitth

Problem 2.1

LucasKanadeAffine.py

```
def LucasKanadeAffine(It, It1, threshold, num_iters):
    r1, c1 = It.shape
    r2, c2 = It.shape

    splinet  = RectBivariateSpline(np.linspace(0, r1, r1), np.linspace(0, c1, c1), It)
    splinet1 = RectBivariateSpline(np.linspace(0, r2, r2), np.linspace(0, c2, c2), It1)

    Iy, Ix = np.gradient(It1) # Affine subtraction
    spline_x = RectBivariateSpline(np.linspace(0, r2, r2), np.linspace(0, c2, c2), Ix)
    spline_y = RectBivariateSpline(np.linspace(0, r2, r2), np.linspace(0, c2, c2), Iy)

    M = np.eye(3)

    #coordinates for the template image
    x, y = np.mgrid[0:c1, 0:r1]

    x_c = np.reshape(x, (1, -1))
    y_c = np.reshape(y, (1, -1))

    #[x, y, 1]
    coor = np.vstack((x_c, y_c, np.ones((1, r1*c1))))
    p  = np.zeros(6)
    dp = np.ones(6) #six parameters to be determined
    n=1

    while(np.square(dp).sum()>threshold and n<num_iters):
        M=np.array([[1+p[0], p[1], p[2]], [p[3], 1+p[4], p[5]], [0, 0, 1]])
        warp = M@coor #3*N

        #xp and yp coordinates
        warp_x = warp[0]
        warp_y = warp[1]

        #gradient splines
        grad_x = spline_x.ev(warp_y, warp_x).flatten()
        grad_y = spline_y.ev(warp_y, warp_x).flatten()

        warp_final = splinet1.ev(warp_y,warp_x).flatten()
        T = splinet.ev(y, x).flatten()

        #error image
        error = np.reshape(T-warp_final, (len(warp_x), 1))

        A1=np.multiply(grad_x, x_c)
        A2=np.multiply(grad_x, y_c)
        A3=np.reshape(grad_x, (1,-1))
        A4=np.multiply(grad_y, x_c)
        A5=np.multiply(grad_y, y_c)
        A6=np.reshape(grad_y, (1,-1))
        A = np.vstack((A1, A2, A3, A4, A5, A6)) #this is the Jaconian and the gradient of I
        A=A.T

        H = A.T@A#We calculate the Hessian

        dp = np.linalg.inv(H) @ A.T @ error
        p = (p + dp.T).ravel()
        n+=1

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

Problem 2.2

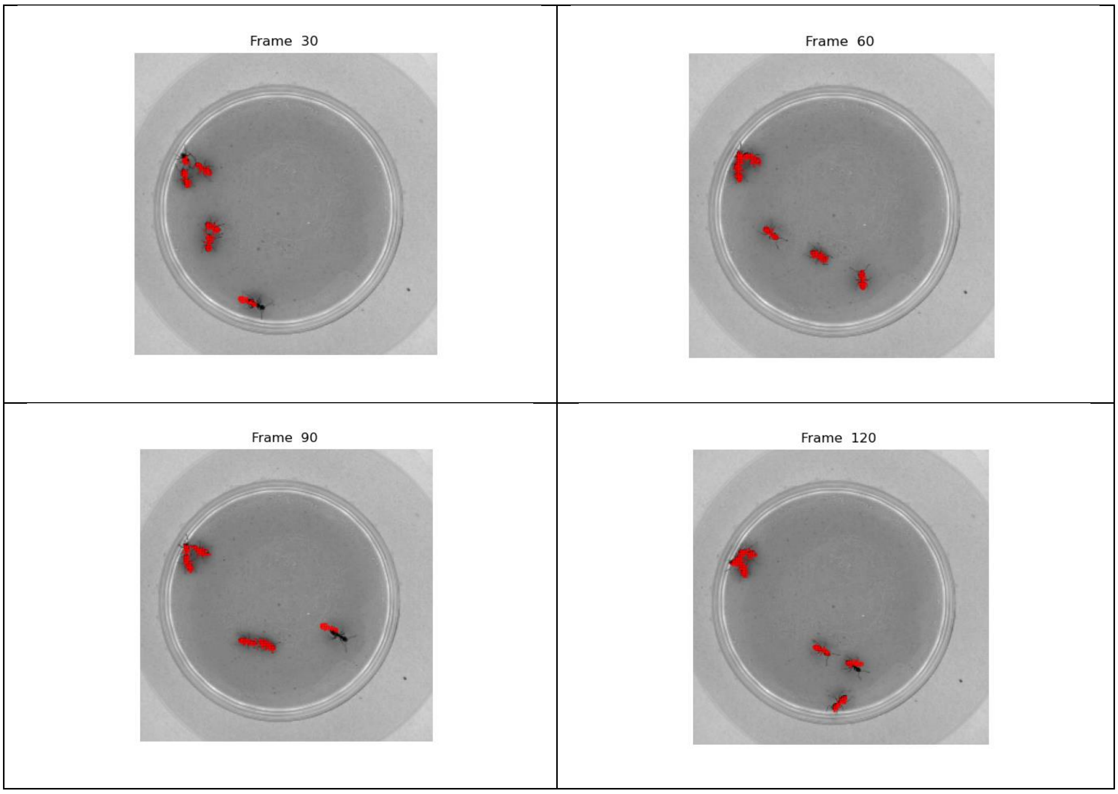
SubtractDominantMotion.py

```python
def SubtractDominantMotion(image1, image2, threshold, num_iters, tolerance):

    mask = np.zeros(image1.shape, dtype=bool)
    '''
    compostition affine
    '''
    M = LucasKanadeAffine(image1, image2, threshold, num_iters)
    '''
    inverse compostition affine
    '''
    # M = InverseCompositionAffine(image1, image2, threshold, num_iters)

    image2_warp=cv2.warpAffine(image2,M[:2],image1.T.shape)

    image2_erode    = binary_erosion(image2_warp)
    image2_dilation = binary_dilation(image2_erode)

    diff = np.abs(image1-image2_dilation)

    mask = (diff>tolerance)

    # print(mask)
    return mask
```

Problem 2.3

testAntSequence.py

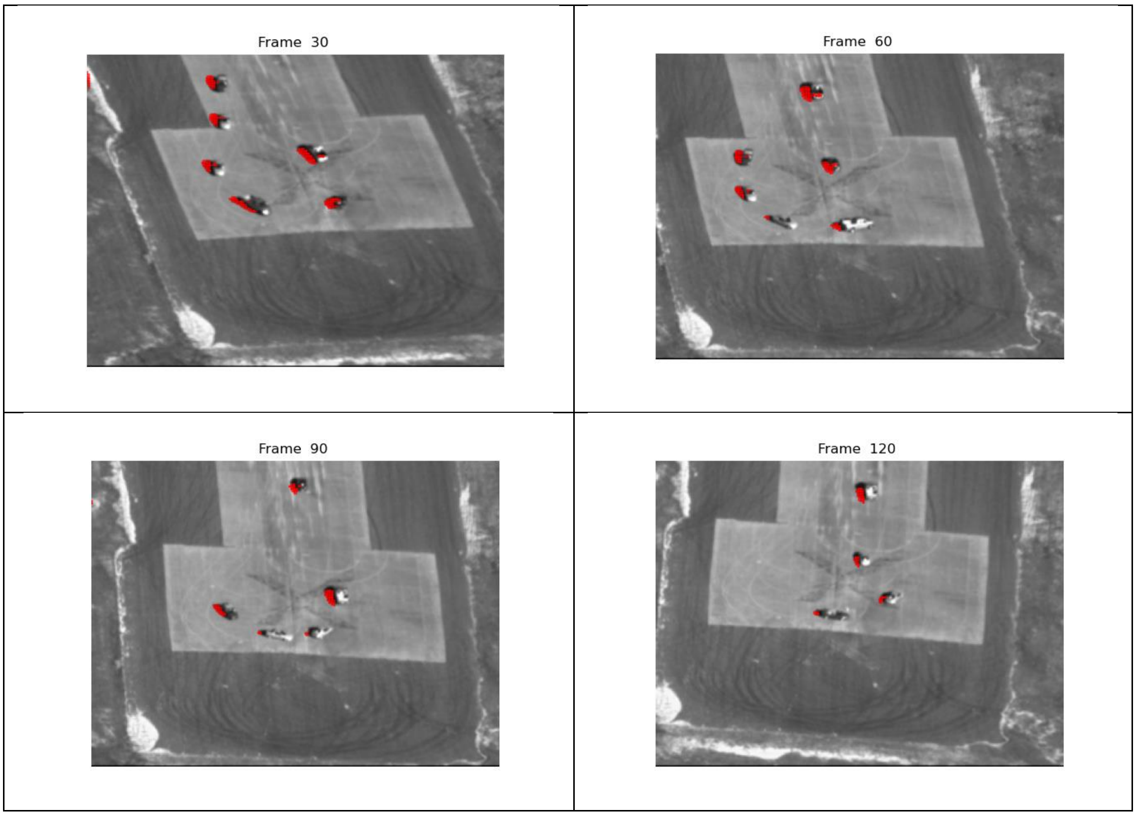
```python
parser = argparse.ArgumentParser()
parser.add_argument('--num_iters', type=int, default=1e3, help='number of iterations of Lucas-Kanade')
parser.add_argument('--threshold', type=float, default=1e-2,
                    help='dp threshold of Lucas-Kanade for terminating optimization')
parser.add_argument('--tolerance', type=float, default=0.75,
                    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')
imH,imW,frames = np.shape(seq)

start=time.time()
for i in range(frames-1):
    image1 = seq[:,:,i]
    image2 = seq[:,:,i+1]
    mask = SubtractDominantMotion.SubtractDominantMotion(image1,image2,threshold, num_iters, tolerance)

    if (i == 29) or (i == 59) or (i == 89) or (i ==119):
        pic = plt.figure()
        plt.imshow(image2, cmap='gray')
        plt.axis('off')
        plt.title("Frame  %d "%(i+1))
        for w in range(mask.shape[0]-1):
            for h in range(mask.shape[1]-1):
                if mask[w,h]:
                    plt.scatter(h, w, s = 1, c = 'r', alpha=0.5)
        plt.show()

stop=time.time()
print("Total time taken:",stop-start)
```



16720 – hw3
mrohitth

testArialSequence.py

```
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=0.75,
                    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')

imH,imW,frames = np.shape(seq)

start=time.time()
for i in range(frames-1):

    image1 = seq[:,:,i]
    image2 = seq[:,:,i+1]
    mask = SubtractDominantMotion.SubtractDominantMotion(image1,image2,threshold, num_iters, tolerance)

    if (i == 29) or (i == 59) or (i == 89) or (i ==119):

        pic = plt.figure()
        plt.imshow(image2, cmap='gray')
        plt.axis('off')
        plt.title("Frame  %d "%(i+1))

        for w in range(mask.shape[0]-1):
            for h in range(mask.shape[1]-1):
                if mask[w,h]:
                    plt.scatter(h, w, s = 1, c = 'r', alpha=0.5)

        plt.show()

stop=time.time()
print("Total time taken:",stop-start)
```



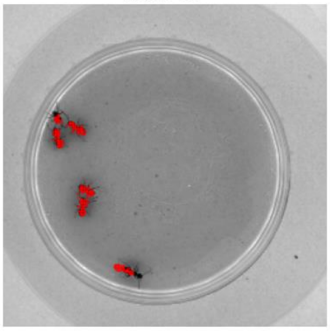
16720 – hw3
mrohitth

Problem 3.1

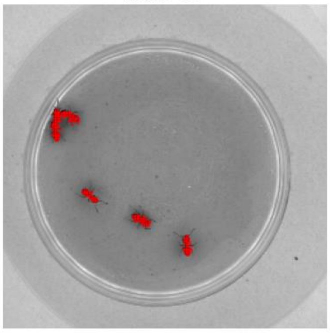
InverseCompositionAffine.py

```
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]
    """
    p  = np.zeros(6)
    dp = np.ones(6) #six parameters to be determined
    M = np.eye(3)
    r1, c1 = It.shape
    r2, c2 = It.shape

    splinet  = RectBivariateSpline(np.linspace(0, r1, r1), np.linspace(0, c1, c1), It)
    splinet1 = RectBivariateSpline(np.linspace(0, r2, r2), np.linspace(0, c2, c2), It1)

    Iy, Ix = np.gradient(It)    #Affine subtraction
    spline_x = RectBivariateSpline(np.linspace(0, r1, r1),np.linspace(0, c1, c1), Ix)
    spline_y = RectBivariateSpline(np.linspace(0, r1, r1),np.linspace(0, c1, c1), Iy)

    x, y = np.mgrid[0:c1, 0:r1]
    x_c = np.reshape(x, (1, -1))
    y_c = np.reshape(y, (1, -1))

    #[x, y, 1]
    coor = np.vstack((x_c, y_c, np.ones((1, r1*c1))))

    grad_x = spline_x.ev(y, x).flatten()
    grad_y = spline_y.ev(y, x).flatten()

    T = splinet.ev(y, x).flatten()

    A1 = np.multiply(grad_x, x_c)
    A2 = np.multiply(grad_x, y_c)
    A3 = np.reshape(grad_x, (1, -1))
    A4 = np.multiply(grad_y, x_c)
    A5 = np.multiply(grad_y, y_c)
    A6 = np.reshape(grad_y, (1, -1))
    A = np.vstack((A1, A2, A3, A4, A5, A6)) #Jaconian and the gradient of I
    A = A.T
    H = A.T@A #Hessian
    n = 1

    while(np.square(dp).sum()>threshold and n<num_iters):
        M = np.array([[1+p[0], p[1], p[2]], [p[3], 1+p[4], p[5]], [0, 0, 1]])
        warp = M@coor

        #xp and yp coordinates
        warp_x = warp[0]
        warp_y = warp[1]

        # gradient splines
        warp_final = splinet1.ev(warp_y, warp_x).flatten()

        #error image
        error = np.reshape(T-warp_final, (len(warp_x), 1))

        dp = np.linalg.inv(H) @ A.T @ error
        p = (p + dp.T).ravel()
        n+=1

        dM = np.vstack((dp.reshape(2, 3), [0, 0, 1]))
        M = M @ np.linalg.inv(dM)
```

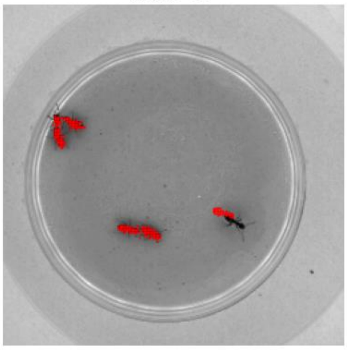
Frame 30



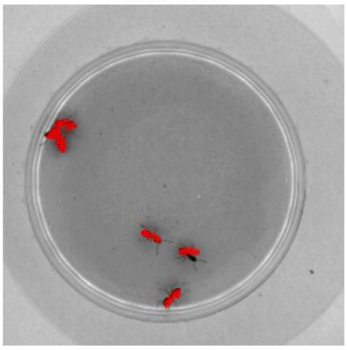
Frame 60



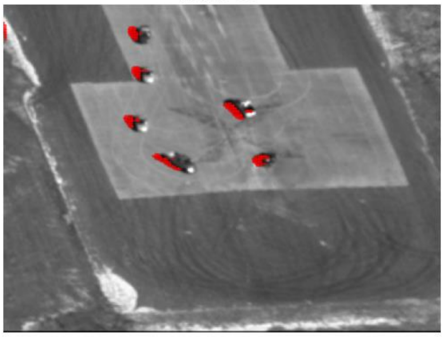
Frame 90



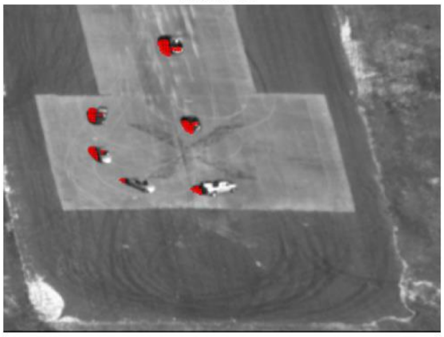
Frame 120



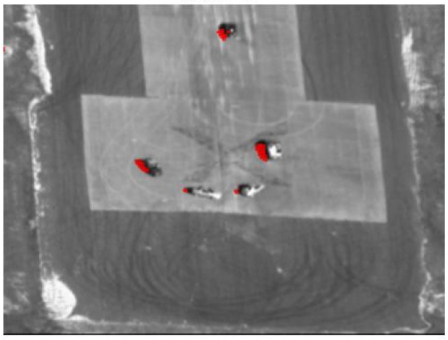
Frame 30



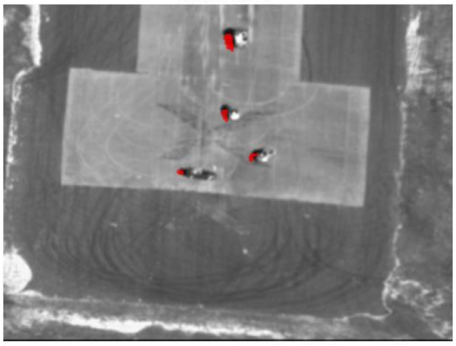
Frame 60



Frame 90



Frame 120



In the classical approach we need to update A and b in every iteration until Δp converges. A being a very large matrix D X 6 matrix, it requires more time for the convergence of Δp. However, with inverse compositional approach, A' and $(A'^T A)^{-1} A'^T$ can be precomputed only once, and then it can be multiplied to updated b until Δp converges, which saves a huge amount of computational time and costs.

```
PS C:\Users\mathe\OneDrive\Desktop\CV\hw3\code> python .\testAntSequence.py
Total time taken: 22.205109119415283
PS C:\Users\mathe\OneDrive\Desktop\CV\hw3\code> python .\testAerialSequence.py
Total time taken: 62.39746308326721
PS C:\Users\mathe\OneDrive\Desktop\CV\hw3\code>
PS C:\Users\mathe\OneDrive\Desktop\CV\hw3\code>
PS C:\Users\mathe\OneDrive\Desktop\CV\hw3\code> python .\testAntSequence.py
Total time taken: 17.871172666549683
PS C:\Users\mathe\OneDrive\Desktop\CV\hw3\code> python .\testAerialSequence.py
Total time taken: 31.759002447128296
PS C:\Users\mathe\OneDrive\Desktop\CV\hw3\code>
```

Here, the latter 2 computations are using the InverseCompositionAffine, while the former two correspond to without using Inverse compositional approach. We can clearly see the jump in performance by looking at the time each program took to compute.