16-720 Lucas-Kanade Tracking

Jiyoon Park (jiyoonp) HW 2

1 Lucas-Kanade Tracking

Q1.1

a)

$$\frac{\partial W(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T} = \begin{bmatrix} \frac{\partial(x+p_x)}{\partial p_x} & \frac{\partial(x+p_x)}{\partial p_y} \\ \frac{\partial(y+p_y)}{\partial p_x} & \frac{\partial(y+p_y)}{\partial p_y} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$ for $$W(\mathbf{x};\mathbf{p}) = \begin{bmatrix} x+p_x \\ y+p_y \end{bmatrix}$$

b) A and b matrix is

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

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

c)

$A^T A$ must be invertible to have a unique solution for $\Delta\mathbf{p}$. In other words, it should be full rank.

16-720 Jiyoon Park (jiyoonp) HW 2

Q1.2

```
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
    # set up the threshold
    ################## TODO Implement Lucas Kanade ##################

    x1, y1, x2, y2 = rect
    rect_size = [int(x2-x1), int(y2-y1)]
    p = p0

    It_interp = RectBivariateSpline(np.arange(It.shape[0]), np.arange(It.shape[1]),It)
    It1_interp = RectBivariateSpline(np.arange(It1.shape[0]),
np.arange(It1.shape[1]),It1)

    x = np.linspace(x1,x2, rect_size[0]) # 59 60 61 ... 144 145
    y = np.linspace(y1,y2, rect_size[1]) # 116 117 ... 150 151
    grid_x , grid_y = np.meshgrid(x, y)
    It_patch = It_interp.ev(grid_y, grid_x)

    itered = 0

    while True:
        itered += 1
        x_it1 = np.linspace(x1 + p[0], x2 + p[0], rect_size[0])
        y_it1 = np.linspace(y1 + p[1], y2 + p[1], rect_size[1])
        grid_x_it1, grid_y_it1 = np.meshgrid(x_it1, y_it1)
        It1_interp_patch = It1_interp.ev(grid_y_it1, grid_x_it1)

        Ix = It1_interp.ev(grid_y_it1, grid_x_it1 , dy=1).flatten()
        Iy = It1_interp.ev(grid_y_it1, grid_x_it1 , dx=1).flatten()

        A = np.zeros((rect_size[0]*rect_size[1], 2))
        A[:, 0] = Ix.flatten()
        A[:, 1] = Iy.flatten()

        B = np.zeros((rect_size[0]*rect_size[1], 1))

        It_d =  It_patch.flatten()-It1_interp_patch.flatten()
        B = It_d

        At = np.transpose(A)
        [u,v] = np.dot(np.linalg.pinv(np.dot(At, A)),np.dot( At, B))
        p_star = np.linalg.norm([u,v], ord=2)
        p[0] += u
        p[1] += v
        if p_star<= threshold or itered>=num_iters:
            break

    return p
```

Q 1.3

The results of Lucas Kanade Tracking

1) Car

2) Girl Sequence

16-720 Jiyoon Park (jiyoonp) HW 2

Q 1.4

The results of Lucas Kanade Tracking with template correction:

The red box is with template correction and the purple is without

1) Car

2) Girl Sequence

Question 2

Q 2.1

```
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
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]])
    ################### TODO Implement Lucas Kanade Affine ###################

    w, h = It.shape
    wh = w*h

    It_interp = RectBivariateSpline(np.arange(It.shape[0]), np.arange(It.shape[1]),
It)
    It1_interp = RectBivariateSpline(np.arange(w), np.arange(h), It1)

    itered = 0

    while True:
        itered += 1

        x = np.linspace(0, w, w)
        y = np.linspace(0, h, h)
        grid_x, grid_y = np.meshgrid(x, y)

        grid_x, grid_y = grid_x.flatten(), grid_y.flatten()

        new_grid_x = M[0, 0]*grid_x.flatten() + M[0, 1]*grid_y.flatten() + M[0, 2]
        new_grid_y = M[1, 0]*grid_x.flatten() + M[1, 1]*grid_y.flatten() + M[1, 2]

        new_grid_x, new_grid_y=new_grid_x.flatten(),  new_grid_y.flatten()

        valid = (new_grid_x>0) & (new_grid_x<w) * (new_grid_y>0) & (new_grid_y<h)

        valid_x_w_grid = new_grid_x[valid]
        valid_y_w_grid = new_grid_y[valid]
        valid_x_grid = grid_x[valid]
        valid_y_grid = grid_y[valid]

        new_len = valid_y_grid.shape[0]

        valid_template = It_interp.ev(valid_y_grid,valid_x_grid)
        valid_warped = It1_interp.ev(valid_y_w_grid,valid_x_w_grid)

        D = valid_template.flatten() - valid_warped.flatten() # wh * 1

        d_It1_x = It1_interp.ev(valid_y_w_grid, valid_x_w_grid, dy=1) #307200
        d_It1_y = It1_interp.ev(valid_y_w_grid, valid_x_w_grid, dx=1) #307200

        d_It1_x = d_It1_x.reshape(new_len,1)
        d_It1_y = d_It1_y.reshape(new_len,1)

        valid_x_w_grid = valid_x_w_grid.reshape(new_len, )
        valid_y_w_grid = valid_y_w_grid.reshape(new_len, )
```

```
        valid_x_grid = valid_x_grid.reshape(new_len,1)
        valid_y_grid = valid_y_grid.reshape(new_len,1)

        # dW = np.zeros((new_len, 2, 6))
        # dW[:, 0, 0] = valid_x_w_grid
        # dW[:, 0, 1] = valid_y_w_grid
        # dW[:, 0, 2] = np.ones(new_len)
        # dW[:, 1, 3] = valid_x_w_grid
        # dW[:, 1, 4] = valid_y_w_grid
        # dW[:, 1, 5] = np.ones(new_len)

        A1 = d_It1_x.T* valid_x_grid.T
        A2 = d_It1_x.T* valid_y_grid.T
        A3 = d_It1_y.T*valid_x_grid.T
        A4 = d_It1_y.T* valid_y_grid.T
        #
        A = np.concatenate((A1, A2, d_It1_x.T, A3, A4, d_It1_y.T), axis=0)
        # print(A.shape)

        # dI = np.zeros((new_len, 1, 2))
        # dI[:, :, 0] = d_It1_x
        # dI[:, :, 1] = d_It1_y

        # A_pre = np.einsum('ijk, imj -> imk', dW, dI)

        # A = A_pre[:, 0, :]
        A =A.T
        At = np.transpose(A)
        H = np.dot(At, A)

        dp = np.dot(np.linalg.pinv(H), np.dot(At, D)).reshape(2, 3)

        p_error = np.linalg.norm(dp, ord=2)

        M += dp

        if p_error <= threshold or itered >= num_iters:
            break

    print("\niter :", itered)
    return M
```

Q 2.2

```
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)

    ################### TODO Implement Substract Dominent Motion ###################

    M = LucasKanadeAffine(image1, image2, threshold, num_iters) # gives M that makes
img2 to match img1
    img2_m = affine_transform(image2, M)
    diff = img2_m - image1
    diff = abs(diff)
    diff = diff.flatten()

    w, h = image1.shape
    wh = w*h

    img2_m = img2_m.flatten()
    valid = np.nonzero(img2_m)
    valid = np.array(valid)

    invalid = np.where(img2_m ==0)

    diff[invalid] = 0

    mask = diff>tolerance
    mask = np.array(mask)
    mask[invalid] = 0
    mask = mask.reshape(w, h)
    img2_m = img2_m.reshape(w, h)
    diff = diff.reshape(w, h)


    mask = binary_erosion(mask, structure=np.ones((1,1))).astype(mask.dtype)
    mask = binary_dilation(mask, structure=np.ones((2,2))).astype(mask.dtype)

    # plt.subplot(3, 2, 1)
    # plt.imshow(image1)
    # plt.title("image 1")
    #
    # plt.subplot(3, 2, 2)
    # plt.imshow(image2)
    # plt.title("image 2")
    #
    # plt.subplot(3, 2, 3)
    # plt.imshow(img2_m)
    # plt.title("img1_m")
    #
    # plt.subplot(3, 2, 4)
    # plt.imshow(diff)
    # plt.title("diff")
    #
    # plt.subplot(3, 2, 5)
    # plt.imshow(mask)
```

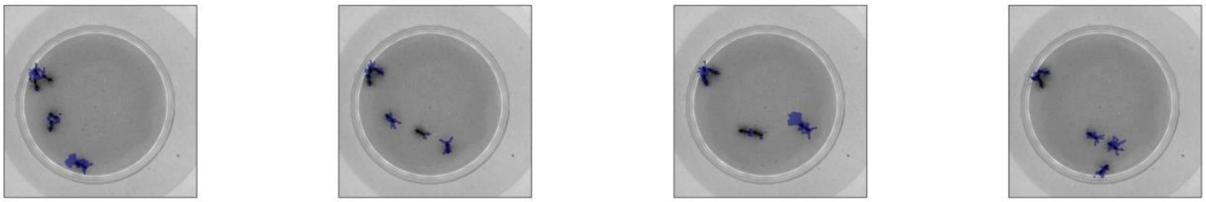
```
    # plt.title("mask")
    # plt.show()

    return mask.astype(bool)
```

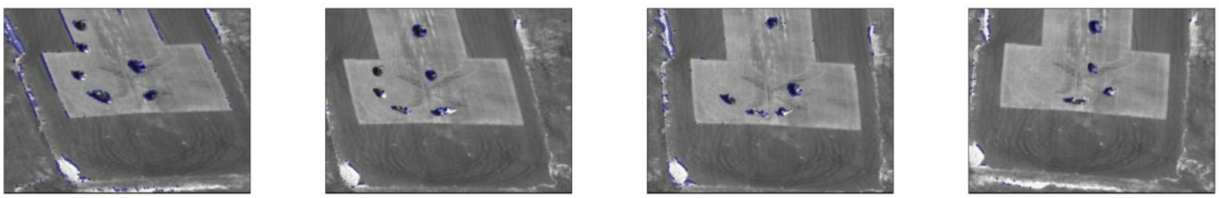
Q 2.3

The results of Lucas Kanade Tracking with motion detection:

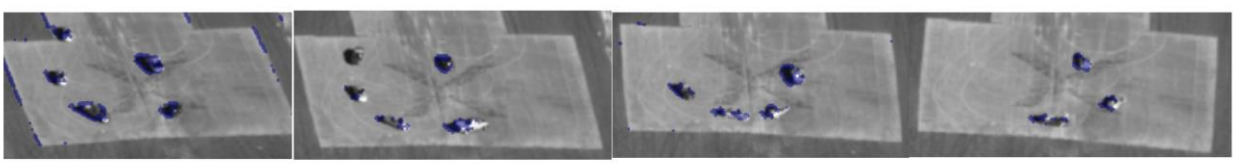
1) Ant



2) Aerial



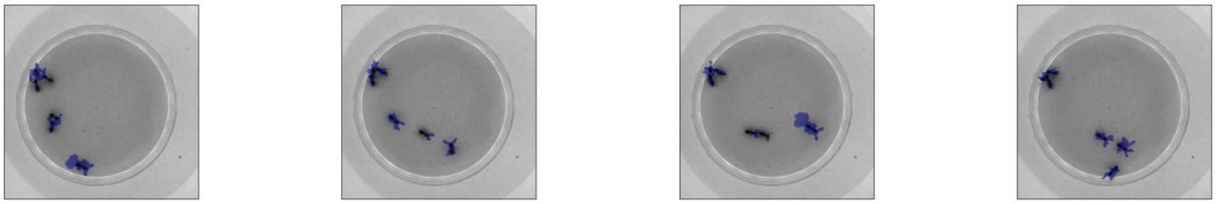
If I zoom in on the moving part,



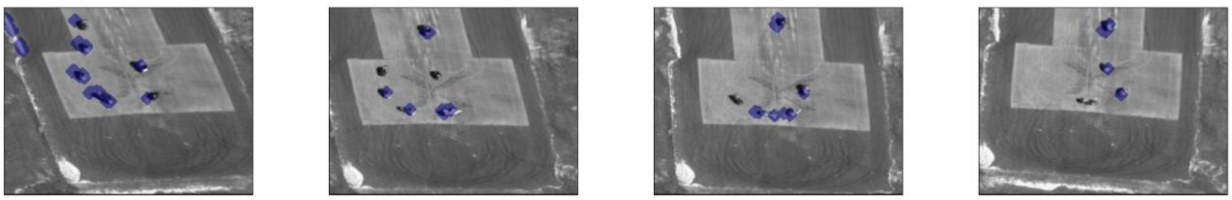
Q 3.1

The inverse composition is very fast compared to the original detection algorithm. It is because the Hessian and the gradient of the image is being calculated only once (outside the while loop). The computation time for aerial decreased from [6 minutes 1 second] to [6 seconds] and the ant sequence decreased from [16 seconds] to [6 seconds].

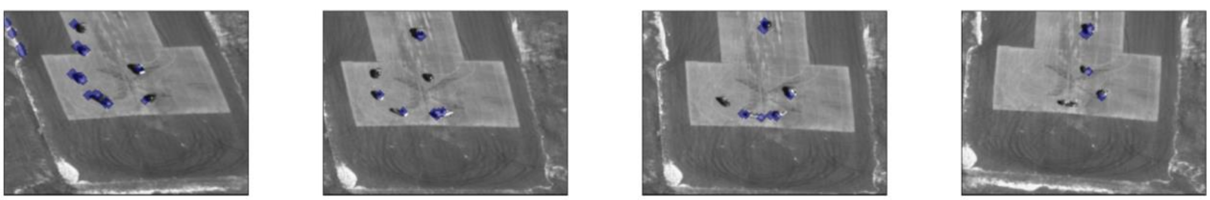
1) Ant



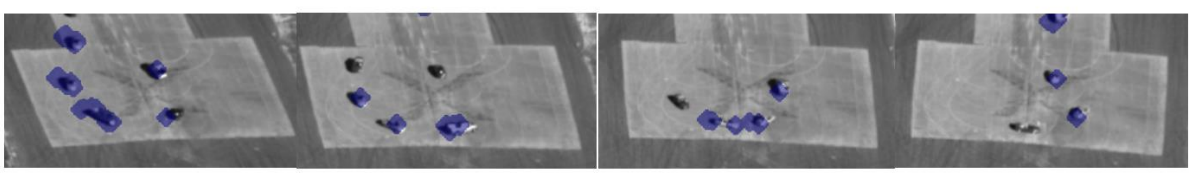
2) Aerial



(same code but different dilation to get better view of the moving parts)



If I zoom in,



Q 4.1

I used a video of someone dancing. It has a lot of movements and I added a white rectangular box to show that it can still detect after the object was partially occluded. In order to make the code work well, I had to take into consideration that the lighting may differ based on time. So I normalized the colors on the image before running any algorithms. Also, I used a gaussian blur on the image to get rid of noise since the video was not filmed in a structured environment.