LUCAS KANADE HOMEWORK 3

Sagar Sachdev (sagarsac)

1.1 a)

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

$$=> \begin{bmatrix} W_x(x;p) \\ W_y(x;p) \end{bmatrix} = \begin{bmatrix} x + p_1 \\ y + p_2 \end{bmatrix}$$

Taking derivative w.r.t p:

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

1.1 b)

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

A is the steepest gradient of the images

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

b is the error image

1.1 c)

To minimize Δp:

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

$$\sum_{x \in \boldsymbol{N}} A^T(A\Delta p - b) = 0$$

$$\sum_{x \in \boldsymbol{N}} A^T A \Delta p = \sum_{x \in \boldsymbol{N}} A^T b$$

$$\Delta p = \sum_{x \in \boldsymbol{N}} A^T A (A^T b)$$

$for\ \Delta p\ to\ have\ a\ unique\ solution\ A^T A\ must\ be\ invertible\ and\ non\ singular.$

1.2

LucasKanade.py > ...

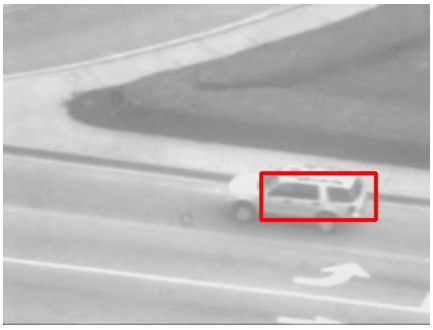
```python
import numpy as np
from scipy.interpolate import RectBivariateSpline

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 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]
    """
    p = p0
    # Adding a threshold value below which delta p should be
    # Put your implementation here
    h1 = It.shape[0]
    w1 = It.shape[1]
    h2 = It1.shape[0]
    w2 = It1.shape[1]

    x0 = rect[0]
    y0 = rect[1]

    x1 = rect[2]
    y1 = rect[3]

    interp_It = RectBivariateSpline(np.arange(h1),np.arange(w1),It)
    interp_It1 = RectBivariateSpline(np.arange(h2),np.arange(w2),It1)

    columns,rows = np.meshgrid(np.linspace(x0,x1,87),np.linspace(y0,y1,36))

    var1 = int((y1-y0)+1)
    var2 = int((x1-x0)+1)

    pixel = interp_It.ev(rows,columns)
    for i in range(int(num_iters)):

        warped_rows = np.linspace(p[1]+y0,p[1]+y1, 36)
        warped_col = np.linspace(p[0]+x0,p[0]+x1, 87)
        grad_col, grad_row = np.meshgrid(warped_col,warped_rows)

        dx = interp_It1.ev(grad_row,grad_col,dx=1,dy=0)
        dy = interp_It1.ev(grad_row,grad_col,dx=0,dy=1)

        A = np.vstack((dy.flatten(),dx.flatten())).T

        pixel_warped = interp_It1.ev(grad_row,grad_col)
        b = (pixel.flatten()-pixel_warped.flatten())

        del_p = np.linalg.pinv(A.T @ A) @ A.T @ b
        p += del_p
        if np.linalg.norm(del_p) < threshold:
            break
    return p

```

1.3

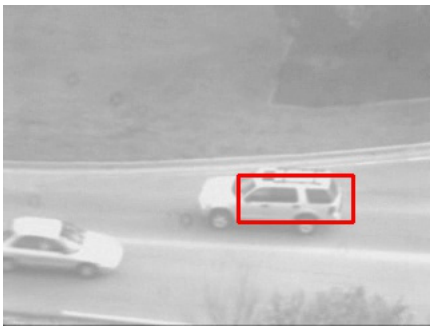
testCarSequence.py Results:

Frame 1

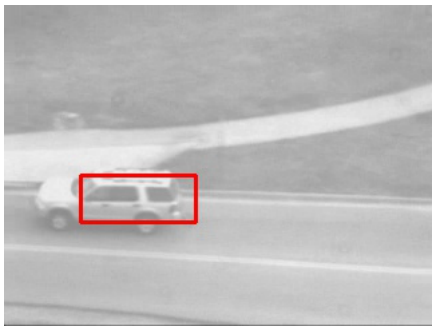
Frame 100



Frame 200

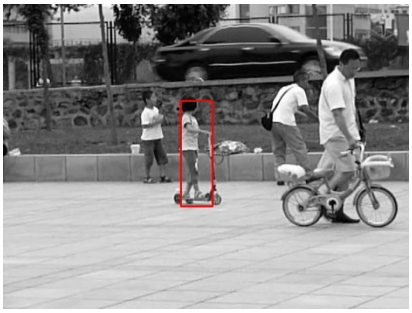


Frame 300

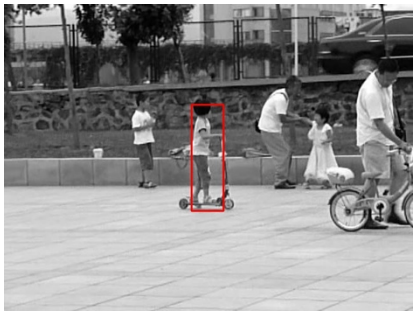


Frame 400

testGirlSequence.py Results:

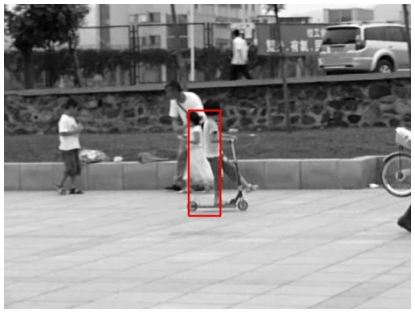


Frame 1

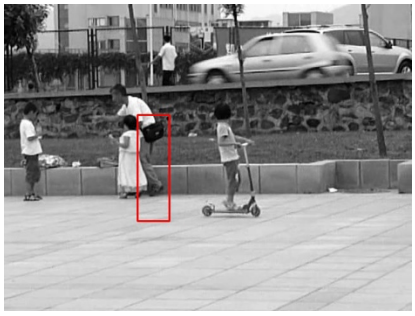


Frame 20

Frame 40



Frame 60



Frame 80

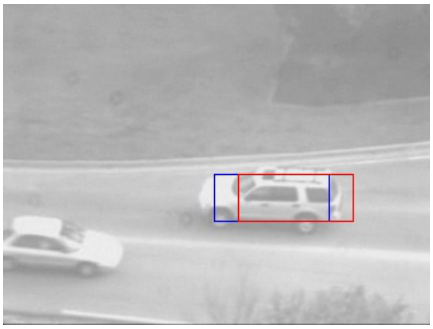
1.4

testCarSequenceWithTemplateCorrection.py

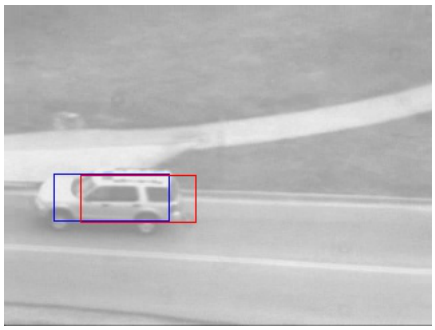
Frame 0

Frame 100

Frame 200



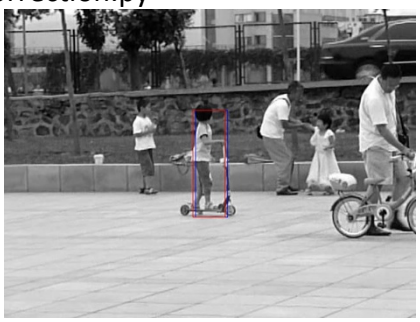
Frame 300



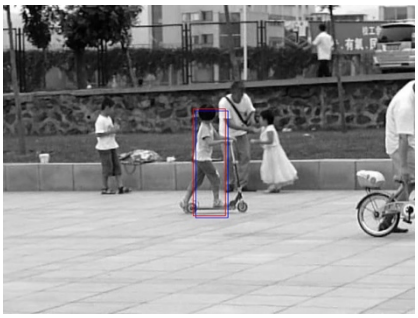
Frame 400

testGirlSequenceWithTemplateCorrection.py

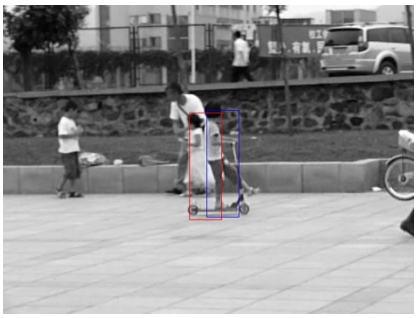
Frame 0



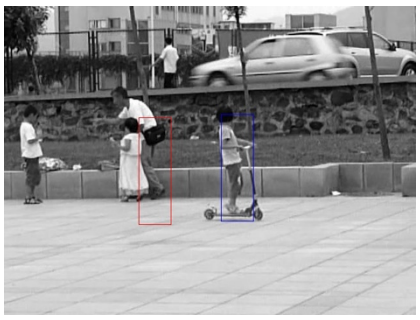
Frame 20



Frame 40



Frame 60



Frame 80

2.1

LucasKanadeAffine.py > LucasKanadeAffine

```python
import numpy as np
from scipy.interpolate import RectBivariateSpline
from scipy.ndimage import affine_transform, sobel

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 optimization
    :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)

    h1 = It.shape[0]
    w1 = It.shape[1]
    h2 = It1.shape[0]
    w2 = It1.shape[1]
    interp_It = RectBivariateSpline(np.arange(h1),np.arange(w1),It)
    interp_It1 = RectBivariateSpline(np.arange(h2),np.arange(w2),It1)

    for i in range(int(num_iters)):
        #Computing b
        It_warp = affine_transform(It, M)

        #Computing A
        grad_col, grad_row = np.meshgrid(np.linspace(0,w1-1,w1),np.linspace(0,h1-1,h1))

        grad_col = grad_col.flatten()
        grad_row = grad_row.flatten()

        dx = affine_transform(sobel(It1,1),M)
        dy = affine_transform(sobel(It1,0),M)

        dx = dx.flatten().T
        dy = dy.flatten().T

        A = np.array([dx * grad_col, dx * grad_row, dx,
                      dy * grad_col, dy * grad_row, dy]).T
        b = (It-It_warp).flatten()
        del_p = np.linalg.pinv(A.T @ A) @ A.T @ b
        p += del_p
        M[0,0] = 1+p[0]
        M[0,1] = p[1]
        M[0,2] = p[2]
        M[1,0] = p[3]
        M[1,1] = 1+p[4]
        M[1,2] = p[5]
        if np.linalg.norm(del_p) < threshold:
            break
    return M
```

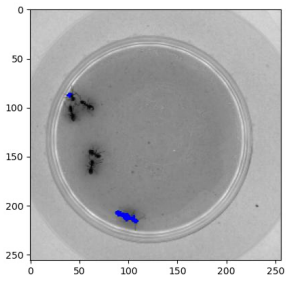
2.2

SubtractDominantMotion.py > SubtractDominantMotion

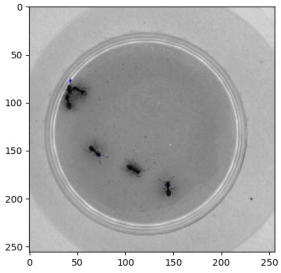
```python
import numpy as np
from scipy.ndimage.morphology import binary_erosion, binary_dilation
from scipy.ndimage import affine_transform
from LucasKanadeAffine import LucasKanadeAffine
from InverseCompositionAffine import InverseCompositionAffine

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)
    # M = LucasKanadeAffine(image1, image2, threshold, num_iters)
    M = InverseCompositionAffine(image1,image2,threshold,num_iters)

    It_warp = affine_transform(image1,M)
    err = np.absolute(It_warp-image2)
    mask = err>tolerance
    mask = binary_dilation(mask,iterations = 5)
    mask = binary_erosion(mask,iterations = 4)

    return mask

```

2.3

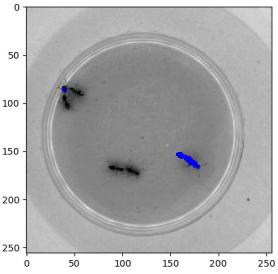
testAntSequence.py



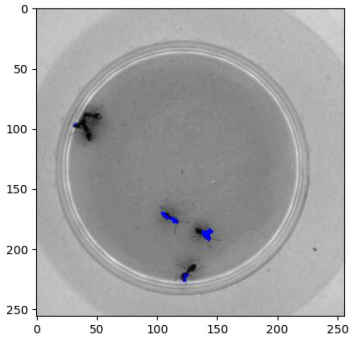
Frame 30



Frame 60

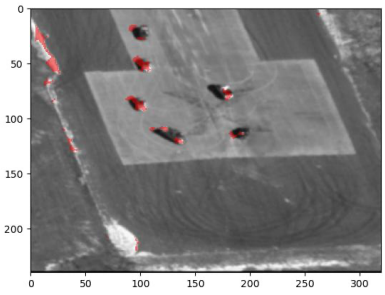


Frame 90

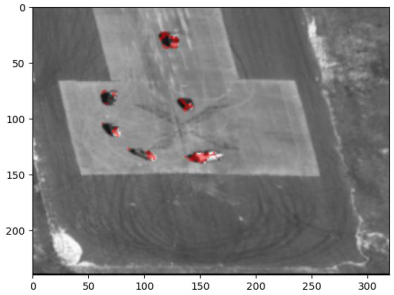


Frame 120

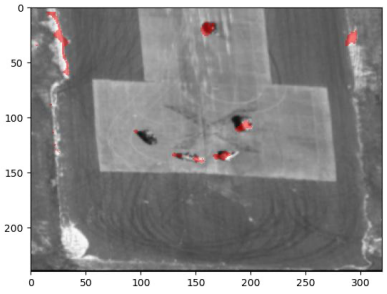
testAerialSequence.py



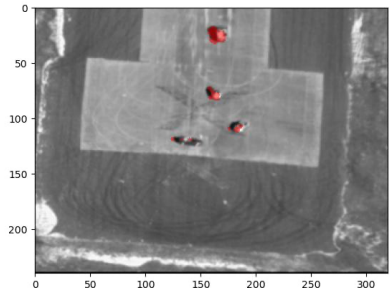
Frame 30



Frame 60



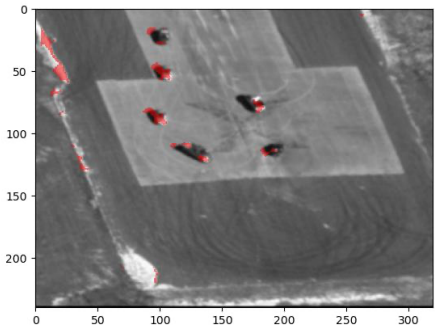
Frame 90



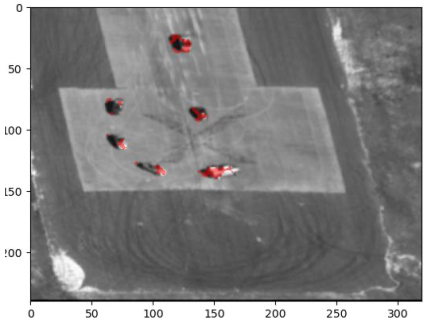
Frame 120

3.1

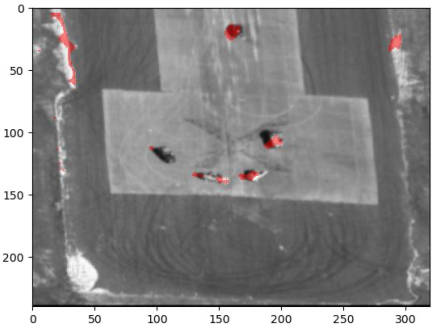
testAerialSequence.py (Inverse Composition)



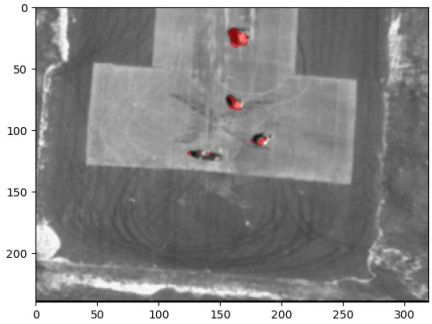
Frame 30



Frame 60

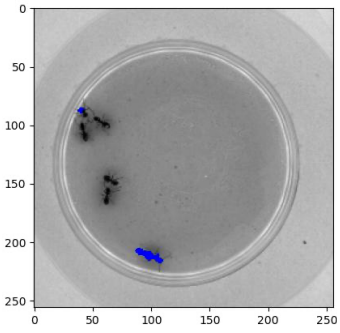


Frame 90

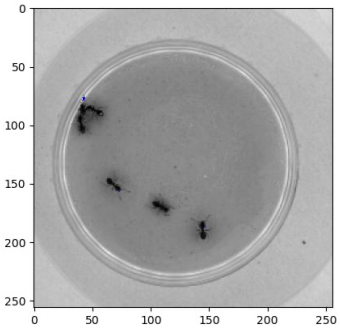


Frame 120

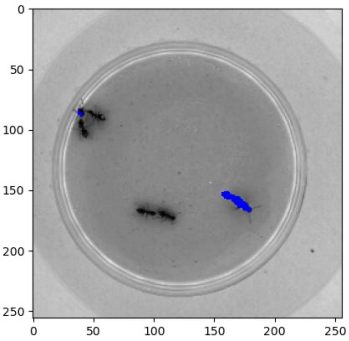
testAntSequence.py



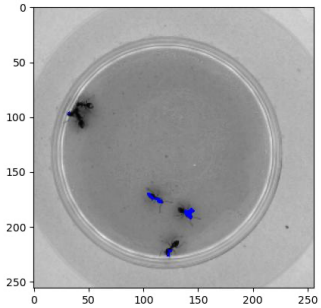
Frame 30



Frame 60



Frame 90

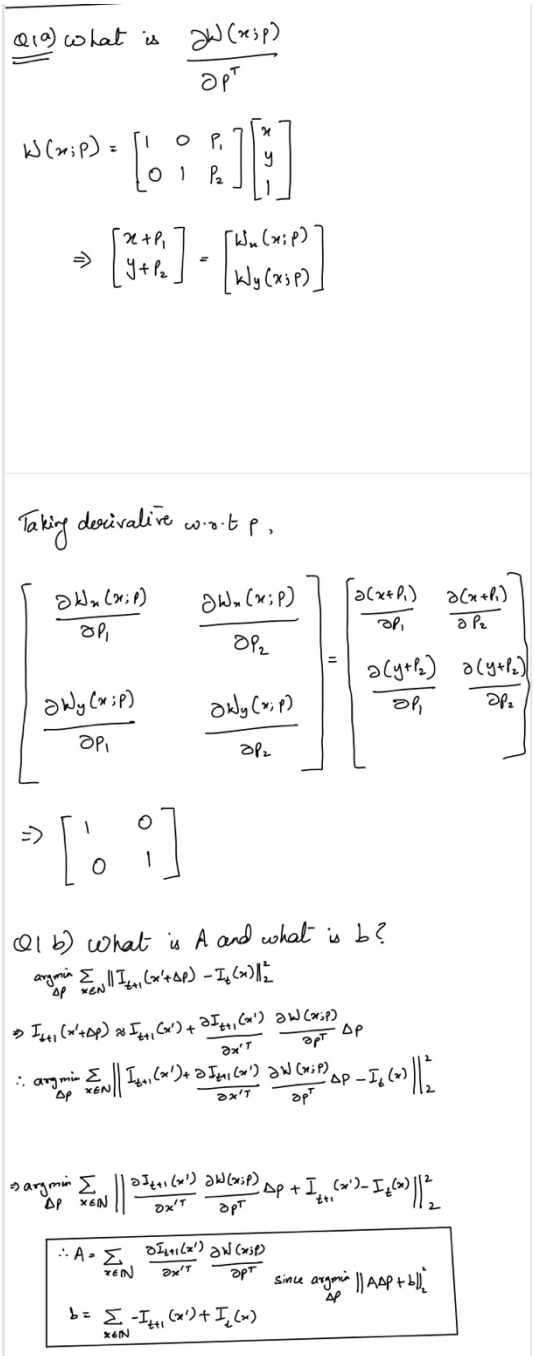


Frame 120

ACKNOWLEDGEMENTS:

1) My research lab mate Bassam Bikdash helped with Lucas Kanade Affine warp calculation for A matrix
2) Resources for Plotting image on image:
    a. https://matplotlib.org/3.5.1/api/_as_gen/matplotlib.pyplot.imshow.html
    b. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html
    c. My research lab mate Bassam Bikdash

ADDITIONAL HAND CALCS FOR REFERENCE:



Q1 c)

$$\underset{\Delta p}{\operatorname{argmin}} \sum_{x \in \mathbb{N}} \left\| \underbrace{\frac{\partial I_{t+1}(x')}{\partial x'^T} \frac{\partial W(x;p)}{\partial p^T}}_{A} \Delta p + \underbrace{I_{t+1}(x') - I_t(x)}_{b} \right\|_2^2$$

$$\Rightarrow \underset{\Delta p}{\operatorname{argmin}} \| A\Delta p - b \|$$

To minimize the above equation :

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

$$\Rightarrow \sum_{x \in \mathbb{N}} A^T A \Delta p - A^T b = 0$$

$$\Rightarrow \sum_{x \in \mathbb{N}} A^T A \Delta p = \sum_{x \in \mathbb{N}} A^T b$$

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

$\therefore A^T A$ needs to be invertible