

16-720B HW3 Writeup

Nirmay Singaram

Andrew id: nsingara

1 Q1.1

Well if $p = [p_x, p_y]^T$ and $x = [x^{(1)}, x^{(2)}]^T$ and $\mathcal{W}(x; p) = x + p = [x^{(1)} + p_x, x^{(2)} + p_y]^T$, then using elementary matrix calculus we see that:

$$\frac{\partial \mathcal{W}(x; p)}{\partial p^T} = \frac{\partial [x^{(1)} + p_x, x^{(2)} + p_y]^T}{\partial [p_x, p_y]} = \begin{pmatrix} \frac{\partial x^{(1)} + p_x}{\partial p_x} & \frac{\partial x^{(2)} + p_y}{\partial p_y} \\ \frac{\partial x^{(1)} + p_x}{\partial p_y} & \frac{\partial x^{(2)} + p_y}{\partial p_y} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = Id_2$$

Honestly I'm not sure what more steps can be showed beyond this - this is just the multivariable equivalent of showing that $\frac{d(x+c)}{dx} = 1$.

The biggest challenge for these questions is the latex typesetting which is painful to do.

$$\mathbf{A} = \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial \mathcal{W}(\mathbf{x}; \mathbf{p})}{\partial \mathbf{p}^T} \text{ and } \mathbf{b} = \mathcal{I}_t(\mathbf{x}) - \mathcal{I}_{t+1}(\mathcal{W}(\mathbf{x}; \mathbf{p}))$$

Here I'm using the notation that $\mathbf{x}' = \mathcal{W}(\mathbf{x}; \mathbf{p})$ when convenient.

(Note we may substitute our answer for $\frac{\partial \mathcal{W}(\mathbf{x}; \mathbf{p})}{\partial \mathbf{p}^T}$ in the equation for \mathbf{A} but we'll do that later.)

These values may be reverse engineered straightforwardly:

We aim to minimize:

$$\left\| \begin{bmatrix} \mathcal{I}_{t+1}(\mathbf{x}_1 + \mathbf{p} + \Delta \mathbf{p}) - \mathcal{I}_t(\mathbf{x}) \\ \vdots \\ \mathcal{I}_{t+1}(\mathbf{x}_D + \mathbf{p} + \Delta \mathbf{p}) - \mathcal{I}_t(\mathbf{x}) \end{bmatrix} \right\|_2^2 \\ = \sum_{\mathbf{x} \in \mathbb{N}} (\mathcal{I}_{t+1}(\mathcal{W}(\mathbf{x}; \mathbf{p}) + \Delta \mathbf{p}) - \mathcal{I}_t(\mathbf{x}))^2$$

And here we use the Taylor approximation to get:

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

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

with \mathbf{A} and \mathbf{b} given as above. Note that in particular when I say $\mathbf{b} = \mathcal{I}_t(\mathbf{x}) - \mathcal{I}_{t+1}(\mathcal{W}(\mathbf{x}; \mathbf{p}))$ for instance, I am referring to a column vector with each row being as given for all $\mathbf{x} \in \mathbb{N}$. And similarly when saying: $\mathbf{A} = \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial \mathcal{W}(\mathbf{x}; \mathbf{p})}{\partial \mathbf{p}'^T}$ we are referring to the matrix with \mathbb{N} rows and 2 columns where each row is as given for every $\mathbf{x} \in \mathbb{N}$.

Then if we want to find the optimal $\Delta\mathbf{p}$ of course we differentiate and set to 0.

$$\begin{aligned} \frac{\partial \|\mathbf{A}\Delta\mathbf{p} - \mathbf{b}\|_2^2}{\partial \Delta\mathbf{p}} &= 0 \implies \frac{\partial (\mathbf{A}\Delta\mathbf{p} - \mathbf{b})^T (\mathbf{A}\Delta\mathbf{p} - \mathbf{b})}{\partial \Delta\mathbf{p}} = 0 \\ \implies \mathbf{A}^T (\mathbf{A}\Delta\mathbf{p} - \mathbf{b}) &= 0 \implies \Delta\mathbf{p} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b} \end{aligned}$$

All of this was basic calculus and algebraic manipulation without anything deep going on.

Note that the last step is possible *only* if $(\mathbf{A}^T \mathbf{A})^{-1}$ exists. Therefore in order to have a unique solution for $\Delta\mathbf{p}$ we must have that $\mathbf{A}^T \mathbf{A}$ is invertible.

2 Q1.2

This is the code:

```

1 import numpy as np
2 from scipy.interpolate import RectBivariateSpline
3
4 def LucasKanade(It, It1, rect, threshold, num_iters, p0=np.zeros(2)):
5     """
6         :param It: template image
7         :param It1: Current image
8         :param rect: Current position of the car (top left, bot right coordinates)
9         :param threshold: if the length of dp is smaller than the threshold, terminate the opt
10        :param num_iters: number of iterations of the optimization
11        :param p0: Initial movement vector [dp_x0, dp_y0]
12        :return: p: movement vector [dp_x, dp_y]
13    """
14    # Put your implementation here
15    p = p0
16    I_y, I_x = np.gradient(It1)
17
18    height, width = It.shape
19    It1_spline = RectBivariateSpline(range(height), range(width), It1)
20    I_x_spline = RectBivariateSpline(range(height), range(width), I_x)

```

```

21     I_y_spline = RectBivariateSpline(range(height), range(width), I_y)
22
23     #Finding the size of the rectangle rect
24     height_rect = rect[3] - rect[1] + 1
25     width_rect = rect[2] - rect[0] + 1
26     #Template thingies
27     inter_x = np.linspace(rect[0], rect[2], round(width_rect))
28     inter_y = np.linspace(rect[1], rect[3], round(height_rect))
29     interSpline = RectBivariateSpline(range(height), range(width), It)
30     Template = interSpline(inter_y, inter_x)
31
32
33     del_p = 10 #Initial value just to enter the while loop is all
34     iters = 0
35     while np.linalg.norm(del_p) > threshold and iters < num_iters:
36         It1_inter_x = np.linspace(rect[0] + p[0], rect[2] + p[0], round(width_rect))
37         It1_inter_y = np.linspace(rect[1] + p[1], rect[3] + p[1], round(height_rect))
38         error = (Template - It1_spline(It1_inter_y, It1_inter_x)).reshape(-1, 1)
39         dIx = I_x_spline(It1_inter_y, It1_inter_x).reshape(-1, 1)
40         dIy = I_y_spline(It1_inter_y, It1_inter_x).reshape(-1, 1)
41         dI = np.hstack((dIx, dIy))
42         dW = np.array([[1, 0], [0, 1]])
43         A = np.matmul(dI, dW)
44         #from Q1.1
45         del_p = np.linalg.inv(np.matmul(np.transpose(A), A)) @ np.transpose(A) @ error
46         #update p + \delta p
47         p[0] = p[0] + del_p[0, 0]
48         p[1] = p[1] + del_p[1, 0]
49         iters = iters + 1
50
51     return p

```

3 Q1.3

Here are the images: As you can see there is template drift, a rather big one in the case of the girl's video.





Code snippets included:

```

1 import argparse
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import matplotlib.patches as patches
5 from scipy import ndimage
6 from LucasKanade import LucasKanade

7 # write your script here, we recommend the above libraries for making your animation
8
9
10 parser = argparse.ArgumentParser()
11 parser.add_argument('--num_iters', type=int, default=1e4, help='number of iterations of Lucas-Kanade')
12 parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-Kanade')
13 args = parser.parse_args()
14 num_iters = args.num_iters
15 threshold = args.threshold

16
17 seq = np.load("./data/carseq.npy")
18 frames = seq.shape[2]
19 rect = [59, 116, 145, 151]
20 width = rect[2] - rect[0]
21 height = rect[3] - rect[1]
22 rects = []
23 rect_copy = np.copy(rect)
24 rects.append(rect_copy)
25 for i in range(0, frames-1):
26     print(i)
27     It = seq[:, :, i]
28     It1 = seq[:, :, i+1]
29     if i+1 == 1 or i+1 == 100 or i+1 == 200 or i+1 == 300 or i+1 == 400:
30         fig, ax = plt.subplots(1)
31         plt.axis('off')
32         ax.imshow(It, cmap = 'gray')
33         plot_rectangle = patches.Rectangle((rect[0], rect[1]), width,
34                                           height, linewidth = 1, edgecolor = 'r', facecolor = 'none')
35         ax.add_patch(plot_rectangle)
36         plt.savefig("CarSeq" + str(i+1)+".png")
37         plt.show()

38
39 p = LucasKanade(It, It1, rect, threshold, num_iters)

```

```

40     rect[0] = rect[0] + p[0]
41     rect[1] = rect[1] + p[1]
42     rect[2] = rect[2] + p[0]
43     rect[3] = rect[3] + p[1]
44     rects.append(np.copy(rect))
45
46     rects = np.array(rects)
47     np.save('carseqrects.npy', rects)

```

and:

```

1  import argparse
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import matplotlib.patches as patches
5  from LucasKanade import LucasKanade
6
7  # write your script here, we recommend the above libraries for making your animation
8
9  parser = argparse.ArgumentParser()
10 parser.add_argument('--num_iters', type=int, default=1e4, help='number of iterations of Lucas-Kanade')
11 parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-Kanade')
12 args = parser.parse_args()
13 num_iters = args.num_iters
14 threshold = args.threshold
15 print(num_iters)
16
17 seq = np.load("./data/girlseq.npy")
18 rect = [280, 152, 330, 318]
19 frames = seq.shape[2]
20 width = rect[2] - rect[0]
21 height = rect[3] - rect[1]
22 rects = []
23 rect_copy = np.copy(rect)
24 rects.append(rect_copy)
25 for i in range(0, frames-1):
26     print(i)
27     It = seq[:, :, i]
28     It1 = seq[:, :, i+1]
29     if i+1 == 1 or i+1 == 20 or i+1 == 40 or i+1 == 60 or i+1 == 80:
30         fig, ax = plt.subplots(1)
31         plt.axis('off')
32         ax.imshow(It, cmap = 'gray')
33         plot_rectangle = patches.Rectangle((rect[0], rect[1]), width,
34                                         height, linewidth = 1, edgecolor = 'r', facecolor = 'none')

```

```

35         ax.add_patch(plot_rectangle)
36         plt.savefig("GirlSeq" + str(i+1)+".png")
37         plt.show()
38
39         p = LucasKanade(It, It1, rect, threshold, num_iters)
40         rect[0] = rect[0] + p[0]
41         rect[1] = rect[1] + p[1]
42         rect[2] = rect[2] + p[0]
43         rect[3] = rect[3] + p[1]
44         rect.append(np.copy(rect))
45
46     rect = np.array(rects)
47     np.save("./girlseqrects.npy", rect)

```

4 Q1.4



And



As you can see there is a marked improvement with the compensate template, especially in the case of the girl.

Here is the code snippet for correctcarseq.py; the correctgirlseq.py is identical (literally copy pasted) but with the filenames changed so I'm not including that:

```

1 import argparse
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import matplotlib.patches as patches
5
6 from LucasKanade import LucasKanade
7
8 # write your script here, we recommend the above libraries for making your animation
9
10 parser = argparse.ArgumentParser()
11 parser.add_argument('--num_iters', type=int, default=1e4, help='number of iterations of Lucas-Kanade')
12 parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-Kanade')

```

```

13     parser.add_argument('--template_threshold', type=float, default=5, help='threshold for detection')
14     args = parser.parse_args()
15     num_iters = args.num_iters
16     threshold = args.threshold
17     template_threshold = args.template_threshold
18
19     seq = np.load("./data/carseq.npy")
20     rect = [59, 116, 145, 151]
21     rect_orig = np.copy(rect)
22
23     min_x, min_y, max_x, max_y = rect[0], rect[1], rect[2], rect[3]
24     sum_of_p = np.zeros((2, 1))
25     temp_1 = seq[:, :, 0]
26     frames = seq.shape[2]
27     temp = True
28     rects = []
29     for i in range(0, frames-1):
30         if(temp):
31             It = seq[:, :, i]
32             It1 = seq[:, :, i+1]
33             rect = np.array([min_x, min_y, max_x, max_y])
34             rect = rect.reshape(4, 1)
35             rects.append(rect[:, 0])
36             p_new = LucasKanade(It, It1, rect, threshold, num_iters).reshape(2, 1)
37             sum_of_p = sum_of_p + p_new
38             p_prime = LucasKanade(temp_1, It1, rect_orig, threshold, num_iters, sum_of_p)
39             if np.linalg.norm(sum_of_p - p_prime) < threshold:
40                 sum_of_p = p_prime
41                 min_x = rect_orig[0] + sum_of_p[0, 0]
42                 max_x = rect_orig[2] + sum_of_p[0, 0]
43                 min_y = rect_orig[1] + sum_of_p[1, 0]
44                 max_y = rect_orig[3] + sum_of_p[1, 0]
45
46         else:
47             sum_of_p = sum_of_p - p_new
48             temp = False
49
50     rect = np.array([min_x, min_y, max_x, max_y])
51     rect = rect.reshape(4, 1)
52     rects.append(rect[:, 0])
53     np.save("carseqrects-wcrt.npy", rects)
54
55     rect_0 = np.load("./carseqrects.npy")
56
57     for i in range(0, frames -1):
58         It = seq[:, :, i]

```

```

59     rect = rects[i]
60     if i + 1 == 1 or (i + 1)%100 == 0:
61         fig, ax = plt.subplots(1)
62         plt.axis('off')
63         ax.imshow(It, cmap = 'gray')
64         rect1 = patches.Rectangle((rect[0], rect[1]), rect[2] - rect[0],
65                                   rect[3] - rect[1], linewidth = 1, edgecolor = 'r', fill = False)
66         ax.add_patch(rect1)
67         rect2 = rect_0[i]
68         rect2 = patches.Rectangle((rect2[0], rect2[1]), rect2[2] - rect2[0],
69                                   rect2[3] - rect2[1], linewidth = 1, edgecolor = 'b', fill = False)
70         ax.add_patch(rect2)
71         plt.savefig("correctedCarSeq" + str(i+1) + ".png")
72         plt.show()

```

On second thought, let's just include it:

```

1  import argparse
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import matplotlib.patches as patches
5  from LucasKanade import LucasKanade
6
7  # write your script here, we recommend the above libraries for making your animation
8
9  parser = argparse.ArgumentParser()
10 parser.add_argument('--num_iters', type=int, default=1e4, help='number of iterations of Lucas-Kanade')
11 parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-Kanade')
12 parser.add_argument('--template_threshold', type=float, default=5, help='threshold for determining template')
13 args = parser.parse_args()
14 num_iters = args.num_iters
15 threshold = args.threshold
16 template_threshold = args.template_threshold
17
18 seq = np.load("./data/girlseq.npy")
19 rect = [280, 152, 330, 318]
20 rect_orig = np.copy(rect)
21
22 min_x, min_y, max_x, max_y = rect[0], rect[1], rect[2], rect[3]
23 sum_of_p = np.zeros((2, 1))
24 temp_1 = seq[:, :, 0]
25 frames = seq.shape[2]
26 temp = True
27 rects = []
28 for i in range(0, frames-1):

```

```

29     if(temp):
30         It = seq[:, :, i]
31         It1 = seq[:, :, i+1]
32         rect = np.array([min_x, min_y, max_x, max_y])
33         rect = rect.reshape(4, 1)
34         rect.append(rect[:, 0])
35         p_new = LucasKanade(It, It1, rect, threshold, num_iters).reshape(2, 1)
36         sum_of_p = sum_of_p + p_new
37         p_prime = LucasKanade(temp_1, It1, rect_orig, threshold, num_iters, sum_of_p)
38         if np.linalg.norm(sum_of_p - p_prime) < threshold:
39             sum_of_p = p_prime
40             min_x = rect_orig[0] + sum_of_p[0, 0]
41             max_x = rect_orig[2] + sum_of_p[0, 0]
42             min_y = rect_orig[1] + sum_of_p[1, 0]
43             max_y = rect_orig[3] + sum_of_p[1, 0]
44
45     else:
46         sum_of_p = sum_of_p - p_new
47         temp = False
48
49     rect = np.array([min_x, min_y, max_x, max_y])
50     rect = rect.reshape(4, 1)
51     rect.append(rect[:, 0])
52     np.save("girlseqrects-wcrt.npy", rect)
53
54     rect_0 = np.load("./girlseqrects.npy")
55
56     for i in range(0, frames -1):
57         It = seq[:, :, i]
58         rect = rect[i]
59         if i + 1 == 1 or (i + 1)%20 == 0:
60             fig, ax = plt.subplots(1)
61             plt.axis('off')
62             ax.imshow(It, cmap = 'gray')
63             rect1 = patches.Rectangle((rect[0], rect[1]), rect[2] - rect[0],
64             rect[3] - rect[1], linewidth = 1, edgecolor = 'r', fill = False)
65             ax.add_patch(rect1)
66             rect2 = rect_0[i]
67             rect2 = patches.Rectangle((rect2[0], rect2[1]), rect2[2] - rect2[0],
68             rect2[3] - rect2[1], linewidth = 1, edgecolor = 'b', fill = False)
69             ax.add_patch(rect2)
70             plt.savefig("correctedGirlSeq" + str(i+1) + ".png")
71             plt.show()

```

5 Q2.1

```
1     def LucasKanadeAffine(It, It1, threshold, num_iters):
2         """
3             :param It: template image
4             :param It1: Current image
5             :param threshold: if the length of dp is smaller than the threshold, terminate the optimization
6             :param num_iters: number of iterations of the optimization
7             :return: M: the Affine warp matrix [2x3 numpy array] put your implementation here
8         """
9         # put your implementation here
10        M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
11        p = np.zeros(6)
12        height, width = It.shape
13        I_y, I_x = np.gradient(It1)
14        It1_spline = RectBivariateSpline(range(height), range(width), It1)
15        I_x_spline = RectBivariateSpline(range(height), range(width), I_x)
16        I_y_spline = RectBivariateSpline(range(height), range(width), I_y)
17        spline = RectBivariateSpline(range(height), range(width), It)
18
19        #Create a meshgrid instead of a rectangle
20        X_mesh, Y_mesh = np.meshgrid(range(height), range(width))
21        x_coords = np.reshape(X_mesh, (-1, 1))
22        y_coords = np.reshape(Y_mesh, (-1, 1))
23
24        Hom_row = np.ones((x_coords.shape[0], 1))
25        It1_hom = np.transpose(np.hstack((y_coords, x_coords, Hom_row)))
26        del_p = 10 #Again just to get past the initial while loop
27        iter = 0
28        while np.linalg.norm(del_p) > threshold and iter < num_iters:
29            W = M + p.reshape(2, 3)
30            It1_hom = W@It1_hom
31
32            #Now we look for the overlapping regions
33            overlap_height = np.logical_and(It1_hom[0]>=0, It1_hom[0] < height)
34            overlap_width = np.logical_and(It1_hom[1]>= 0, It1_hom[1] < width)
35            overlap_coords = np.logical_and(overlap_height, overlap_width).nonzero()[0]
36            It1_x = It1_hom[0, overlap_coords]
37            It1_y = It1_hom[1, overlap_coords]
38            warpedIt1 = It1_spline.ev(It1_x, It1_y)
39            dIx = np.array(I_x_spline.ev(It1_x, It1_y))
40            dIy = np.array(I_y_spline.ev(It1_x, It1_y))
41            It_x = It1_hom[0, overlap_coords]
42            It_y = It1_hom[1, overlap_coords]
43            Template = np.array(spline.ev(It_x, It_y))
```

```

44     error = Template - warpedIt1
45     A_hom = np.stack((It_y*dIy, It_x*dIy, dIy, It_y*dIx, It_x*dIx, dIx), axis = 1)
46     del_p = np.linalg.pinv(A_hom.T@A_hom)@A_hom.T@error.reshape(error.shape[0], 1)
47
48     p[0] = p[0] + del_p[0, 0]
49     p[1] = p[1] + del_p[1, 0]
50     p[2] = p[2] + del_p[2, 0]
51     p[3] = p[3] + del_p[3, 0]
52     p[4] = p[4] + del_p[4, 0]
53     p[5] = p[5] + del_p[5, 0]
54     iter = iter + 1
55
56 M = M + p.reshape(2, 3)
57 return M

```

6 Q2.2

```

1 import numpy as np
2 import cv2
3 from LucasKanade import LucasKanade
4
5 from LucasKanadeAffine import LucasKanadeAffine
6 from scipy.ndimage.morphology import binary_dilation
7 from scipy.ndimage.morphology import binary_erosion
8 from scipy.ndimage import affine_transform
9
10 def SubtractDominantMotion(image1, image2, threshold, num_iters, tolerance):
11     """
12     :param image1: Images at time t
13     :param image2: Images at time t+1
14     :param threshold: used for LucasKanadeAffine
15     :param num_iters: used for LucasKanadeAffine
16     :param tolerance: binary threshold of intensity difference when computing the mask
17     :return: mask: [nxm]
18     """
19
20     # put your implementation here
21     mask = np.ones(image1.shape, dtype=bool)
22     height, width = image2.shape
23     M = LucasKanadeAffine(image1, image2, threshold, num_iters)
24     #image1_warped = affine_transform(image1, M, (width, height))
25     #I was having trouble with affine_transform so in the end I went with cv2.warpAffine
26     image2_warped = cv2.warpAffine(image1, M, (width, height))

```

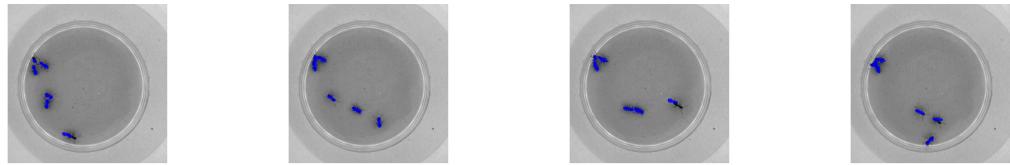
```

27     image2_warped = binary_erosion(image2_warped)
28     image2_warped = binary_dilation(image2_warped)
29     mask = abs(image2_warped - image1)
30     mask = mask > tolerance
31
32
33     return mask

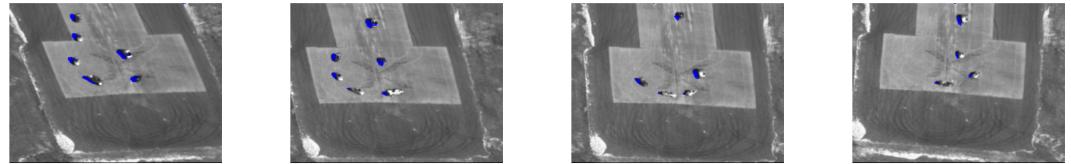
```

7 Q2.3

Here are the images:



And the aerial video:



Here is the code for the ant sequence:

```

1 import argparse
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import matplotlib.patches as patches
5 from SubtractDominantMotion import SubtractDominantMotion
6
7 # write your script here, we recommend the above libraries for making your animation
8
9 parser = argparse.ArgumentParser()
10 parser.add_argument('--num_iters', type=int, default=1e3, help='number of iterations of Lucas-Kanade')
11 parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-Kanade')
12 parser.add_argument('--tolerance', type=float, default=0.2, help='binary threshold of intensity')
13 args = parser.parse_args()
14 num_iters = args.num_iters
15 threshold = args.threshold
16 tolerance = args.tolerance
17

```

```

18 seq = np.load('~/data/antseq.npy')
19 frames = seq.shape[2]
20 for i in range(frames-1):
21     print(i)
22     image1 = seq[:, :, i]
23     image2 = seq[:, :, i+1]
24     mask = SubtractDominantMotion(image1, image2, threshold, num_iters, tolerance)
25     if (i+1)%30==0:
26         plt.imshow(image2, cmap = 'gray')
27         plt.axis('off')
28         for r in range(mask.shape[0]-1):
29             for c in range(mask.shape[1]-1):
30                 if mask[r, c] == 1:
31                     plt.scatter(c, r, s = 1, c = 'b', alpha=0.5)
32         plt.savefig("AntSeq" + str(i+1) + ".png")
33         plt.show()

```

And the code for the aerial sequence:

```

1 import argparse
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import matplotlib.patches as patches
5
6 from SubtractDominantMotion import SubtractDominantMotion
7
8 # write your script here, we recommend the above libraries for making your animation
9
10 parser = argparse.ArgumentParser()
11 parser.add_argument('--num_iters', type=int, default=1e3, help='number of iterations of Lucas-Kanade algorithm')
12 parser.add_argument('--threshold', type=float, default=1, help='dp threshold of Lucas-Kanade algorithm')
13 parser.add_argument('--tolerance', type=float, default=0.75, help='binary threshold of intensity difference')
14 args = parser.parse_args()
15 num_iters = args.num_iters
16 threshold = args.threshold
17 tolerance = args.tolerance
18
19 seq = np.load('~/data/aerialseq.npy')
20 frames = seq.shape[2]
21 for i in range(frames-1):
22     print(i)
23     image1 = seq[:, :, i]
24     image2 = seq[:, :, i+1]
25     mask = SubtractDominantMotion(image1, image2, threshold, num_iters, tolerance)
26     if (i+1)%30==0:

```

```

27         plt.imshow(image2, cmap = 'gray')
28         plt.axis('off')
29         for r in range(mask.shape[0]-1):
30             for c in range(mask.shape[1]-1):
31                 if mask[r, c] == 1:
32                     plt.scatter(c, r, s = 1, c = 'b', alpha=0.5)
33         plt.savefig("AerialSeq" + str(i+1) + ".png")
34         plt.show()

```

8 Q3.1

Here is the code for the method:

```

1 import numpy as np
2 from scipy.interpolate import RectBivariateSpline
3
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
13    # put your implementation here
14    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
15    p = np.zeros(6)
16    height, width = It.shape
17    I_y, I_x = np.gradient(It)
18
19    It1_spline = RectBivariateSpline(range(height), range(width), It1)
20    I_x_spline = RectBivariateSpline(range(height), range(width), I_x)
21    I_y_spline = RectBivariateSpline(range(height), range(width), I_y)
22    spline = RectBivariateSpline(range(height), range(width), It)
23
24    #Create a meshgrid instead of a rectangle
25    X_mesh, Y_mesh = np.meshgrid(range(width), range(height))
26    x_coords = np.reshape(X_mesh, (-1, 1))
27    y_coords = np.reshape(Y_mesh, (-1, 1))
28
29    #Now the divergences:
30
31    dI_x = I_x_spline(range(height), range(width)).reshape(-1,1)

```

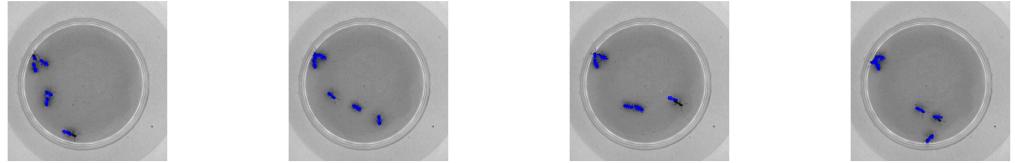
```

32     dI_y = I_y_spline(range(height), range(width)).reshape(-1, 1)
33     Template = spline(range(height), range(width)).reshape(-1)
34
35     A_new = np.squeeze(np.stack((y_coords*dI_y, x_coords*dI_y, dI_y, y_coords*dI_x, x_coords*dI_x, hom_row)))
36     hom_row = np.ones((x_coords.shape[0], 1))
37     It_hom = np.transpose(np.hstack((y_coords, x_coords, hom_row)))
38
39     del_p = 10 #Again just to get past the first while loop
40     iter = 0
41     while np.linalg.norm(del_p) > threshold and iter < num_iters:
42         W = M + p.reshape(2, 3)
43         It1_hom = W@It_hom
44         #overlap
45         overlap_height = np.logical_and(It1_hom[0]>=0, It1_hom[0] < height)
46         overlap_width = np.logical_and(It1_hom[1]>= 0, It1_hom[1] < width)
47         overlap_coords = np.logical_and(overlap_height, overlap_width).nonzero()[0]
48         It1_x = It1_hom[0, overlap_coords]
49         It1_y = It1_hom[1, overlap_coords]
50         warptedIt1 = It1_spline.ev(It1_x, It1_y)
51         Template_overlap = Template[overlap_coords]
52         error = Template_overlap - warptedIt1
53         error = error.reshape(error.shape[0], 1)
54         A_overlap_times_error = np.matmul(A_new.T[:, overlap_coords], error)
55         del_p = np.linalg.pinv(A_new[overlap_coords, :].T@A_new[overlap_coords, :])@A_overlap_times_error
56         p[0] = p[0] + del_p[0]
57         p[1] = p[1] + del_p[1]
58         p[2] = p[2] + del_p[2]
59         p[3] = p[3] + del_p[3]
60         p[4] = p[4] + del_p[4]
61         p[5] = p[5] + del_p[5]
62
63         iter = iter + 1
64         M = M + p.reshape(2, 3)
65
66
67
68
69     return M

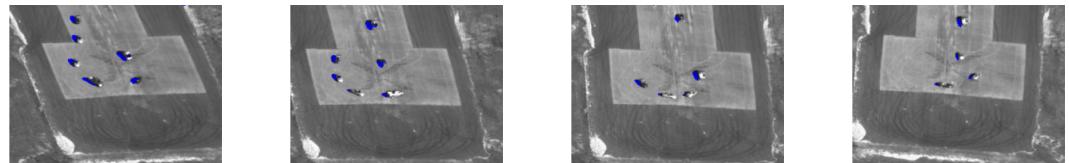
```

Of course we need to change the other files as well like test(ant/aerial)seq.py and SubtractDominantMotion.py but that's as simple as calling a different function (i.e, calling InverseCompositionAffine.py instead of LucasKanadeAffine.py) so I'm not going to paste it again.

Here are the images with the Inverse Composition method:



And the aerial video:



Honestly accuracy wise they seem fairly similar. But there are certainly performance gains:

Time for Aerial Sequence under the old/regular method: 104.8397 seconds

Time for Ant Sequence under the old/regular method: 20.4446 seconds

Time for Aerial Sequence under the new/inverse composition method: 28.1966 seconds

Time for Ant Sequence under the new/inverse composition method: 13.9562

As you can see, there is a significant performance boost. The performance boost is largely because our computations of the derivatives (the Hessian/double derivative in particular) do not need to be redone every loop (whereas in traditional LucasKanade we do) but are instead pre-computed and stored.