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:

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

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 typsetting 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:

$$\| \begin{bmatrix} \mathcal{I}_{t+1}(\mathbf{x}_1 + \mathbf{p} + \Delta\mathbf{p}) - \mathcal{I}_t(\mathbf{x}) \\ \vdots \\ \mathcal{I}_{t+1}(\mathbf{x}_D + \mathbf{p} + \Delta\mathbf{p}) - \mathcal{I}_t(\mathbf{x}) \end{bmatrix} \|_2^2$$

$$= \sum_{\mathbf{x} \in \mathbb{N}} \Big(\mathcal{I}_{t+1}(\mathcal{W}(\mathbf{x}; \mathbf{p}) + \Delta\mathbf{p}) - \mathcal{I}_t(\mathbf{x}) \Big)^2$$

And here we use the Taylor approximation to get:

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

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

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

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.

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

$$\implies \mathbf{A}^T(\mathbf{A}\Delta\mathbf{p} - \mathbf{b}) = 0 \implies \Delta\mathbf{p} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T b$$

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:

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

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 opt
    :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
    I_y, I_x = np.gradient(It1)

    height, width = It.shape
    It1_spline = RectBivariateSpline(range(height), range(width), It1)
    I_x_spline = RectBivariateSpline(range(height), range(width), I_x)
```

```
    I_y_spline = RectBivariateSpline(range(height), range(width), I_y)

    #Finding the size of the rectangle rect
    height_rect = rect[3] - rect[1] + 1
    width_rect = rect[2] - rect[0] + 1
    #Template thingies
    inter_x = np.linspace(rect[0], rect[2], round(width_rect))
    inter_y = np.linspace(rect[1], rect[3], round(height_rect))
    interSpline = RectBivariateSpline(range(height), range(width), It)
    Template = interSpline(inter_y, inter_x)


    del_p = 10 #Initial value just to enter the while loop is all
    iters = 0
    while np.linalg.norm(del_p)> threshold and iters < num_iters:
        It1_inter_x = np.linspace(rect[0] + p[0], rect[2] + p[0], round(width_rect))
        It1_inter_y = np.linspace(rect[1] + p[1], rect[3] + p[1], round(height_rect))
        error = (Template - It1_spline(It1_inter_y, It1_inter_x)).reshape(-1, 1)
        dIx = I_x_spline(It1_inter_y, It1_inter_x).reshape(-1, 1)
        dIy = I_y_spline(It1_inter_y, It1_inter_x).reshape(-1,1)
        dI = np.hstack((dIx, dIy))
        dW = np.array([[1, 0], [0, 1]])
        A = np.matmul(dI, dW)
        #from Q1.1
        del_p = np.linalg.inv(np.matmul(np.transpose(A),A))@np.transpose(A)@error
        #update p + \delta p
        p[0] = p[0] + del_p[0,0]
        p[1] = p[1] + del_p[1,0]
        iters = iters + 1

    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:

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from scipy import ndimage
from LucasKanade import LucasKanade

# write your script here, we recommend the above libraries for making your animation

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

seq = np.load("./data/carseq.npy")
frames = seq.shape[2]
rect = [59, 116, 145, 151]
width = rect[2] - rect[0]
height = rect[3] - rect[1]
rects = []
rect_copy =np.copy(rect)
rects.append(rect_copy)
for i in range(0, frames-1):
    print(i)
    It = seq[:, :, i]
    It1 = seq[:, :, i+1]
    if i+1 == 1 or i+1 == 100 or i+1 == 200 or i+1 == 300 or i+1 == 400:
        fig, ax = plt.subplots(1)
        plt.axis('off')
        ax.imshow(It, cmap = 'gray')
        plot_rectangle = patches.Rectangle((rect[0], rect[1]), width,
            height, linewidth = 1, edgecolor = 'r', facecolor = 'none')
        ax.add_patch(plot_rectangle)
        plt.savefig("CarSeq" + str(i+1)+".png")
        plt.show()

    p = LucasKanade(It, It1, rect, threshold, num_iters)
```

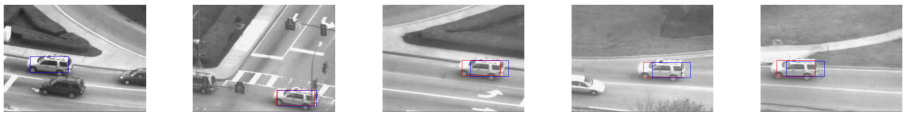
```
    rect[0] = rect[0] + p[0]
    rect[1] = rect[1] + p[1]
    rect[2] = rect[2] + p[0]
    rect[3] = rect[3] + p[1]
    rects.append(np.copy(rect))

rects = np.array(rects)
np.save('./carseqrects.npy', rects)
```

and:

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from LucasKanade import LucasKanade

# write your script here, we recommend the above libraries for making your animation

parser = argparse.ArgumentParser()
parser.add_argument('--num_iters', type=int, default=1e4, help='number of iterations of Lu
parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-K
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold
print(num_iters)

seq = np.load("./data/girlseq.npy")
rect = [280, 152, 330, 318]
frames = seq.shape[2]
width = rect[2] - rect[0]
height = rect[3] - rect[1]
rects = []
rect_copy =np.copy(rect)
rects.append(rect_copy)
for i in range(0, frames-1):
    print(i)
    It = seq[:, :, i]
    It1 = seq[:, :, i+1]
    if i+1 == 1 or i+1 == 20 or i+1 == 40 or i+1 == 60 or i+1 == 80:
        fig, ax = plt.subplots(1)
        plt.axis('off')
        ax.imshow(It, cmap = 'gray')
        plot_rectangle = patches.Rectangle((rect[0], rect[1]), width,
            height, linewidth = 1, edgecolor = 'r', facecolor = 'none')
```

```
            ax.add_patch(plot_rectangle)
            plt.savefig("GirlSeq" + str(i+1)+".png")
            plt.show()

        p = LucasKanade(It, It1, rect, threshold, num_iters)
        rect[0] = rect[0] + p[0]
        rect[1] = rect[1] + p[1]
        rect[2] = rect[2] + p[0]
        rect[3] = rect[3] + p[1]
        rects.append(np.copy(rect))

    rects = np.array(rects)
    np.save('./girlseqrects.npy', rects)
```

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:

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches

from LucasKanade import LucasKanade

# write your script here, we recommend the above libraries for making your animation

parser = argparse.ArgumentParser()
parser.add_argument('--num_iters', type=int, default=1e4, help='number of iterations of Lu
parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-K
```

```
parser.add_argument('--template_threshold', type=float, default=5, help='threshold for det
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]
rect_orig = np.copy(rect)

min_x, min_y, max_x, max_y = rect[0], rect[1], rect[2], rect[3]
sum_of_p = np.zeros((2, 1))
temp_1  = seq[:,:,0]
frames = seq.shape[2]
temp = True
rects = []
for i in range(0, frames-1):
    if(temp):
        It = seq[:, :, i]
    It1 = seq[:, :, i+1]
    rect = np.array([min_x, min_y, max_x, max_y])
    rect = rect.reshape(4, 1)
    rects.append(rect[:, 0])
    p_new = LucasKanade(It, It1, rect, threshold, num_iters).reshape(2, 1)
    sum_of_p = sum_of_p + p_new
    p_prime = LucasKanade(temp_1, It1, rect_orig, threshold, num_iters, sum_of_p)
    if np.linalg.norm(sum_of_p - p_prime) < threshold:
        sum_of_p = p_prime
        min_x = rect_orig[0] + sum_of_p[0, 0]
        max_x = rect_orig[2] + sum_of_p[0, 0]
        min_y = rect_orig[1] + sum_of_p[1, 0]
        max_y = rect_orig[3] + sum_of_p[1, 0]

    else:
        sum_of_p = sum_of_p - p_new
        temp = False

rect = np.array([min_x, min_y, max_x, max_y])
rect = rect.reshape(4, 1)
rects.append(rect[:, 0])
np.save("carseqrects-wcrt.npy", rects)

rect_0 = np.load('./carseqrects.npy')

for i in range(0, frames -1):
    It = seq[:, :, i]
```

```
    rect = rects[i]
    if i + 1 == 1 or (i + 1)%100 == 0:
        fig, ax = plt.subplots(1)
        plt.axis('off')
        ax.imshow(It, cmap = 'gray')
        rect1 = patches.Rectangle((rect[0], rect[1]), rect[2] - rect[0],
        rect[3] - rect[1], linewidth = 1, edgecolor = 'r', fill = False)
        ax.add_patch(rect1)
        rect2 = rect_0[i]
        rect2 = patches.Rectangle((rect2[0], rect2[1]), rect2[2] - rect2[0],
        rect2[3] - rect2[1], linewidth = 1, edgecolor = 'b', fill = False)
        ax.add_patch(rect2)
        plt.savefig("correctedCarSeq" + str(i+1) + ".png")
        plt.show()
```

On second thought, let's just include it:

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from LucasKanade import LucasKanade

# write your script here, we recommend the above libraries for making your animation

parser = argparse.ArgumentParser()
parser.add_argument('--num_iters', type=int, default=1e4, help='number of iterations of Lu
parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-K
parser.add_argument('--template_threshold', type=float, default=5, help='threshold for det
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]
rect_orig = np.copy(rect)

min_x, min_y, max_x, max_y = rect[0], rect[1], rect[2], rect[3]
sum_of_p = np.zeros((2, 1))
temp_1  = seq[:,:,0]
frames = seq.shape[2]
temp = True
rects = []
for i in range(0, frames-1):
```

```
        if(temp):
            It = seq[:, :, i]
        It1 = seq[:, :, i+1]
        rect = np.array([min_x, min_y, max_x, max_y])
        rect = rect.reshape(4, 1)
        rects.append(rect[:, 0])
        p_new = LucasKanade(It, It1, rect, threshold, num_iters).reshape(2, 1)
        sum_of_p = sum_of_p + p_new
        p_prime = LucasKanade(temp_1, It1, rect_orig, threshold, num_iters, sum_of_p)
        if np.linalg.norm(sum_of_p - p_prime) < threshold:
            sum_of_p = p_prime
            min_x = rect_orig[0] + sum_of_p[0, 0]
            max_x = rect_orig[2] + sum_of_p[0, 0]
            min_y = rect_orig[1] + sum_of_p[1, 0]
            max_y = rect_orig[3] + sum_of_p[1, 0]

        else:
            sum_of_p = sum_of_p - p_new
            temp = False

    rect = np.array([min_x, min_y, max_x, max_y])
    rect = rect.reshape(4, 1)
    rects.append(rect[:, 0])
    np.save("girlseqrects-wcrt.npy", rects)

    rect_0 = np.load('./girlseqrects.npy')

    for i in range(0, frames -1):
        It = seq[:, :, i]
        rect = rects[i]
        if i + 1 == 1 or (i + 1)%20 == 0:
            fig, ax = plt.subplots(1)
            plt.axis('off')
            ax.imshow(It, cmap = 'gray')
            rect1 = patches.Rectangle((rect[0], rect[1]), rect[2] - rect[0],
            rect[3] - rect[1], linewidth = 1, edgecolor = 'r', fill = False)
            ax.add_patch(rect1)
            rect2 = rect_0[i]
            rect2 = patches.Rectangle((rect2[0], rect2[1]), rect2[2] - rect2[0],
            rect2[3] - rect2[1], linewidth = 1, edgecolor = 'b', fill = False)
            ax.add_patch(rect2)
            plt.savefig("correctedGirlSeq" + str(i+1) + ".png")
            plt.show()
```

5 Q2.1

```
            def LucasKanadeAffine(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 optimiz
    :param num_iters: number of iterations of the optimization
    :return: M: the Affine warp matrix [2x3 numpy array] put your implementation here
    """
    # put your implementation here
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
    p = np.zeros(6)
    height, width = It.shape
    I_y, I_x = np.gradient(It1)
    It1_spline = RectBivariateSpline(range(height), range(width), It1)
    I_x_spline = RectBivariateSpline(range(height), range(width), I_x)
    I_y_spline = RectBivariateSpline(range(height), range(width), I_y)
    spline = RectBivariateSpline(range(height), range(width), It)

    #Create a meshgrid instead of a rectangle
    X_mesh, Y_mesh = np.meshgrid(range(height), range(width))
    x_coords = np.reshape(X_mesh, (-1, 1))
    y_coords = np.reshape(Y_mesh, (-1, 1))

    Hom_row = np.ones((x_coords.shape[0], 1))
    It_hom = np.transpose(np.hstack((y_coords, x_coords, Hom_row)))
    del_p = 10 #Again just to get past the initial while loop
    iter = 0
    while np.linalg.norm(del_p) > threshold and iter < num_iters:
        W = M + p.reshape(2, 3)
        It1_hom = W@It_hom

        #Now we look for the overlapping regions
        overlap_height = np.logical_and(It1_hom[0]>=0, It1_hom[0] < height)
        overlap_width = np.logical_and(It1_hom[1]>= 0, It1_hom[1] < width)
        overlap_coords = np.logical_and(overlap_height, overlap_width).nonzero()[0]
        It1_x = It1_hom[0, overlap_coords]
        It1_y = It1_hom[1, overlap_coords]
        warpedIt1 = It1_spline.ev(It1_x, It1_y)
        dIx = np.array(I_x_spline.ev(It1_x, It1_y))
        dIy = np.array(I_y_spline.ev(It1_x, It1_y))
        It_x = It_hom[0, overlap_coords]
        It_y = It_hom[1, overlap_coords]
        Template = np.array(spline.ev(It_x, It_y))
```

```
        error = Template - warpedIt1
        A_hom = np.stack((It_y*dIy, It_x*dIy, dIy, It_y*dIx, It_x*dIx, dIx), axis = 1)
        del_p = np.linalg.pinv(A_hom.T@A_hom)@A_hom.T@error.reshape(error.shape[0], 1)

        p[0] = p[0] + del_p[0, 0]
        p[1] = p[1] + del_p[1, 0]
        p[2] = p[2] + del_p[2, 0]
        p[3] = p[3] + del_p[3, 0]
        p[4] = p[4] + del_p[4, 0]
        p[5] = p[5] + del_p[5, 0]
        iter = iter + 1

    M = M + p.reshape(2, 3)
    return M
```

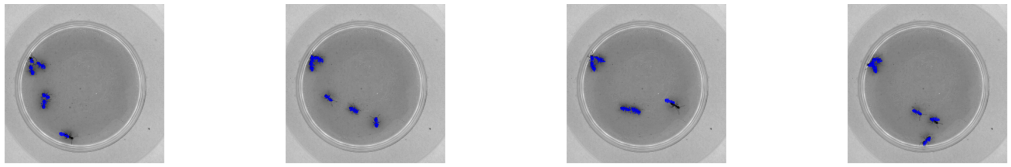
6 Q2.2

```
import numpy as np
import cv2
from LucasKanade import LucasKanade

from LucasKanadeAffine import LucasKanadeAffine
from scipy.ndimage.morphology import binary_dilation
from scipy.ndimage.morphology import binary_erosion
from scipy.ndimage import affine_transform

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)
    height, width = image2.shape
    M = LucasKanadeAffine(image1, image2, threshold, num_iters)
    #image1_warped = affine_transform(image1, M, (width, height))
    #I was having trouble with affine_transform so in the end I went with cv2.warpAffine
    image2_warped = cv2.warpAffine(image1, M, (width, height))
```

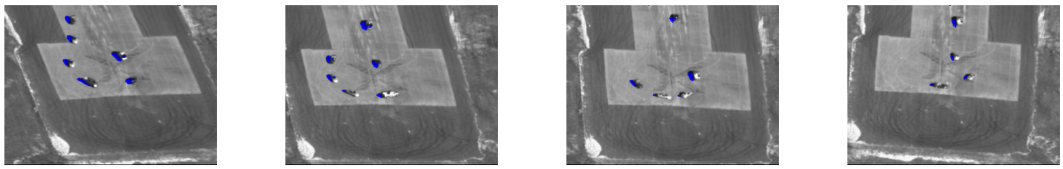
```
    image2_warped = binary_erosion(image2_warped)
    image2_warped = binary_dilation(image2_warped)
    mask = abs(image2_warped - image1)
    mask = mask > tolerance


    return mask
```

7 Q2.3

Here are the images:



And the aerial video:



Here is the code for the ant sequence:

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from SubtractDominantMotion import SubtractDominantMotion

# write your script here, we recommend the above libraries for making your animation

parser = argparse.ArgumentParser()
parser.add_argument('--num_iters', type=int, default=1e3, help='number of iterations of Lu
parser.add_argument('--threshold', type=float, default=1e-2, help='dp threshold of Lucas-K
parser.add_argument('--tolerance', type=float, default=0.2, help='binary threshold of inte
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold
tolerance = args.tolerance

```

```
seq = np.load('./data/antseq.npy')
frames = seq.shape[2]
for i in range(frames-1):
    print(i)
    image1 = seq[:, :, i]
    image2 = seq[:, :, i+1]
    mask = SubtractDominantMotion(image1, image2, threshold, num_iters, tolerance)
    if (i+1)%30==0:
        plt.imshow(image2, cmap = 'gray')
        plt.axis('off')
        for r in range(mask.shape[0]-1):
            for c in range(mask.shape[1]-1):
                if mask[r, c] == 1:
                    plt.scatter(c, r, s = 1, c = 'b', alpha=0.5)
        plt.savefig("AntSeq" + str(i+1) + ".png")
        plt.show()
```

And the code for the aerial sequence:

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches

from SubtractDominantMotion import SubtractDominantMotion

# write your script here, we recommend the above libraries for making your animation

parser = argparse.ArgumentParser()
parser.add_argument('--num_iters', type=int, default=1e3, help='number of iterations of Lu
parser.add_argument('--threshold', type=float, default=1, help='dp threshold of Lucas-Kana
parser.add_argument('--tolerance', type=float, default=0.75, help='binary threshold of int
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold
tolerance = args.tolerance

seq = np.load('./data/aerialseq.npy')
frames = seq.shape[2]
for i in range(frames-1):
    print(i)
    image1 = seq[:, :, i]
    image2 = seq[:, :, i+1]
    mask = SubtractDominantMotion(image1, image2, threshold, num_iters, tolerance)
    if (i+1)%30==0:
```

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

8 Q3.1

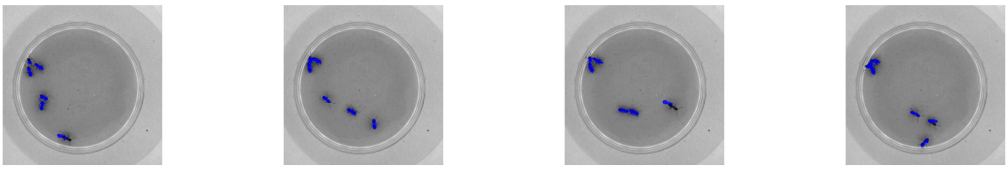
Here is the code for the method:

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

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 opt
    :param num_iters: number of iterations of the optimization
    :return: M: the Affine warp matrix [2x3 numpy array]
    """

    # put your implementation here
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
    p = np.zeros(6)
    height, width = It.shape
    I_y, I_x = np.gradient(It)

    It1_spline = RectBivariateSpline(range(height), range(width), It1)
    I_x_spline = RectBivariateSpline(range(height), range(width), I_x)
    I_y_spline = RectBivariateSpline(range(height), range(width), I_y)
    spline = RectBivariateSpline(range(height), range(width), It)

    #Create a meshgrid instead of a rectangle
    X_mesh, Y_mesh = np.meshgrid(range(width), range(height))
    x_coords = np.reshape(X_mesh, (-1, 1))
    y_coords = np.reshape(Y_mesh, (-1, 1))

    #Now the divergences:

    dI_x = I_x_spline(range(height), range(width)).reshape(-1,1)
```

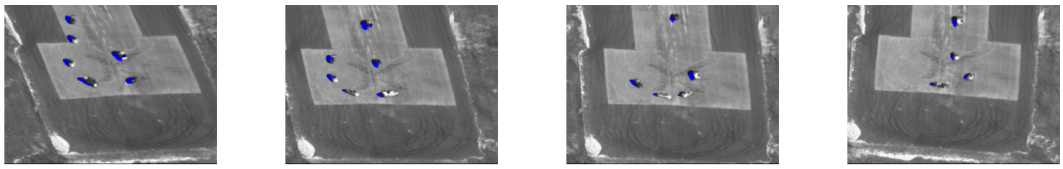
```
    dI_y = I_y_spline(range(height), range(width)).reshape(-1, 1)
    Template = spline(range(height), range(width)).reshape(-1)

    A_new = np.squeeze(np.stack((y_coords*dI_y, x_coords*dI_y, dI_y, y_coords*dI_x, x_coor
    hom_row = np.ones((x_coords.shape[0], 1))
    It_hom = np.transpose(np.hstack((y_coords, x_coords, hom_row)))

    del_p = 10 #Again just to get past the first while loop
    iter = 0
    while np.linalg.norm(del_p) > threshold and iter < num_iters:
        W = M + p.reshape(2, 3)
        It1_hom = W@It_hom
        #overlap
        overlap_height = np.logical_and(It1_hom[0]>=0, It1_hom[0] < height)
        overlap_width = np.logical_and(It1_hom[1]>= 0, It1_hom[1] < width)
        overlap_coords = np.logical_and(overlap_height, overlap_width).nonzero()[0]
        It1_x = It1_hom[0, overlap_coords]
        It1_y = It1_hom[1, overlap_coords]
        warptedIt1 = It1_spline.ev(It1_x, It1_y)
        Template_overlap = Template[overlap_coords]
        error = Template_overlap - warptedIt1
        error = error.reshape(error.shape[0], 1)
        A_overlap_times_error = np.matmul(A_new.T[:, overlap_coords], error)
        del_p = np.linalg.pinv(A_new[overlap_coords, :].T@A_new[overlap_coords, :])@A_over
        p[0] = p[0] + del_p[0]
        p[1] = p[1] + del_p[1]
        p[2] = p[2] + del_p[2]
        p[3] = p[3] + del_p[3]
        p[4] = p[4] + del_p[4]
        p[5] = p[5] + del_p[5]

        iter = iter + 1
    M = M + p.reshape(2, 3)




    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.