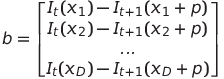
**16-720 Computer Vision: Homework 1 (Fall 2022)**

**Lucas-Kanade Tracking**

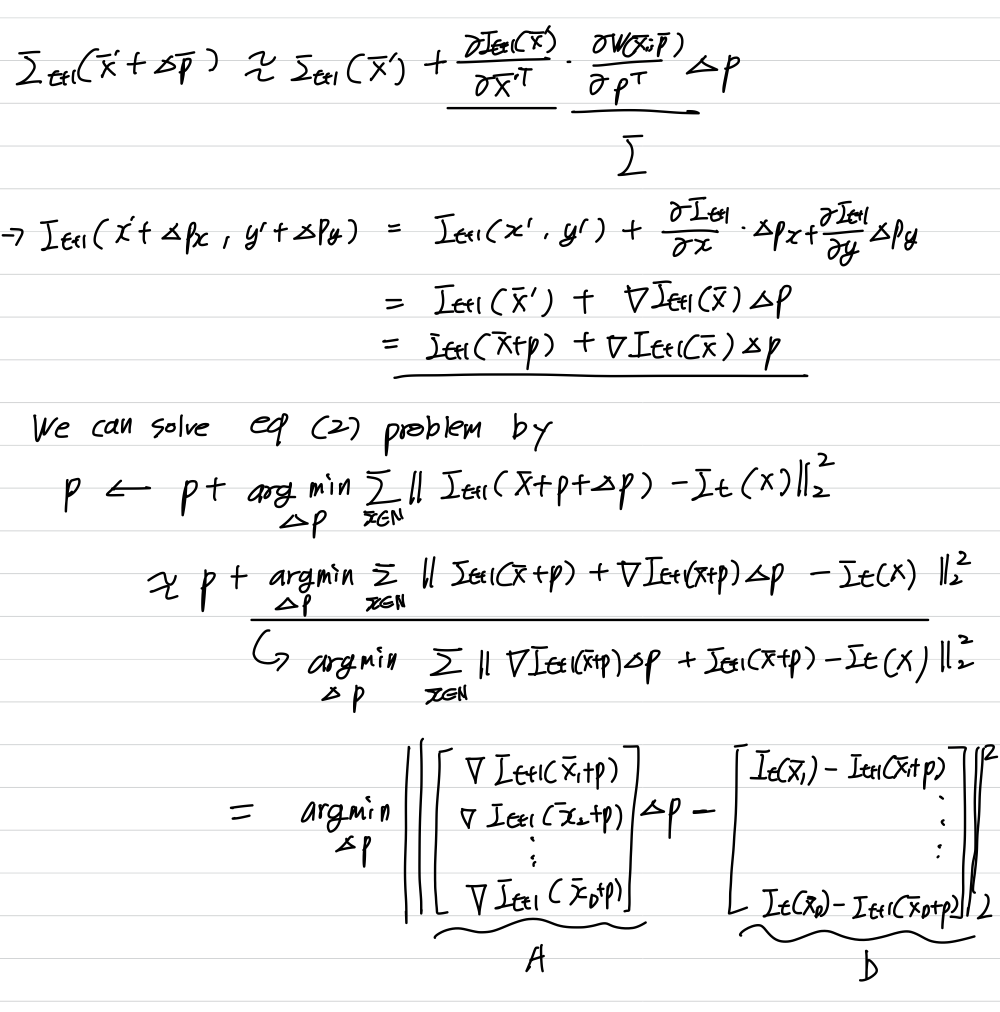
Haejoon Lee

**Q1.1**

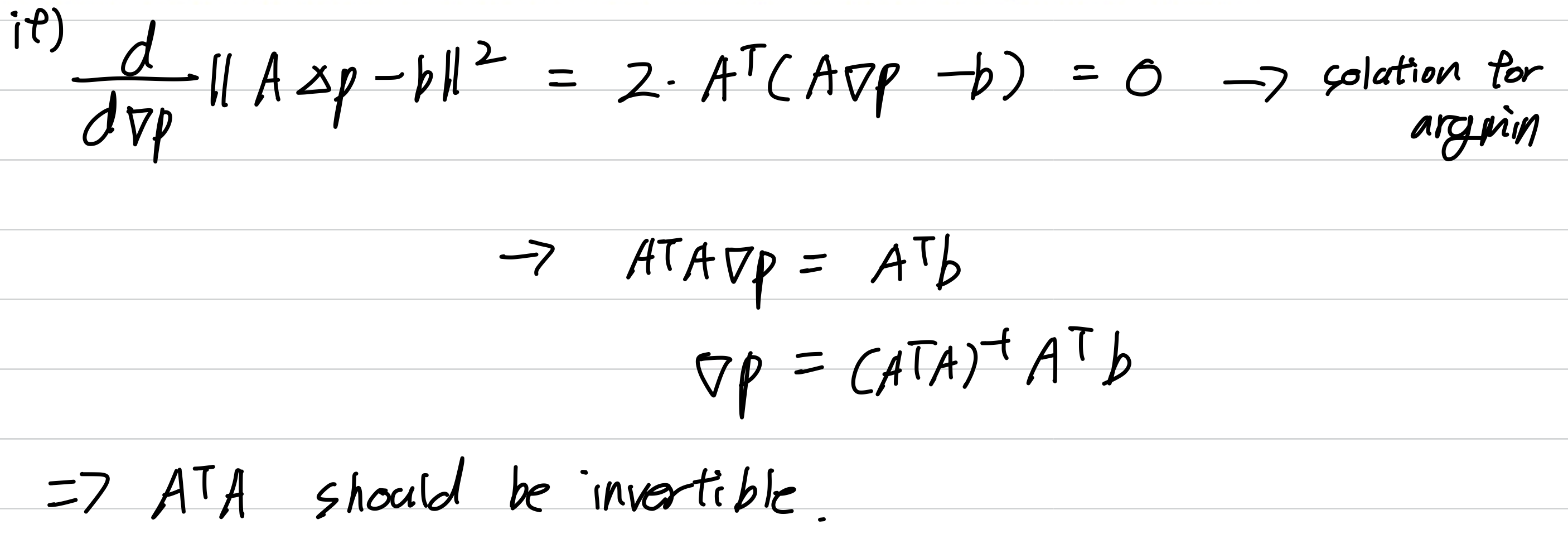
• What is (should a 2x2 matrix)?

텍스트이(가) 표시된 사진

자동 생성된 설명• What is A and b?



• What conditions must AT A meet so that a unique solution to ∆p can be found?



**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 ###################

H\_It, W\_It = It.shape

x1, y1, x2, y2 = rect[0], rect[1], rect[2], rect[3] #writen in image coordinates, not matrix

H\_rect, W\_rect = y2 - y1, x2 - x1

It1\_y, It1\_x = np.gradient(It1) #Gradient of current img

x\_axis\_rect = np.linspace(x1, x2, int(W\_rect))

y\_axis\_rect = np.linspace(y1, y2, int(H\_rect))

X\_grid\_rect, Y\_grid\_rect = np.meshgrid(x\_axis\_rect, y\_axis\_rect)

x = np.arange(0, W\_It, 1)

y = np.arange(0, H\_It, 1)

spline\_It = RectBivariateSpline(y, x, It) #Spline interpolation over a rectangular mesh

spline\_It1 = RectBivariateSpline(y, x, It1)

spline\_It1\_x = RectBivariateSpline(y, x, It1\_x)

spline\_It1\_y = RectBivariateSpline(y, x, It1\_y)

template\_rect = spline\_It.ev(Y\_grid\_rect, X\_grid\_rect) #Bring template\_rect from It

# Iterations for finding optimal dp

p = p0

dp = [[1000], [1000]] #Big dp for while loop

counter = 1

while np.square(dp).sum() > threshold and counter <= num\_iters:

# translate the rectangle

x1\_tr, y1\_tr, x2\_tr, y2\_tr = x1+p[0], y1+p[1], x2+p[0], y2+p[1]

x\_axis\_rect\_tr = np.linspace(x1\_tr, x2\_tr, int(W\_rect))

y\_axis\_rect\_tr = np.linspace(y1\_tr, y2\_tr, int(H\_rect))

X\_grid\_rect\_tr, Y\_grid\_rect\_tr = np.meshgrid(x\_axis\_rect\_tr, y\_axis\_rect\_tr)

spline\_It1\_tr = spline\_It1.ev(Y\_grid\_rect\_tr, X\_grid\_rect\_tr)

#A, b

It1\_rect\_tr\_x = spline\_It1\_x.ev(Y\_grid\_rect\_tr, X\_grid\_rect\_tr)

It1\_rect\_tr\_y = spline\_It1\_y.ev(Y\_grid\_rect\_tr, X\_grid\_rect\_tr)

A = np.vstack((It1\_rect\_tr\_x.ravel(),It1\_rect\_tr\_y.ravel())).T

b = (template\_rect - spline\_It1\_tr).reshape(-1, 1)

b = b.reshape(-1,1) #Columnize

#Solve argmin|Ax-b|^2 for finding dp

dp = np.linalg.inv(A.T@A) @ (A.T) @ b

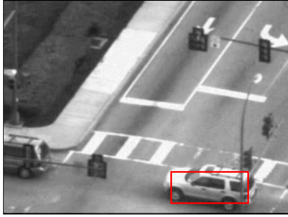
#update parameters

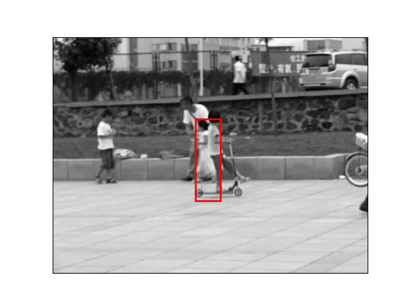
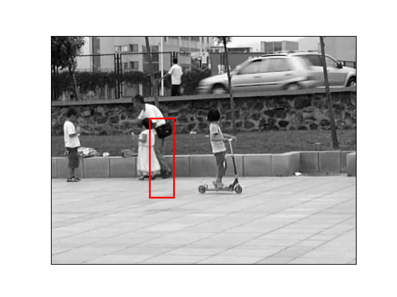
p[0] += dp[0,0]

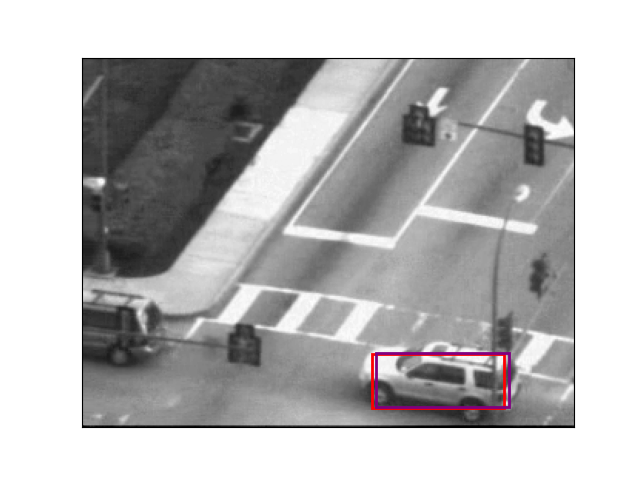
p[1] += dp[1,0]

counter += 1

return p

**Q1.3**

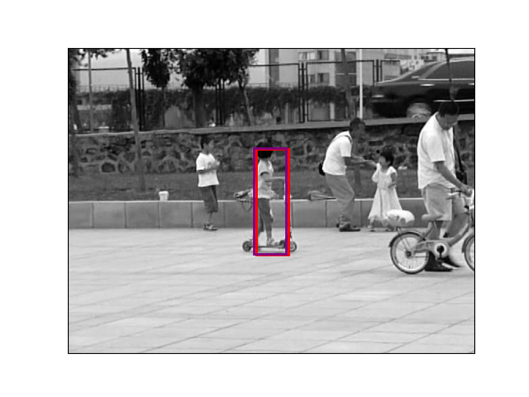


**텍스트이(가) 표시된 사진

자동 생성된 설명Q1.4**

텍스트, 운동경기, 스포츠이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명텍스트, 실외, 스케이트, 사람이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명\* Red: with template correction, Purple: original

def LucasKanade\_difftemp(It, It1, rect\_temp, rect\_cur, threshold, num\_iters, p0=np.zeros(2)):

"""

:param It: template image

:param It1: Current image

:param rect\_cur: Current position of the car (top left, bot right coordinates)

:param rect\_temp: Template rec position in the template image (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 ###################

H\_It, W\_It = It.shape

H\_It1, W\_It1 = It1.shape

x1, y1, x2, y2 = rect\_cur[0], rect\_cur[1], rect\_cur[2], rect\_cur[3] #writen in image coordinates, not matrix

xt1, yt1, xt2, yt2 = rect\_temp[0], rect\_temp[1], rect\_temp[2], rect\_temp[3] #writen in image coordinates, not matrix

H\_rect, W\_rect = y2 - y1, x2 - x1

H\_rect\_temp, W\_rect\_temp = yt2 - yt1, xt2 - xt1

It1\_y, It1\_x = np.gradient(It1) #Gradient of current img

# x\_axis\_rect = np.linspace(x1, x2, int(W\_rect))

# y\_axis\_rect = np.linspace(y1, y2, int(H\_rect))

# X\_grid\_rect, Y\_grid\_rect = np.meshgrid(x\_axis\_rect, y\_axis\_rect)

x\_axis\_rect\_temp = np.linspace(xt1, xt2, int(round(W\_rect\_temp)))

y\_axis\_rect\_temp = np.linspace(yt1, yt2, int(round(H\_rect\_temp)))

X\_grid\_rect\_temp, Y\_grid\_rect\_temp = np.meshgrid(x\_axis\_rect\_temp, y\_axis\_rect\_temp)

xt = np.arange(0, W\_It, 1)

yt = np.arange(0, H\_It, 1)

x = np.arange(0, W\_It1, 1)

y = np.arange(0, H\_It1, 1)

spline\_It = RectBivariateSpline(yt, xt, It) #Spline interpolation over a rectangular mesh

spline\_It1 = RectBivariateSpline(y, x, It1)

spline\_It1\_x = RectBivariateSpline(y, x, It1\_x)

spline\_It1\_y = RectBivariateSpline(y, x, It1\_y)

template\_rect = spline\_It.ev(Y\_grid\_rect\_temp, X\_grid\_rect\_temp) #Bring template\_rect from It

# Iterations for finding optimal dp

p = p0

dp = [[1000], [1000]] #Big dp for while loop

counter = 1

while np.square(dp).sum() > threshold and counter <= num\_iters:

# translate the rectangle

x1\_tr, y1\_tr, x2\_tr, y2\_tr = x1+p[0], y1+p[1], x2+p[0], y2+p[1]

x\_axis\_rect\_tr = np.linspace(x1\_tr, x2\_tr, int(round(W\_rect)))

y\_axis\_rect\_tr = np.linspace(y1\_tr, y2\_tr, int(round(H\_rect)))

X\_grid\_rect\_tr, Y\_grid\_rect\_tr = np.meshgrid(x\_axis\_rect\_tr, y\_axis\_rect\_tr)

spline\_It1\_tr = spline\_It1.ev(Y\_grid\_rect\_tr, X\_grid\_rect\_tr)

#A, b

It1\_rect\_tr\_x = spline\_It1\_x.ev(Y\_grid\_rect\_tr, X\_grid\_rect\_tr)

It1\_rect\_tr\_y = spline\_It1\_y.ev(Y\_grid\_rect\_tr, X\_grid\_rect\_tr)

A = np.vstack((It1\_rect\_tr\_x.ravel(),It1\_rect\_tr\_y.ravel())).T

b = (template\_rect - spline\_It1\_tr).reshape(-1, 1)

b = b.reshape(-1,1) #Columnize

#Solve argmin|Ax-b|^2 for finding dp

dp = np.linalg.inv(A.T@A) @ (A.T) @ b

#update parameters

p[0] += dp[0,0]

p[1] += dp[1,0]

counter += 1

return p

**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 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 ###################

H\_It, W\_It = It.shape

cor\_leftop = np.array([[0, 0, 1]])

cor\_rightbot = np.array([[H\_It, W\_It, 1]])

It1\_y, It1\_x = np.gradient(It1) #Gradient of current img

x = np.arange(0, W\_It, 1)

y = np.arange(0, H\_It, 1)

spline\_It = RectBivariateSpline(y, x, It) #Spline interpolation over a rectangular mesh

spline\_It1 = RectBivariateSpline(y, x, It1)

spline\_It1\_x = RectBivariateSpline(y, x, It1\_x)

spline\_It1\_y = RectBivariateSpline(y, x, It1\_y)

x\_axis = np.linspace(cor\_leftop[0,0], cor\_rightbot[0,0], int(round(W\_It)))

y\_axis = np.linspace(cor\_leftop[0,1], cor\_rightbot[0,1], int(round(H\_It)))

X\_grid, Y\_grid = np.meshgrid(x\_axis, y\_axis)

template = spline\_It.ev(Y\_grid, X\_grid)

# Iterations for finding optimal dp

dp = [[1000], [1000]] #Big dp for while loop

counter = 1

while np.square(dp).sum() > threshold and counter <= num\_iters:

# Warp Current Image

cor\_leftop\_wp = M@cor\_leftop.T

cor\_rightbot\_wp = M@cor\_rightbot.T

x\_axis\_wp = np.linspace(cor\_leftop\_wp[0,0], cor\_rightbot\_wp[0,0], int(round(W\_It)))

y\_axis\_wp = np.linspace(cor\_leftop\_wp[1,0], cor\_rightbot\_wp[1,0], int(round(H\_It)))

X\_grid\_wp, Y\_grid\_wp = np.meshgrid(x\_axis\_wp, y\_axis\_wp)

It1\_wp = spline\_It1.ev(Y\_grid\_wp, X\_grid\_wp)

#Get A and b

It1\_wp\_x = spline\_It1\_x.ev(Y\_grid\_wp, X\_grid\_wp)

It1\_wp\_y = spline\_It1\_y.ev(Y\_grid\_wp, X\_grid\_wp)

Del\_It1\_wp = np.vstack((It1\_wp\_x.ravel(),It1\_wp\_y.ravel())).T

A = np.zeros((H\_It\*W\_It, 6))

for i in range(H\_It):

for j in range(W\_It):

#I is (1,2) for each pixel

#Jacobiani is (2,6)for each pixel

Del\_It1\_wp\_point = np.array([Del\_It1\_wp[i\*W\_It+j]]).reshape(1,2)

jacob\_point = np.array([[j, i, 1, 0, 0, 0], [0, 0, 0, j, i, 1]])

A[i\*W\_It+j] = Del\_It1\_wp\_point @ jacob\_point

b = (template - It1\_wp).reshape(-1,1)

#Solve argmin|Ax-b|^2 for finding dp

dp = np.linalg.inv(A.T@A) @ (A.T) @ b

#Updating

M[0,0] += dp[0,0]

M[0,1] += dp[1,0]

M[0,2] += dp[2,0]

M[1,0] += dp[3,0]

M[1,1] += dp[4,0]

M[1,2] += dp[5,0]

counter += 1

return M

**Q2.2**

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)

image2\_wp = affine\_transform(image2, M, output\_shape = image1.shape)

image2\_wp = binary\_dilation(binary\_erosion(image2\_wp))

abs\_diff = np.abs(image1 - image2\_wp)

mask = (abs\_diff > tolerance)

return mask.astype(bool)

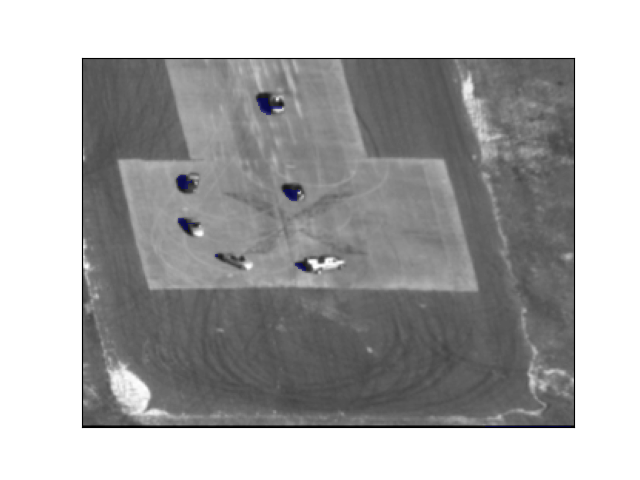
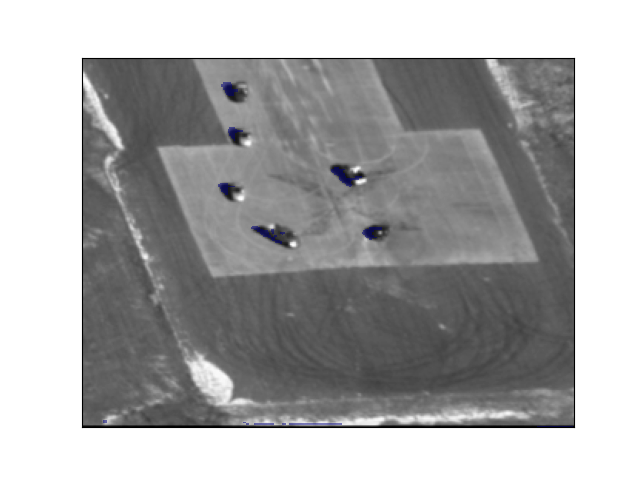
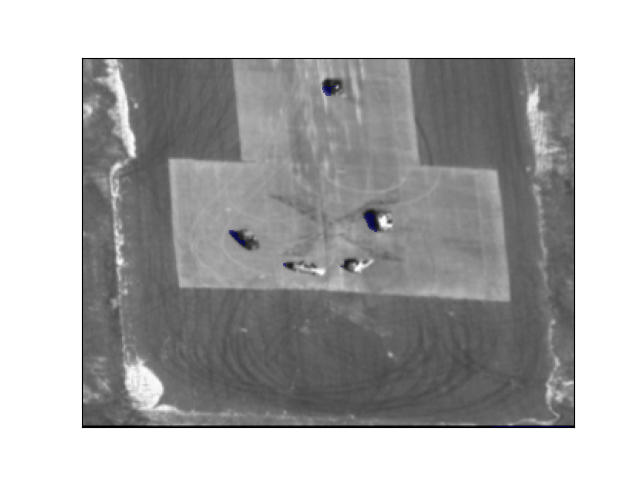
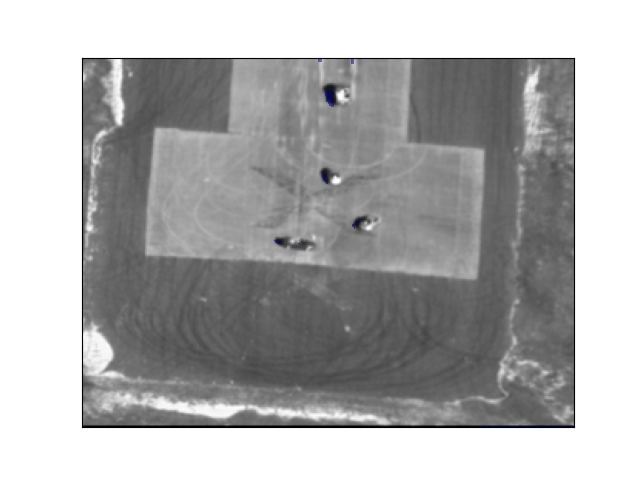
**실내, 하얀색이(가) 표시된 사진

자동 생성된 설명실내이(가) 표시된 사진

자동 생성된 설명실내, 하얀색이(가) 표시된 사진

자동 생성된 설명실내이(가) 표시된 사진

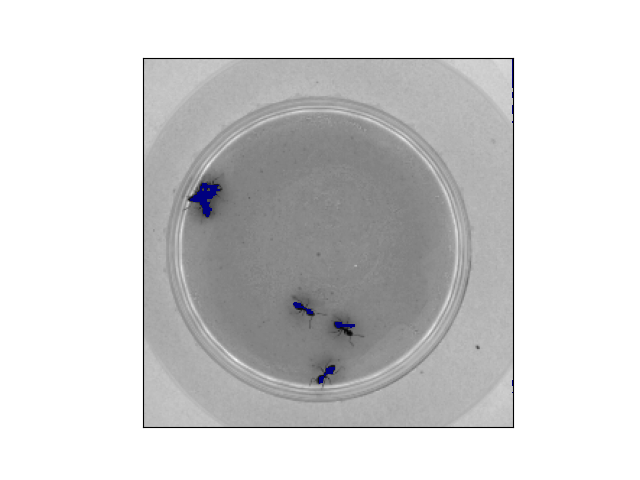
자동 생성된 설명Q2.3**

Masks here are very faint since of a small alpha value as bellow:

ax.imshow(np.ma.masked\_where(np.invert(mask), mask), cmap='jet', alpha=1)

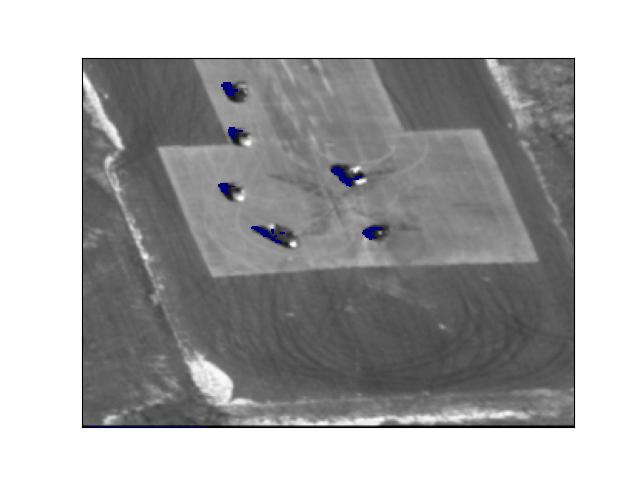
**Q3.1**

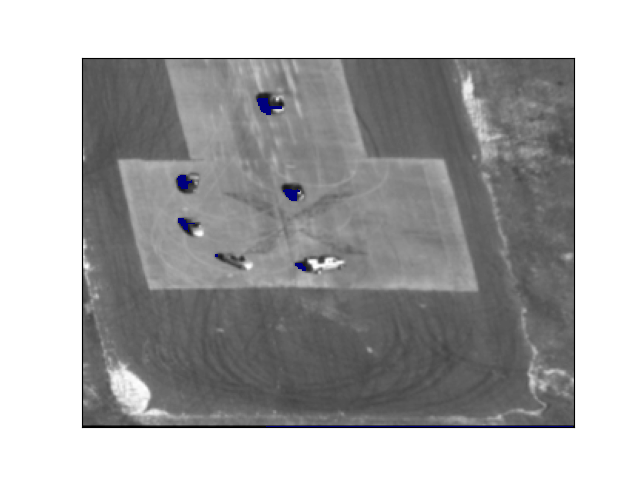
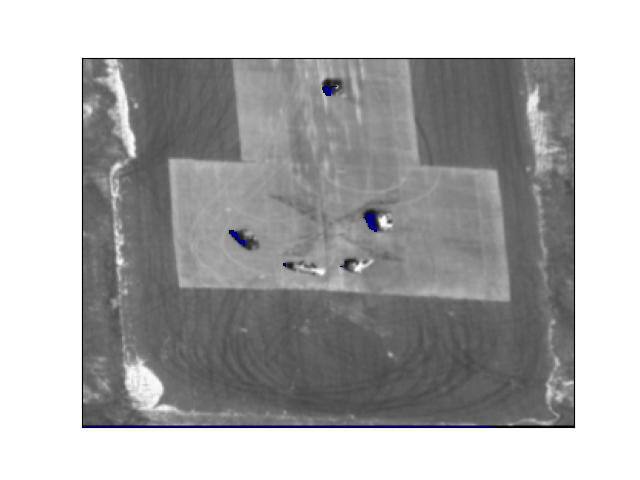
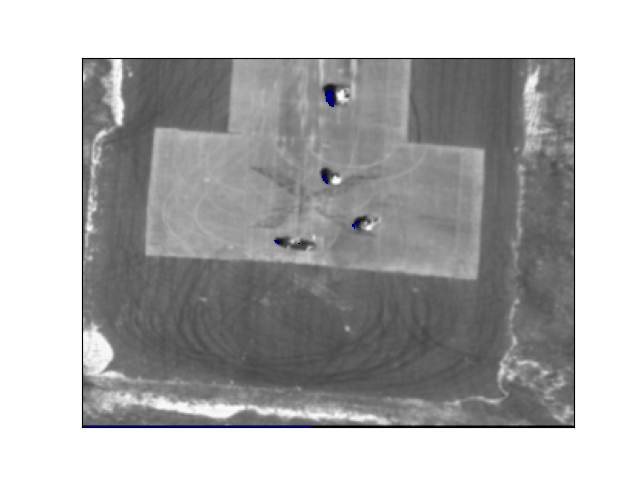
전자기기이(가) 표시된 사진

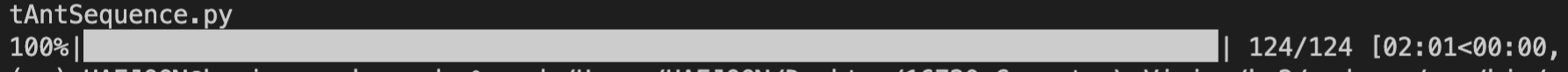
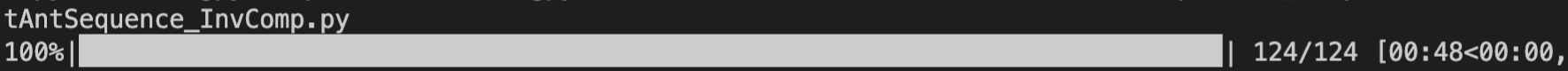
자동 생성된 설명실내이(가) 표시된 사진

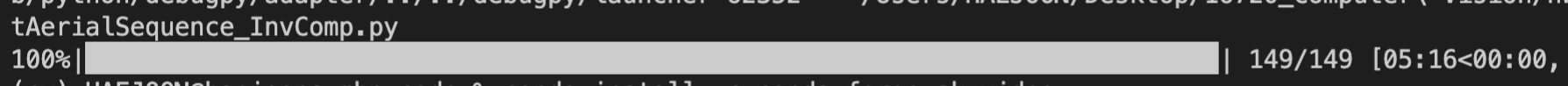
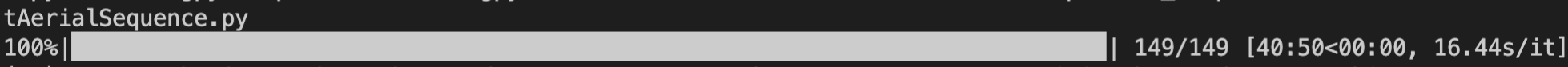
자동 생성된 설명실내이(가) 표시된 사진

자동 생성된 설명







After implementing Inverse Composition, the tracking time for ants reduced 2m 1s to 48s (x 2.52 faster)

In terms of tracking vehicles in Aerial sequence, the tracking time reduced from 40m 50s to 5m 16s (x 7.75 faster).

Inverse compositional approach is more computationally efficient because the template doesn’t change, so we can pre-compute term at outside of every iteration.

**Q4**

I tried to track motions in 10s my squat clean video. To perform more robust tracking with the illumination effect of the gym, I normalized each frame individually to [0, 1] using frame.min(), and frame.max().

cap = cv2.VideoCapture('../data/Clean.mp4')

frameCount = int(cap.get(cv2.CAP\_PROP\_FRAME\_COUNT))

frameWidth = int(cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH))

frameHeight = int(cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT))

buf = np.empty((frameHeight, frameWidth, int(frameCount/2)), np.dtype('float32'))

fc = 0

ret = True

i = 0

while (fc < frameCount and ret):

if (fc%2 == 0):

ret, frame = cap.read()

frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

frame\_min = frame.min(axis=(0, 1), keepdims=True)

frame\_max = frame.max(axis=(0, 1), keepdims=True)

frame = (frame - frame\_min) / (frame\_max - frame\_min)

buf[:, :, i] = frame

i += 1

fc += 1

cap.release()

np.save('../data/squatclean\_norm.npy', buf)

