

Problem 1.1.1

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

Here, $\mathbf{p} = [p_1, p_2]^T$

$\mathbf{x}' = W(\mathbf{x}; \mathbf{p}) = \mathbf{x} + \mathbf{p}$

Hence, $W(\mathbf{x}; \mathbf{p})$ has two parameters p_1 and p_2

$$W(\mathbf{x}; \mathbf{p}) = \begin{bmatrix} 1 & 0 & p_1 \\ 0 & 1 & p_2 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + p_1 + 0 \\ 0 + p_2 + y \end{bmatrix} = \begin{bmatrix} x + p_1 \\ y + p_2 \end{bmatrix}$$

$$\text{So, } \partial W / \partial \mathbf{p}^T = \begin{bmatrix} \frac{\partial W_x}{\partial p_1} & \frac{\partial W_x}{\partial p_2} \\ \frac{\partial W_y}{\partial p_1} & \frac{\partial W_y}{\partial p_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

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.

$$\begin{aligned} & \arg \min_{\Delta \mathbf{p}} \sum_{\mathbf{x} \in \mathbb{N}} \|I_{t+1}(\mathbf{x}' + \Delta \mathbf{p}) - I_t(\mathbf{x})\|_2^2 \\ & \text{where } I_{t+1}(\mathbf{x}' + \Delta \mathbf{p}) \approx I_{t+1}(\mathbf{x}') + \frac{\partial I_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial W(\mathbf{x}; \mathbf{p})}{\partial \mathbf{p}^T} \Delta \mathbf{p} \\ & \arg \min_{\Delta \mathbf{p}} \sum_{\mathbf{x} \in \mathbb{N}} \left\| I_{t+1}(\mathbf{x}') + \frac{\partial I_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial W(\mathbf{x}; \mathbf{p})}{\partial \mathbf{p}^T} \Delta \mathbf{p} - I_t(\mathbf{x}) \right\|_2^2 \quad (2) \\ & \arg \min_{\Delta \mathbf{p}} \sum_{\mathbf{x} \in \mathbb{N}} \left\| \frac{\partial I_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial W(\mathbf{x}; \mathbf{p})}{\partial \mathbf{p}^T} \Delta \mathbf{p} - (I_t(\mathbf{x}) - I_{t+1}(\mathbf{x}')) \right\|_2^2 \\ & \arg \min_{\Delta \mathbf{p}} \| \mathbf{A} \Delta \mathbf{p} - \mathbf{b} \|_2^2 \end{aligned}$$

On comparing the last two above forms we get that:

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

$$\mathbf{b} = \sum_{\mathbf{x} \in \mathbb{N}} I_t(\mathbf{x}) - I_{t+1}(\mathbf{x}')$$

\mathbf{A} represents the steepest descent images and \mathbf{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 , $\mathbf{A}^\top \mathbf{A}$ must be **invertible** or **non-singular matrix** or must **have a non-zero determinant** to obtain a unique solution for Δp .

Problem 1.2

```
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)

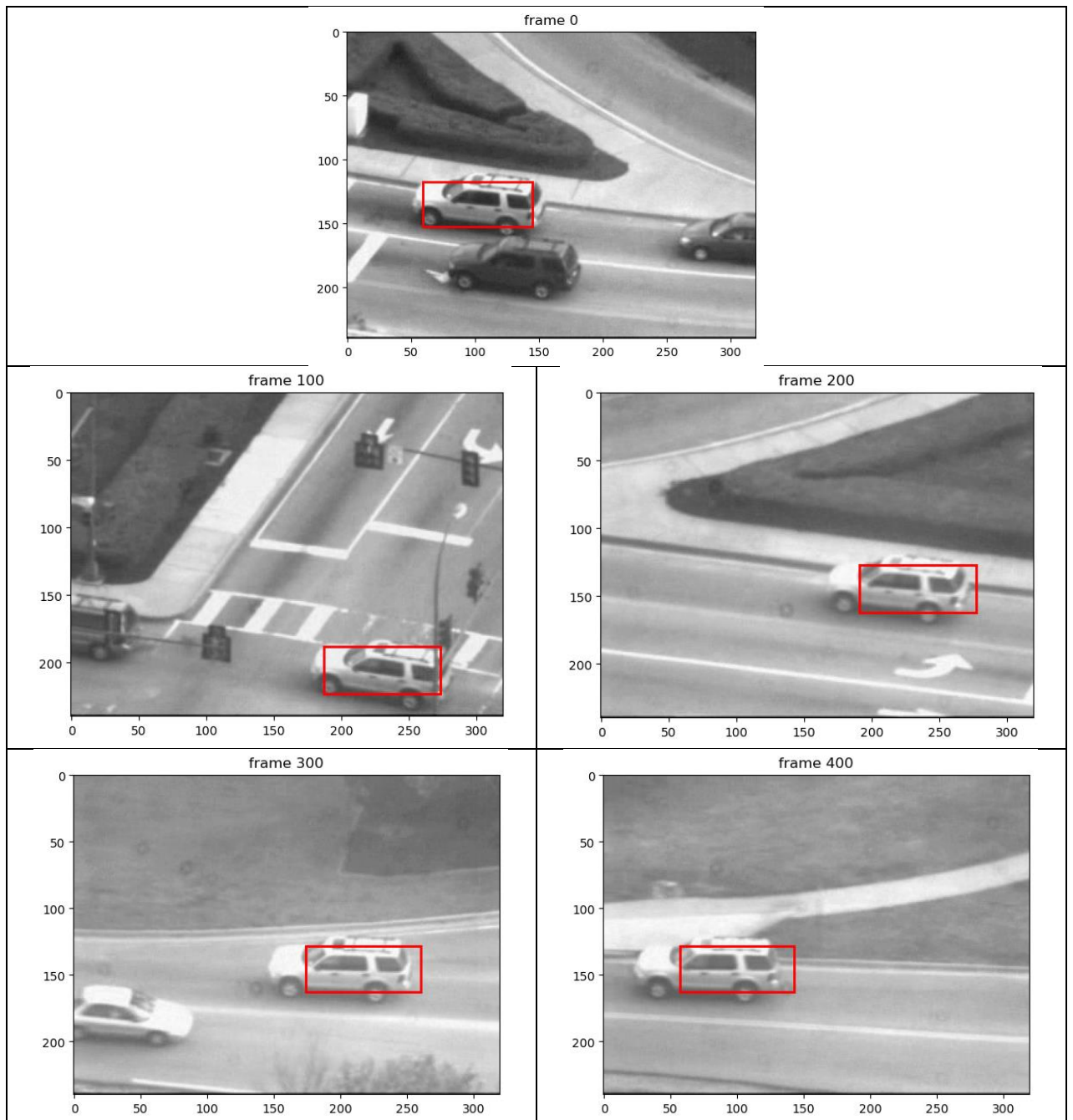
    c = 1
    k = 1
    x, y = np.mgrid[x1:x2+1:w*1j, y1:y2+1:h*1j]
    # print(x, y)
    while (c > threshold and k < num_iters):
        # print(k, c)
        dxp = s1.ev(y+p[1], x+p[0], dy = 1).flatten()
        dyp = s1.ev(y+p[1], x+p[0], dx = 1).flatten()
        It1p = s1.ev(y+p[1], x+p[0]).flatten()
        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
```

Problem 1.3

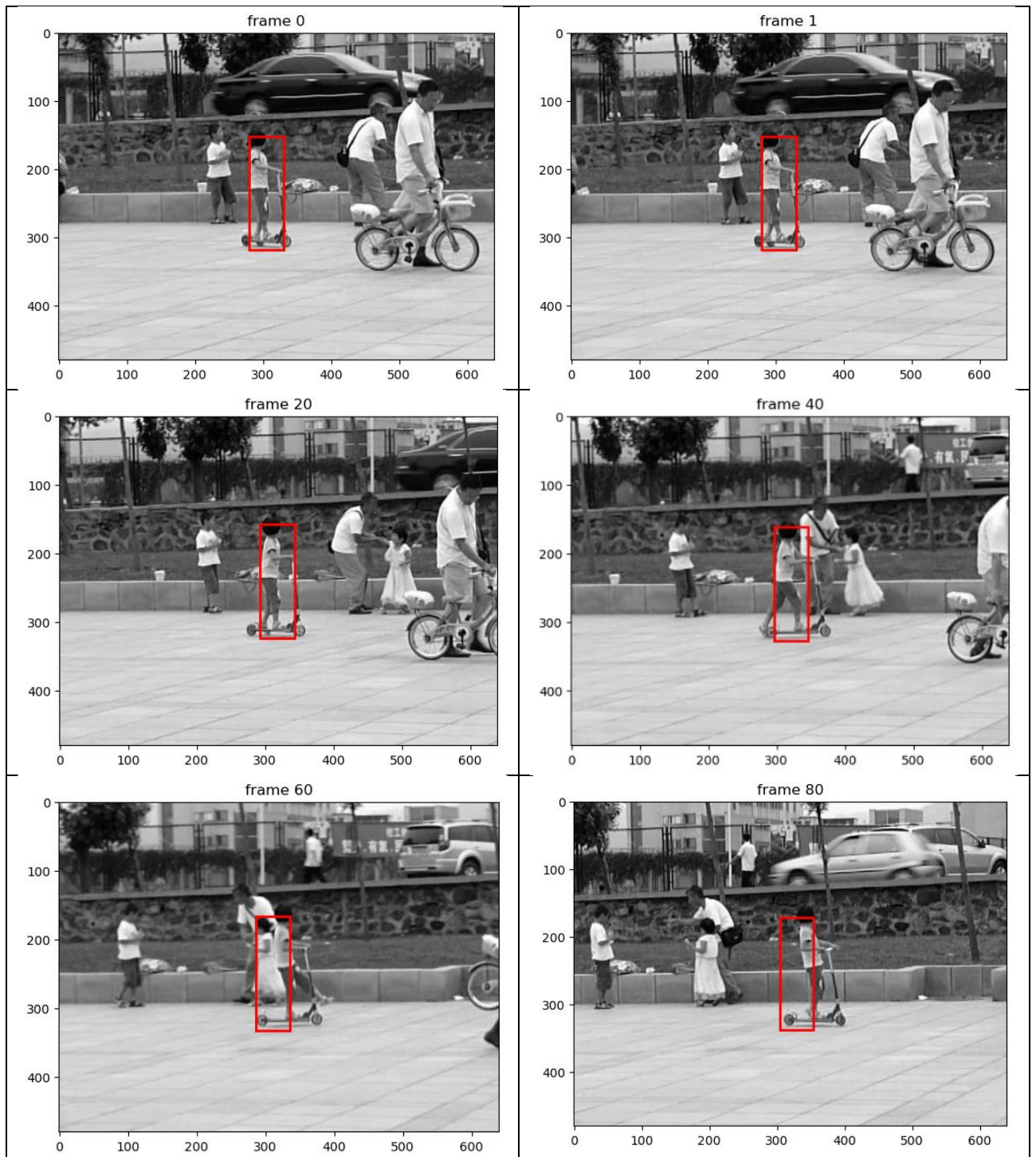
testCarSequence.py

```
9  parser = argparse.ArgumentParser()
10  parser.add_argument('--num_iters', type=int, default=1e4,
11  |                  |                  |                  |                  |
11  |                  |                  |                  |                  | help='number of iterations of Lucas-Kanade')
12  parser.add_argument('--threshold', type=float, default=1e-2,
13  |                  |                  |                  |                  |
13  |                  |                  |                  |                  | help='dp threshold of Lucas-Kanade for terminating optimization')
14  args = parser.parse_args()
15  num_iters = args.num_iters
16  threshold = args.threshold
17
18  seq = np.load("../data/carseq.npy")
19  rect = [59, 116, 145, 151]
20  r_list = np.zeros((seq.shape[2]-1,4))
21
22  h,w,f = np.shape(seq)
23  for frame in range(f-1):
24  |      |      |      |      |
24  |      |      |      |      | It = seq[:, :, frame]
25  |      |      |      |      | It1 = seq[:, :, frame+1]
26  |      |      |      |      | l = LucasKanade(It, It1, rect, threshold, num_iters)
27  |      |      |      |      | rect[0] += l[0] #x1
28  |      |      |      |      | rect[1] += l[1] #y1
29  |      |      |      |      | rect[2] += l[0] #x2
30  |      |      |      |      | rect[3] += l[1] #y2
31
32  |      |      |      |      | r_list[frame] = rect
33
34  |      |      |      |      | if (frame % 100 == 0 or frame == 0):
35  |      |      |      |      |
36  |      |      |      |      |     plt.figure()
37  |      |      |      |      |     plt.imshow(seq[:, :, frame], cmap='gray')
38  |      |      |      |      |     rectangle = patches.Rectangle((int(rect[0]), int(rect[1])), (rect[2]-rect[0]),
39  |      |      |      |      |     |                  |                  |                  |                  |
39  |      |      |      |      |     |                  |                  |                  |                  | (rect[3]-rect[1]), fill=False, edgecolor='r', linewidth=2)
40
41  |      |      |      |      |     plt.gca().add_patch(rectangle)
42  |      |      |      |      |     plt.title('frame %d'%frame)
43  |      |      |      |      |     plt.savefig('carseqframe' + str(frame) + '.png', bbox_inches='tight')
44  |      |      |      |      |     plt.show()
45
46  np.save('carseqrects.npy', r_list)
```



testGirlSequence.py

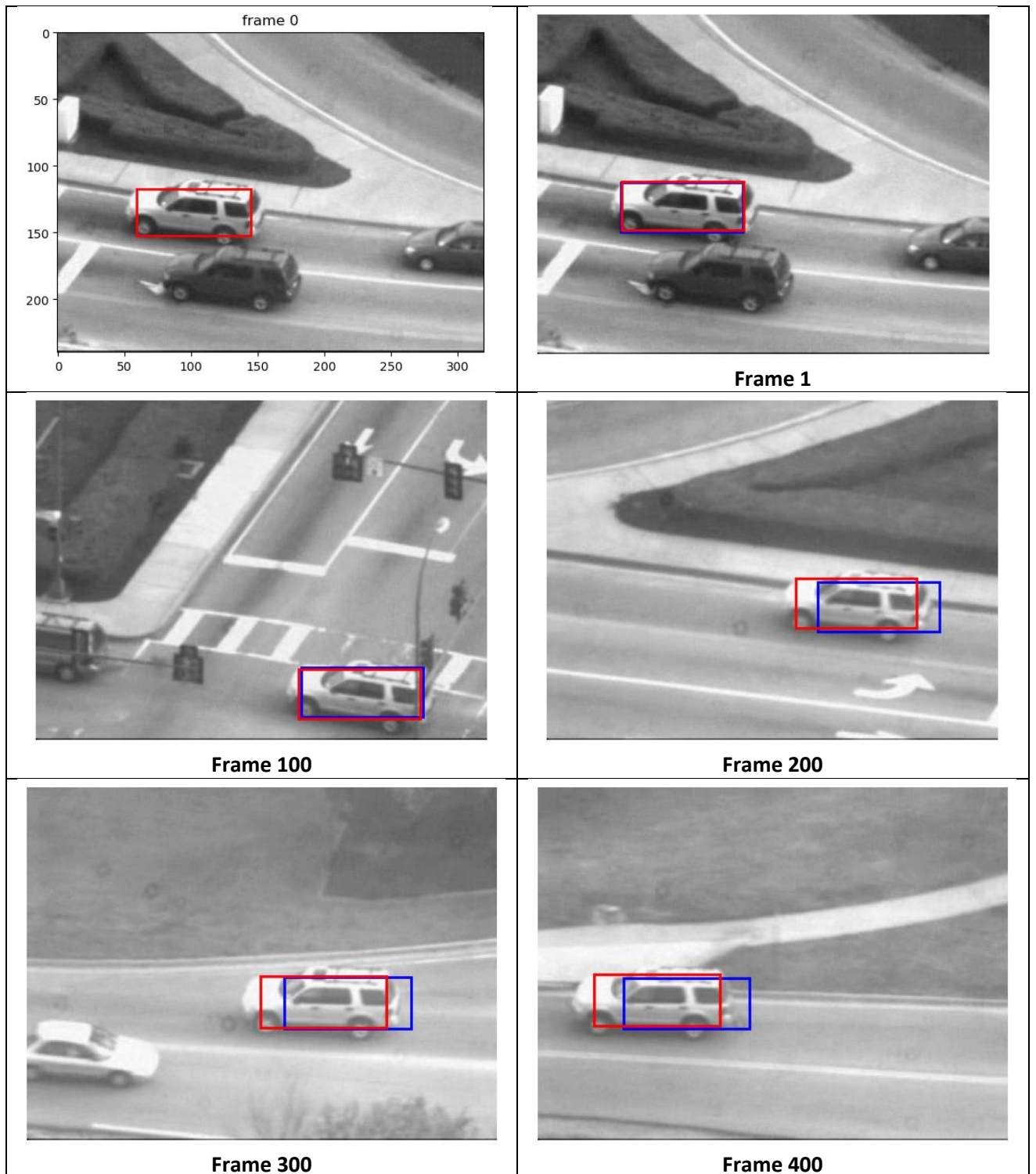
```
9  parser = argparse.ArgumentParser()
10  parser.add_argument('--num_iters', type=int, default=1e4,
11  |                  help='number of iterations of Lucas-Kanade')
12  parser.add_argument('--threshold', type=float, default=1e-2,
13  |                  help='dp threshold of Lucas-Kanade for terminating optimization')
14  args = parser.parse_args()
15  num_iters = args.num_iters
16  threshold = args.threshold
17
18  seq = np.load("../data/girlseq.npy")
19  rect = [280, 152, 330, 318]
20  r_list = np.zeros((seq.shape[2]-1,4))
21
22  h,w,f = np.shape(seq)
23  print(h, w, f)
24  for frame in range(f-1):
25  |      It = seq[:, :, frame]
26  |      It1 = seq[:, :, frame+1]
27  |      l = LucasKanade(It, It1, rect, threshold, num_iters)
28  |      rect[0] += l[0] #x1
29  |      rect[1] += l[1] #y1
30  |      rect[2] += l[0] #x2
31  |      rect[3] += l[1] #y2
32
33  |      r_list[frame] = rect
34  |      if (frame % 20 == 0 or frame == 1):
35  |          plt.figure()
36  |          plt.imshow(seq[:, :, frame], cmap='gray')
37  |          rectangle = patches.Rectangle((int(rect[0]), int(rect[1])), (rect[2]-rect[0]),
38  |          |                  (rect[3]-rect[1]), fill=False, edgecolor='r', linewidth=2)
39  |
40  |          plt.gca().add_patch(rectangle)
41  |          plt.title('frame %d'%frame)
42  |          plt.savefig('girlseqframe' + str(frame) + '.png', bbox_inches='tight')
43  |          plt.show()
44
45
46  np.save('girlseqrects.npy', r_list)
```



Problem 1.4

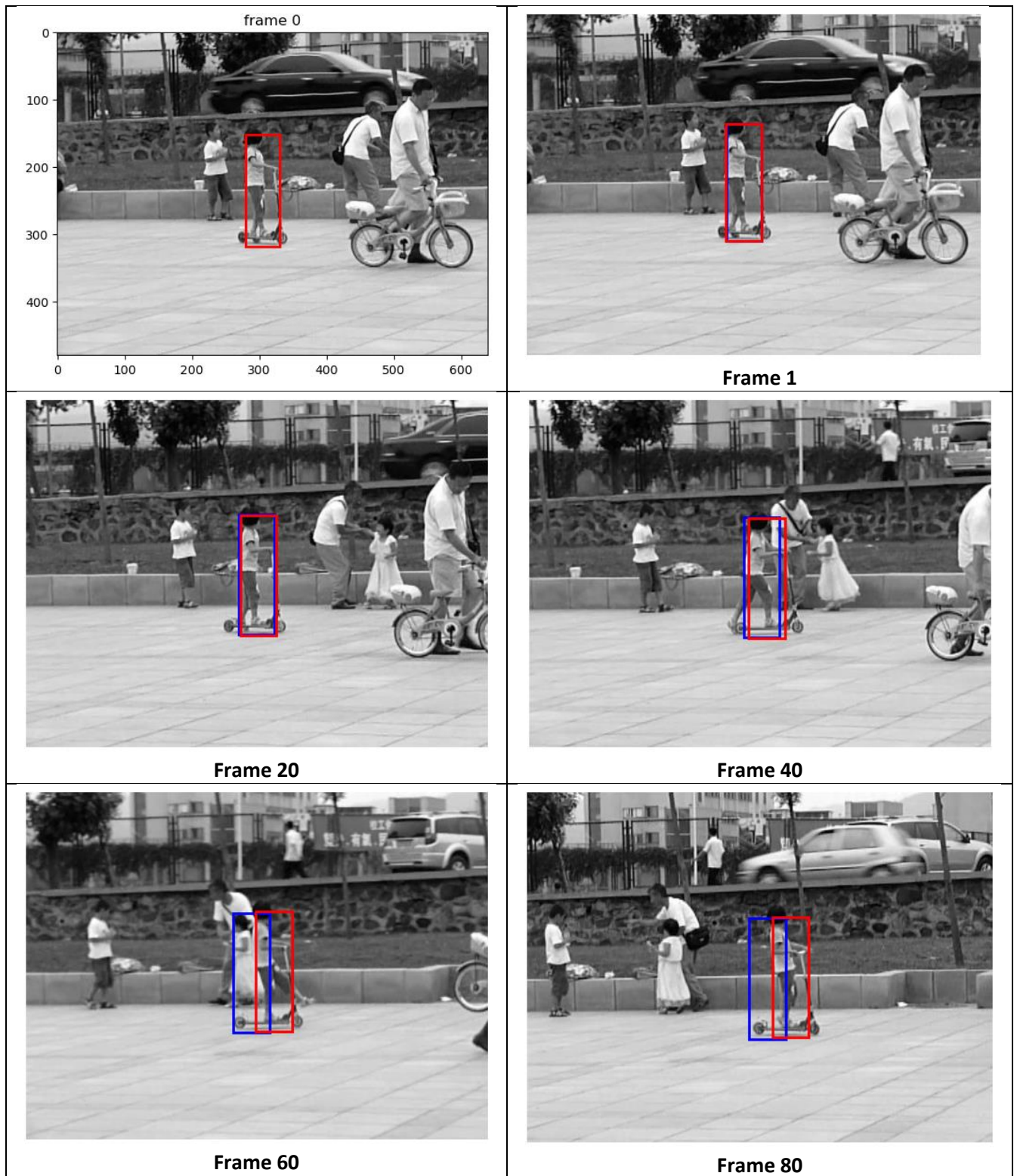
testCarSequenceWithTemplateCorrection.py

```
16 #ncip = threshold for determining whether to update template
17 args = parser.parse_args()
18 num_iters = args.num_iters
19 threshold = args.threshold
20 template_threshold = args.template_threshold
21
22 seq = np.load("../data/carseq.npy")
23 rect = [59, 116, 145, 151]
24 r = rect[:]
25 rects = rect[:]
26 h, w, frames = np.shape(seq)
27 update = True
28 It = seq[:, :, 0]
29 p0 = np.zeros(2)
30 print(frames)
31 for f in range(frames-1):
32     print(f)
33     It1 = seq[:, :, f+1]
34     p = LucasKanade(It, It1, r, threshold, num_iters, p0)
35
36     pdp = p + [r[0] - rect[0], r[1] - rect[1]] #shifting the p
37     p_star = LucasKanade(seq[:, :, 0], It1, rect, threshold, num_iters, pdp)
38
39     change = np.linalg.norm(pdp-p_star)
40     if change<threshold:
41         p_2= (p_star - [r[0] - rect[0], r[1] - rect[1]])
42         r[0] += p_2[0]
43         r[2] += p_2[0]
44         r[1] += p_2[1]
45         r[3] += p_2[1]
46         It = seq[:, :, f+1]
47         rects = np.vstack((rects, r))
48         p0 = np.zeros(2)
49     else:
50         rects = np.vstack((rects, [r[0]+p[0], r[1]+p[1], r[2]+p[0], r[3]+p[1]]))
51         p0 = p
52
53 np.save('carseqrects-wcrt.npy', rects)
54 carseqrects = np.load('carseqrects.npy')
55 carseqrects_ct = np.load('carseqrects-wcrt.npy')
56 frame_req= [1, 100, 200, 300, 400]
57
58 for index in range(len(frame_req)):
59     i = frame_req[index]
60     fig = plt.figure()
61     frame = seq[:, :, i]
62     rect_nc = carseqrects[i, :]
63     rect_ct = carseqrects_ct[i, :]
64     plt.imshow(frame, cmap='gray')
65     plt.axis('off')
66     patch1 = patches.Rectangle((rect_nc[0],rect_nc[1]), (rect_nc[2]-rect_nc[0]),
67                                (rect_nc[3]-rect_nc[1]), edgecolor = 'b', facecolor='none', linewidth=2)
68     patch2 = patches.Rectangle((rect_ct[0],rect_ct[1]), (rect_ct[2]-rect_ct[0]),
69                                (rect_ct[3]-rect_ct[1]), edgecolor = 'r', facecolor='none', linewidth=2)
70     ax = plt.gca()
71     ax.add_patch(patch1)
72     ax.add_patch(patch2)
73     fig.savefig('carseq-wcrtframe' + str(i) + '.png', bbox_inches='tight')
```

testGirlSequenceWithTemplateCorrection.py

```
13 parser.add_argument('--template_threshold', type=float, default=5,  
14                       help='threshold for determining whether to update template')  
15 args = parser.parse_args()  
16 num_iters = args.num_iters  
17 threshold = args.threshold  
18 template_threshold = args.template_threshold  
19  
20 seq = np.load("../data/girlseq.npy")  
21 rect = [280, 152, 330, 318]  
22 r = rect[:]  
23 rects = rect[:]  
24 h, w, frames = np.shape(seq)  
25 update = True  
26 It = seq[:, :, 0]  
27 p0 = np.zeros(2)  
28 print(frames)  
29 for f in range(frames-1):  
30     print(f)  
31     It1 = seq[:, :, f+1]  
32     p = LucasKanade(It, It1, r, threshold, num_iters, p0)  
33     pdp = p + [r[0] - rect[0], r[1] - rect[1]] #shifting the p  
34     p_star = LucasKanade(seq[:, :, 0], It1, rect, threshold, num_iters, pdp)  
35     change = np.linalg.norm(pdp-p_star)  
36     if change<threshold:  
37         p_2 = (p_star - [r[0] - rect[0], r[1] - rect[1]])  
38         r[0] += p_2[0]  
39         r[2] += p_2[0]  
40         r[1] += p_2[1]  
41         r[3] += p_2[1]  
42         It = seq[:, :, f+1]  
43         rects = np.vstack((rects, r))  
44         p0 = np.zeros(2)  
45     else:  
46         rects = np.vstack((rects, [r[0]+p[0], r[1]+p[1], r[2]+p[0], r[3]+p[1]]))  
47         p0 = p  
48  
49 np.save('girlseqrects-wcrt.npy', rects)  
50 carseqrects = np.load('girlseqrects.npy')  
51 carseqrects_ct = np.load('girlseqrects-wcrt.npy')  
52 frame_req= [1, 20, 40, 60, 80]  
53  
54 for index in range(len(frame_req)):  
55     i = frame_req[index]  
56     fig = plt.figure()  
57     frame = seq[:, :, i]  
58     rect_nc = carseqrects[i, :]  
59     rect_ct = carseqrects_ct[i, :]  
60     plt.imshow(frame, cmap='gray')  
61     plt.axis('off')  
62     patch1 = patches.Rectangle((rect_nc[0],rect_nc[1]), (rect_nc[2]-rect_nc[0]),  
63                               (rect_nc[3]-rect_nc[1]), edgecolor = 'b', facecolor='none', linewidth=2)  
64     patch2 = patches.Rectangle((rect_ct[0],rect_ct[1]), (rect_ct[2]-rect_ct[0]),  
65                               (rect_ct[3]-rect_ct[1]), edgecolor = 'r', facecolor='none', linewidth=2)  
66     ax = plt.gca()  
67     ax.add_patch(patch1)  
68     ax.add_patch(patch2)  
69     fig.savefig('girlseq-wcrtframe' + str(i) + '.png', bbox_inches='tight')
```



Problem 2.1

LucasKanadeAffine.py

```
5 def LucasKanadeAffine(It, It1, threshold, num_iters):
6     r1, c1 = It.shape
7     r2, c2 = It1.shape
8
9     splinet = RectBivariateSpline(np.linspace(0, r1, r1), np.linspace(0, c1, c1), It)
10    splinet1 = RectBivariateSpline(np.linspace(0, r2, r2), np.linspace(0, c2, c2), It1)
11
12    Iy, Ix = np.gradient(It1) # Affine subtraction
13    spline_x = RectBivariateSpline(np.linspace(0, r2, r2), np.linspace(0, c2, c2), Ix)
14    spline_y = RectBivariateSpline(np.linspace(0, r2, r2), np.linspace(0, c2, c2), Iy)
15
16    M = np.eye(3)
17
18    #coordinates for the template image
19    x, y = np.mgrid[0:c1, 0:r1]
20
21    x_c = np.reshape(x, (1, -1))
22    y_c = np.reshape(y, (1, -1))
23
24    #[x, y, 1]
25    coor = np.vstack((x_c, y_c, np.ones((1, r1*c1))))
26    p = np.zeros(6)
27    dp = np.ones(6) #six parameters to be determined
28    n=1
29
30    while(np.square(dp).sum()>threshold and n<num_iters):
31        M=np.array([[1+p[0], p[1], p[2]], [p[3], 1+p[4], p[5]], [0, 0, 1]])
32        warp = M@coor #3*N
33
34        #xp and yp coordinates
35        warp_x = warp[0]
36        warp_y = warp[1]
37
38        #gradient splines
39        grad_x = spline_x.ev(warp_y, warp_x).flatten()
40        grad_y = spline_y.ev(warp_y, warp_x).flatten()
41
42        warp_final = splinet1.ev(warp_y, warp_x).flatten()
43        T = splinet.ev(y, x).flatten()
44
45        #error image
46        error = np.reshape(T-warp_final, (len(warp_x), 1))
47
48        A1=np.multiply(grad_x, x_c)
49        A2=np.multiply(grad_x, y_c)
50        A3=np.reshape(grad_x, (1,-1))
51        A4=np.multiply(grad_y, x_c)
52        A5=np.multiply(grad_y, y_c)
53        A6=np.reshape(grad_y, (1,-1))
54        A = np.vstack((A1, A2, A3, A4, A5, A6)) #this is the Jaconian and the gradient of I
55        A=A.T
56
57        H = A.T@A#We calculate the Hessian
58
59        dp = np.linalg.inv(H) @ A.T @ error
60        p = (p + dp.T).ravel()
61        n+=1
62
63    M = np.array([[1+p[0], p[1],p[2]], [p[3], 1+p[4], p[5]], [0, 0, 1]])
64    return M
```

Problem 2.2

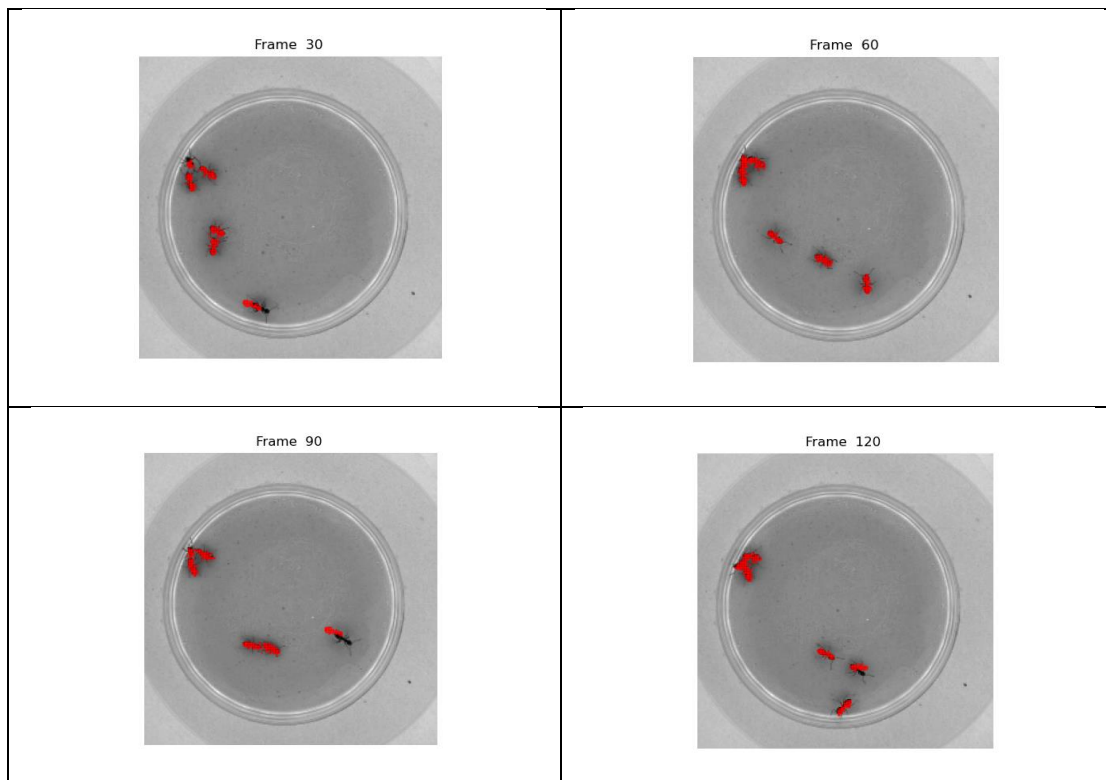
SubtractDominantMotion.py

```
19
20 def SubtractDominantMotion(image1, image2, threshold, num_iters, tolerance):
21
22     mask = np.zeros(image1.shape, dtype=bool)
23     '''
24     composition affine
25     '''
26     M = LucasKanadeAffine(image1, image2, threshold, num_iters)
27     '''
28     inverse composition affine
29     '''
30     # M = InverseCompositionAffine(image1, image2, threshold, num_iters)
31
32     image2_warp=cv2.warpAffine(image2,M[:2],image1.T.shape)
33
34     image2_erode    = binary_erosion(image2_warp)
35     image2_dilation = binary_dilation(image2_erode)
36
37     diff = np.abs(image1-image2_dilation)
38
39     mask = (diff>tolerance)
40
41     # print(mask)
42     return mask
```


Problem 2.3

testAntSequence.py

```
13 parser = argparse.ArgumentParser()
14 parser.add_argument('--num_iters', type=int, default=1e3, help='number of iterations of Lucas-Kanade')
15 parser.add_argument('--threshold', type=float, default=1e-2,
16                     help='dp threshold of Lucas-Kanade for terminating optimization')
17 parser.add_argument('--tolerance', type=float, default=0.75,
18                     help='binary threshold of intensity difference when computing the mask')
19 args = parser.parse_args()
20 num_iters = args.num_iters
21 threshold = args.threshold
22 tolerance = args.tolerance
23
24 seq = np.load('../data/antseq.npy')
25 imH,imW,frames = np.shape(seq)
26
27 start=time.time()
28 for i in range(frames-1):
29     image1 = seq[:, :, i]
30     image2 = seq[:, :, i+1]
31     mask = SubtractDominantMotion.SubtractDominantMotion(image1, image2, threshold, num_iters, tolerance)
32
33     if (i == 29) or (i == 59) or (i == 89) or (i == 119):
34         pic = plt.figure()
35         plt.imshow(image2, cmap='gray')
36         plt.axis('off')
37         plt.title("Frame %d"%(i+1))
38         for w in range(mask.shape[0]-1):
39             for h in range(mask.shape[1]-1):
40                 if mask[w,h]:
41                     plt.scatter(h, w, s = 1, c = 'r', alpha=0.5)
42         plt.show()
43
44 stop=time.time()
45 print("Total time taken:",stop-start)
```

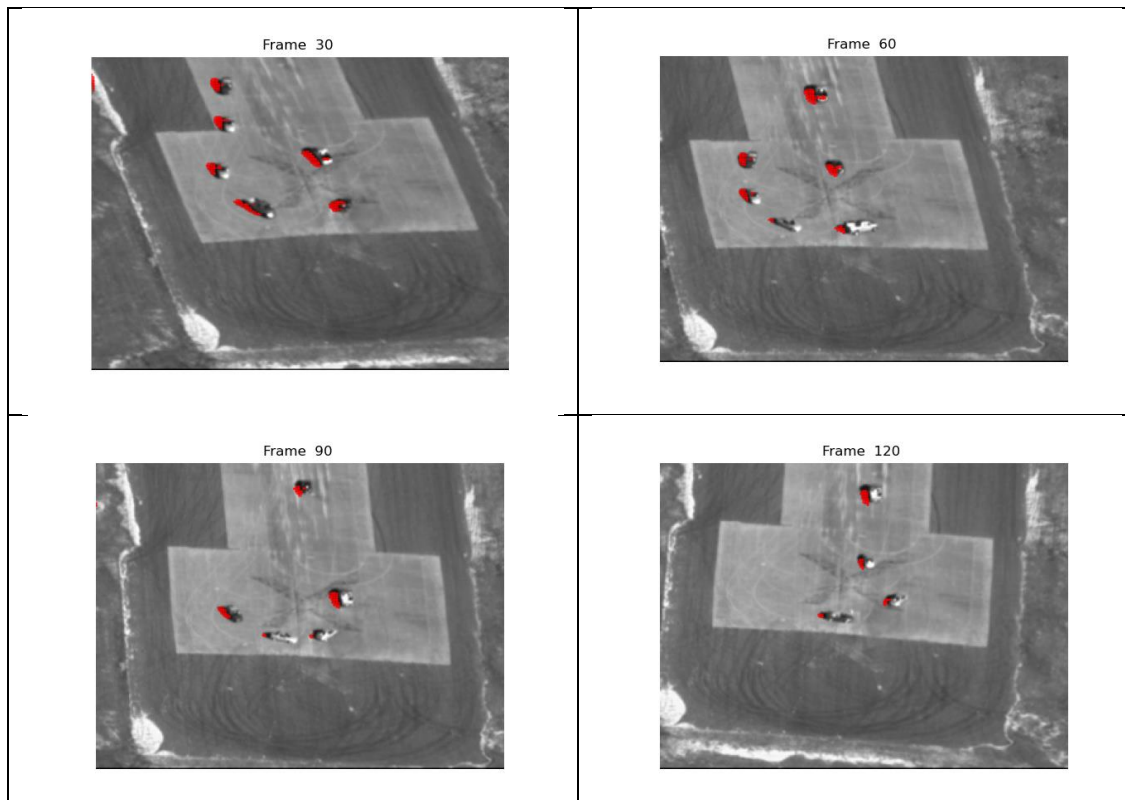


testAerialSequence.py

```

10
11 parser = argparse.ArgumentParser()
12 parser.add_argument('--num_iters', type=int, default=1e3, help='number of iterations of Lucas-Kanade')
13 parser.add_argument('--threshold', type=float, default=1e-2,
14                     help='dp threshold of Lucas-Kanade for terminating optimization')
15 parser.add_argument('--tolerance', type=float, default=0.75,
16                     help='binary threshold of intensity difference when computing the mask')
17 args = parser.parse_args()
18 num_iters = args.num_iters
19 threshold = args.threshold
20 tolerance = args.tolerance
21
22 seq = np.load('../data/aerialseq.npy')
23
24 imH,imW,frames = np.shape(seq)
25
26 start=time.time()
27 for i in range(frames-1):
28
29     image1 = seq[:, :, i]
30     image2 = seq[:, :, i+1]
31     mask = SubtractDominantMotion.SubtractDominantMotion(image1, image2, threshold, num_iters, tolerance)
32
33     if (i == 29) or (i == 59) or (i == 89) or (i == 119):
34
35         pic = plt.figure()
36         plt.imshow(image2, cmap='gray')
37         plt.axis('off')
38         plt.title("Frame %d" % (i+1))
39
40         for w in range(mask.shape[0]-1):
41             for h in range(mask.shape[1]-1):
42                 if mask[w,h]:
43                     plt.scatter(h, w, s = 1, c = 'r', alpha=0.5)
44
45         plt.show()
46
47 stop=time.time()
48 print("Total time taken:", stop-start)

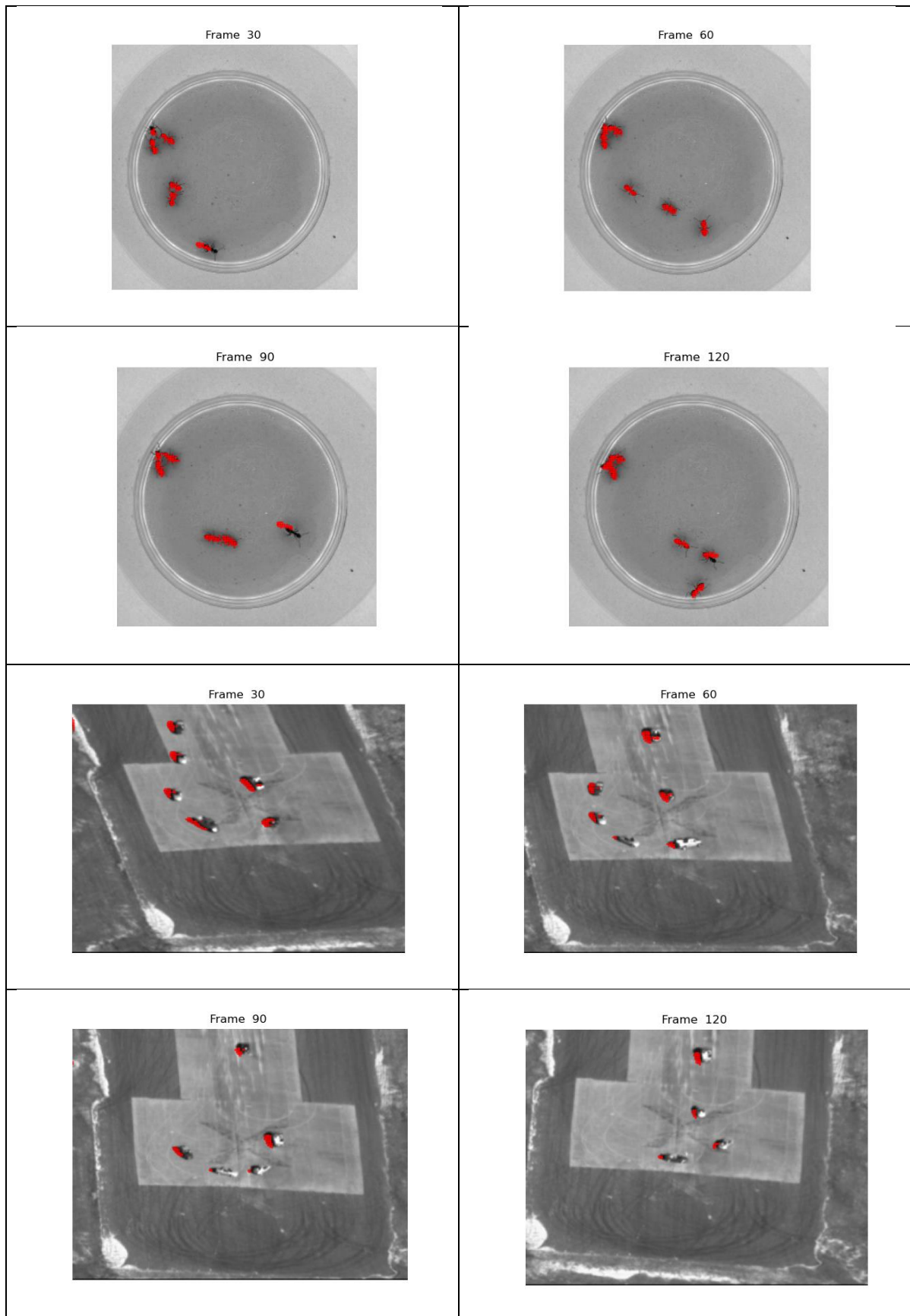
```



Problem 3.1

InverseCompositionAffine.py

```
4 def InverseCompositionAffine(It, It1, threshold, num_iters):
5     """
6     :param It: template image
7     :param It1: Current image
8     :param threshold: if the length of dp is smaller than the threshold, terminate the optimization
9     :param num_iters: number of iterations of the optimization
10    :return: M: the Affine warp matrix [2x3 numpy array]
11    """
12    p = np.zeros(6)
13    dp = np.ones(6) #six parameters to be determined
14    M = np.eye(3)
15    r1, c1 = It.shape
16    r2, c2 = It1.shape
17
18    splinet = RectBivariateSpline(np.linspace(0, r1, r1), np.linspace(0, c1, c1), It)
19    splinet1 = RectBivariateSpline(np.linspace(0, r2, r2), np.linspace(0, c2, c2), It1)
20
21    Iy, Ix = np.gradient(It) #Affine subtraction
22    spline_x = RectBivariateSpline(np.linspace(0, r1, r1), np.linspace(0, c1, c1), Ix)
23    spline_y = RectBivariateSpline(np.linspace(0, r1, r1), np.linspace(0, c1, c1), Iy)
24
25    x, y = np.mgrid[0:c1, 0:r1]
26    x_c = np.reshape(x, (1, -1))
27    y_c = np.reshape(y, (1, -1))
28
29    #[x, y, 1]
30    coor = np.vstack((x_c, y_c, np.ones((1, r1*c1))))
31
32    grad_x = spline_x.ev(y, x).flatten()
33    grad_y = spline_y.ev(y, x).flatten()
34
35    T = splinet.ev(y, x).flatten()
36
37    A1 = np.multiply(grad_x, x_c)
38    A2 = np.multiply(grad_x, y_c)
39    A3 = np.reshape(grad_x, (1, -1))
40    A4 = np.multiply(grad_y, x_c)
41    A5 = np.multiply(grad_y, y_c)
42    A6 = np.reshape(grad_y, (1, -1))
43    A = np.vstack((A1, A2, A3, A4, A5, A6)) #Jacobian and the gradient of I
44    A = A.T
45    H = A.T@A #Hessian
46    n = 1
47
48    while(np.square(dp).sum())>threshold and n<num_iters:
49        M = np.array([[1+p[0], p[1], p[2]], [p[3], 1+p[4], p[5]], [0, 0, 1]])
50        warp = M@coor
51
52        #xp and yp coordinates
53        warp_x = warp[0]
54        warp_y = warp[1]
55
56        # gradient splines
57        warp_final = splinet1.ev(warp_y, warp_x).flatten()
58
59        #error image
60        error = np.reshape(T-warp_final, (len(warp_x), 1))
61
62        dp = np.linalg.inv(H) @ A.T @ error
63        p = (p + dp.T).ravel()
64        n+=1
65
66        dM = np.vstack((dp.reshape(2, 3), [0, 0, 1]))
67        M = M @ np.linalg.inv(dM)
```

In the classical approach we need to update A and b in every iteration until Δp converges. A being a very large matrix $D \times 6$ 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.