Initialization

Run the following code to import the modules you'll need. After your finish the assignment, **remember to run all cells** and save the note book to your local machine as a PDF for gradescope submission.

In [ ]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
```

Download data

In this section we will download the data and setup the paths.

In [ ]:
```
# Download the data (Commented out because downloaded onto local machine)
# if not os.path.exists('/content/carseq.npy'):
#     !wget https://www.cs.cmu.edu/~deva/data/carseq.npy -O /content/carseq.npy
# if not os.path.exists('/content/girlseq.npy'):
#     !wget https://www.cs.cmu.edu/~deva/data/girlseq.npy -O /content/girlseq.npy
```

Q2.1: Theory Questions (5 points)

Please refer to the handout for the detailed questions.

Q2.1.1: What is $\frac{\partial \mathbf{W}(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T}$? (**Hint**: It should be a 2x2 matrix)

===== your answer here! =====

$$W(x;p) = x + p = \begin{bmatrix} x + p_x \\ y + p_y \end{bmatrix}$$

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

===== end of your answer =====

Q2.1.2: What is $\mathbf{A}$ and $\mathbf{b}$?

===== your answer here! =====

A = $\nabla I \, \frac{\partial \mathbf{W}}{\partial \mathbf{p}}$

b = $T(X) - I(W(x;p))$

x is Δp.

Ax = b

===== end of your answer =====

Q2.1.3 What conditions must $\mathbf{A}^T\mathbf{A}$ meet so that a unique solution to $\Delta\mathbf{p}$ can be found?

===== your answer here! =====

A^TA must be positive semi-definite.

===== end of your answer =====

Q2.2: Lucas-Kanade (20 points)

Make sure to comment your code and use proper names for your variables.

In [ ]:
```
from scipy.interpolate import RectBivariateSpline
from numpy.linalg import lstsq

def LucasKanade(It, It1, rect, threshold, num_iters, p0):
    """
    :param[np.array(H, W)] It   : Grayscale image at time t [float]
    :param[np.array(H, W)] It1  : Grayscale image at time t+1 [float]
    :param[np.array(4, 1)] rect : [x1 y1 x2 y2] coordinates of the rectangular temp
                                  where [x1, y1] is the top-left, and [x2, y2] is t
                                  [floats] that maybe fractional.
    :param[float] threshold     : If change in parameters is less than thresh, term
    :param[int] num_iters       : Maximum number of optimization iterations
    :param[np.array(2, 1)] p0   : Initial translation parameters [p_x0, p_y0] to ad
    :return[np.array(2, 1)] p   : Final translation parameters [p_x, p_y]
    """

    # ===== your code here! =====
    # Hint: Iterate over num_iters and for each iteration, construct a linear syste
    # Construct [A] by computing image gradients at (possibly fractional) pixel loc
    # We suggest using RectBivariateSpline from scipy.interpolate to interpolate pi
    # We suggest using lstsq from numpy.linalg to solve the linear system
    # Once you solve for [delta_p], add it to [p] (and move on to next iteration)
    #
    # HINT/WARNING:
```

```
    # RectBivariateSpline and Meshgrid use inconsistent defaults with respect to 'x
    # https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.RectBi
    # https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html

    p = p0

    px = p[0]
    py = p[1]

    i = 0
    delta_p_norm_square = threshold

    x1 = rect[0]
    y1 = rect[1]
    x2 = rect[2]
    y2 = rect[3]

    # T_t = It[y1:y2+1,x1:x2+1]
    # T_t_flat = np.reshape(T_t, (T_t.size,1))

    dWdp = np.eye(2)

    height = It.shape[0]
    width = It.shape[1]
    height_arr = np.arange(height)
    width_arr = np.arange(width)

    rbs_T_t = RectBivariateSpline(height_arr,width_arr,It)
    iX = np.arange(x1+px,x2+px+0.5,1)
    iY = np.arange(y1+py,y2+py+0.5,1)
    xgrid, ygrid = np.meshgrid(iX,iY,indexing="ij")
    T_t = rbs_T_t.ev(ygrid,xgrid).T
    T_t_flat = np.reshape(T_t, (T_t.size,1))

    rbs_IW = RectBivariateSpline(height_arr,width_arr,It1)
    gradientAllX = np.gradient(It1,axis=1)
    gradientAllY = np.gradient(It1,axis=0)
    rbs_gx = RectBivariateSpline(height_arr,width_arr,gradientAllX)
    rbs_gy = RectBivariateSpline(height_arr,width_arr,gradientAllY)

    while (i < num_iters) and (delta_p_norm_square >= threshold):
        px = p[0]
        py = p[1]

        iX = np.arange(x1+px,x2+px+0.5,1)
        iY = np.arange(y1+py,y2+py+0.5,1)
        xgrid, ygrid = np.meshgrid(iX,iY,indexing="ij")
        IW = rbs_IW.ev(ygrid,xgrid).T
        IW_flat = np.reshape(IW, (IW.size,1))

        gradientVecX = np.ravel(rbs_gx.ev(ygrid,xgrid).T)
        gradientVecY = np.ravel(rbs_gy.ev(ygrid,xgrid).T)
        gradientMatrix = np.vstack((gradientVecX,gradientVecY)).T # (pixPatch,2) gr

        A = gradientMatrix@dWdp
        b = T_t_flat-IW_flat
```

```
        #delta_p = np.linalg.lstsq(A.T@A,A.T@b,rcond=None)[0]
        delta_p = np.linalg.lstsq(A,b,rcond=None)[0]

        p = p + delta_p

        delta_p_norm_square = np.square(np.linalg.norm(delta_p,ord=2,keepdims=True)
        i = i + 1

    # plt.figure(2)
    # plt.imshow(T_t)
    # plt.figure(3)
    # plt.imshow(IW)
    # ===== End of code =====
    return p
```

Debug Q2.2

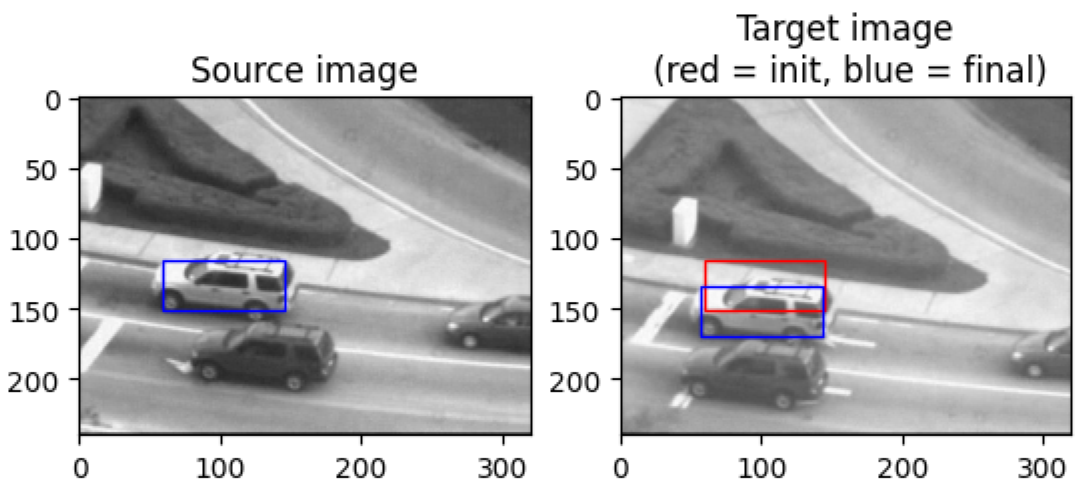
A few tips to debug your implementation:

- Feel free to use and modify the following snippet to debug your implementation. The snippet simply visualizes the translation resulting from running LK on a single frame. You should be able to see a slight shift in the template.
- You may also want to visualize the image gradients you compute within your LK implementation
- Plot iterations vs the norm of delta_p

In [ ]:
```
def draw_rect(rect,color):
    w = rect[2] - rect[0]
    h = rect[3] - rect[1]
    plt.gca().add_patch(patches.Rectangle((rect[0],rect[1]), w, h, linewidth=1, edg
```

In [ ]:
```
num_iters = 100
threshold = 0.01
seq = np.load("data\\carseq.npy")
rect = np.array([59, 116, 145, 151])
rect = np.reshape(rect, (rect.size,1))
It = seq[:,:,0]

# Source frame
plt.figure()
plt.subplot(1,2,1)
plt.imshow(It, cmap='gray')
plt.title('Source image')
draw_rect(rect,'b')

# Target frame + LK
It1  = seq[:,:, 20]
plt.subplot(1,2,2)
```

```python
plt.imshow(It1, cmap='gray')
plt.title('Target image\n (red = init, blue = final)')
p = LucasKanade(It, It1, rect, threshold, num_iters, np.zeros((2,1)))

rect_t1 = (rect + np.concatenate((p,p)))

draw_rect(rect,'r')
draw_rect(rect_t1,'b')
```



Q2.3: Tracking with template update (15 points)

In [ ]:
```python
def TrackSequence(seq, rect, num_iters, threshold):
    """
    :param seq       : (H, W, T), sequence of frames
    :param rect      : (4, 1), coordinates of template in the initial frame. top-le
    :param num_iters : int, number of iterations for running the optimization
    :param threshold : float, threshold for terminating the LK optimization
    :return: rects   : (T, 4) tracked rectangles for each frame
    """
    H, W, N = seq.shape

    rects = np.zeros((N,4))
    It = seq[:,:,0]
    pNow = np.zeros((2,1))
    #rect = np.reshape(rect, (4,1))

    # Iterate over the car sequence and track the car
    for i in range(seq.shape[2]):

        # ===== your code here! =====
        # TODO: add your code track the object of interest in the sequence
        if (i == seq.shape[2]-1):
            It = seq[:,:,i]
            It1 = seq[:,:,i]
        else:
            It = seq[:,:,i]
            It1 = seq[:,:,i+1]
        pNow = LucasKanade(It,It1,rect,threshold,num_iters,pNow)
        rects[i,:] = (np.copy(rect) + np.concatenate((pNow,pNow))).T
```

```
        # ===== End of code =====

    #rects = np.array(rects)
    assert rects.shape == (N, 4), f"Your output sequence {rects.shape} is not ({N}x
    return rects
```

Q2.3 (a) - Track Car Sequence

Run the following snippets. If you have implemented LucasKanade and TrackSequence function correctly, you should see the box tracking the car accurately. Please note that the tracking might drift slightly towards the end, and that is entirely normal.

Feel free to play with these snippets of code by playing with the parameters.

In [ ]:
```
def visualize_track(seq,rects,frames,rect):
    # Visualize tracks on an image sequence for a select number of frames
    plt.figure(figsize=(15,15))
    for i in range(len(frames)):
        idx = frames[i]
        frame = seq[:, :, idx]
        plt.subplot(1,len(frames),i+1)
        plt.imshow(frame, cmap='gray')
        plt.axis('off')
        draw_rect(rects[idx],'b')
```

In [ ]:
```
seq = np.load("data\\carseq.npy")
rect = np.array([59, 116, 145, 151])
rect = np.reshape(rect, (rect.size,1))

# NOTE: feel free to play with these parameters
num_iters = 10000
threshold = 0.01

rects = TrackSequence(seq, rect, num_iters, threshold)

visualize_track(seq,rects,[0, 79, 159, 279, 409],rect)
#scale = 20
#visualize_track(seq,rects,[0*scale, 1*scale, 2*scale, 3*scale, 4*scale],rect) # Ch
```

Q2.3 (b) - Track Girl Sequence

Same as the car sequence.

In [ ]:
```
# Loads the squence
seq = np.load("data\\girlseq.npy")
rect = np.array([280, 152, 330, 318])
```

```
rect = np.reshape(rect, (rect.size,1))

# NOTE: feel free to play with these parameters
num_iters = 10000
threshold = 0.01

rects = TrackSequence(seq, rect, num_iters, threshold)

visualize_track(seq,rects,[0, 14, 34, 64, 84],rect)
```

Initialization

Run the following code to import the modules you'll need. After your finish the assignment, **remember to run all cells** and save the note book to your local machine as a PDF for gradescope submission.

In [ ]:
```
import time
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
```

Download data

In this section we will download the data and setup the paths.

In [ ]:
```
# Download the data
# if not os.path.exists('/content/aerialseq.npy'):
#     !wget https://www.cs.cmu.edu/~deva/data/aerialseq.npy -O /content/aerialseq.n
# if not os.path.exists('/content/antseq.npy'):
#     !wget https://www.cs.cmu.edu/~deva/data/antseq.npy -O /content/antseq.npy
```

Q3: Affine Motion Subtraction

Q3.1: Dominant Motion Estimation (15 points)

In [ ]:
```
from scipy.interpolate import RectBivariateSpline

def LucasKanadeAffine(It, It1, threshold, num_iters):
    """
    :param It        : (H, W), current image
    :param It1       : (H, W), next image
    :param threshold : (float), if the length of dp < threshold, terminate the opti
    :param num_iters : (int), number of iterations for running the optimization

    :return: M       : (2, 3) The affine transform matrix
    """
    # Initial M
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])

    p = np.zeros((6,1))
    delta_p = np.zeros((6,1))
    delta_p_length = threshold

    H = It.shape[0]
```

```
    W = It.shape[1]

    xVals = np.arange(0,W,1)
    yVals = np.arange(0,H,1)

    yGrid, xGrid = np.meshgrid(yVals,xVals,indexing="ij")
    xGrid_flat = xGrid.ravel()
    yGrid_flat = yGrid.ravel()
    oneGrid_flat = np.zeros_like(xGrid_flat) + 1
    points_flat = np.vstack((xGrid_flat,yGrid_flat,oneGrid_flat)) # [x_list;y_list;

    T_t = np.copy(It)
    T_t_flat = T_t.ravel()

    rbs_IW = RectBivariateSpline(yVals,xVals,It1)
    gradientX = np.ravel(np.gradient(It1,axis=1))
    gradientY = np.ravel(np.gradient(It1,axis=0))

    i = 0
    while (i < num_iters) and (delta_p_length >= threshold):
        p1 = p[0,0]
        p2 = p[1,0]
        p3 = p[2,0]
        p4 = p[3,0]
        p5 = p[4,0]
        p6 = p[5,0]
        M = np.array([[1 + p1, p2, p3], [p4, 1 + p5, p6]])

        points_flat_tf = M@points_flat
        xGrid_tf_flat = points_flat_tf[0,:]
        yGrid_tf_flat = points_flat_tf[1,:]
        xGrid_tf = np.reshape(xGrid_tf_flat,(H,W))
        yGrid_tf = np.reshape(yGrid_tf_flat,(H,W))
        IW = rbs_IW.ev(xGrid_tf,yGrid_tf)
        IW_flat = IW.ravel()

        maskX = np.logical_and(xGrid_tf_flat<W,xGrid_tf_flat>=0)
        maskY = np.logical_and(yGrid_tf_flat<H,yGrid_tf_flat>=0)
        maskAll = maskX*maskY
        x_flat_short = xGrid_tf_flat[maskAll]
        y_flat_short = yGrid_tf_flat[maskAll]
        IW_flat_short = IW_flat[maskAll]
        T_t_flat_short = T_t_flat[maskAll]

        b = (T_t_flat_short - IW_flat_short).T # (N,1)
        #b = (It1.ravel()[maskAll] - IW_flat_short).T # (N,1)

        gradientX_short = gradientX[maskAll]
        gradientY_short = gradientY[maskAll]
        gradientMatrix = np.vstack((gradientX_short,gradientY_short)).T # (N,6)

        N = gradientX_short.size

        A = np.zeros((N,6))
        A[:,0] = gradientX_short*x_flat_short
        A[:,1] = gradientY_short*x_flat_short
```

```
        A[:,2] = gradientX_short*y_flat_short
        A[:,3] = gradientY_short*y_flat_short
        A[:,4] = gradientX_short*1
        A[:,5] = gradientY_short*1

        delta_p = np.linalg.lstsq(A,b,rcond=None)[0]

        # A = gradientMatrix*dWdp # (1,2)(2,6) = (1,6) --> (N,6)
        # b = T_t - IW # (N,1)

        # Ax = b

        i = i + 1
        delta_p_length = np.square(np.linalg.norm(delta_p,ord=2))
        p = p + delta_p


    p1 = p[0,0]
    p2 = p[1,0]
    p3 = p[2,0]
    p4 = p[3,0]
    p5 = p[4,0]
    p6 = p[5,0]
    M = np.array([[1 + p1, p2, p3], [p4, 1 + p5, p6]])
    # print(M)
    # plt.figure(4)
    # plt.imshow(T_t)
    # plt.figure(5)
    # plt.imshow(IW)
    # print(T_t.shape)
    # print(IW.shape)

    # print(points_flat.shape)

    return M
```

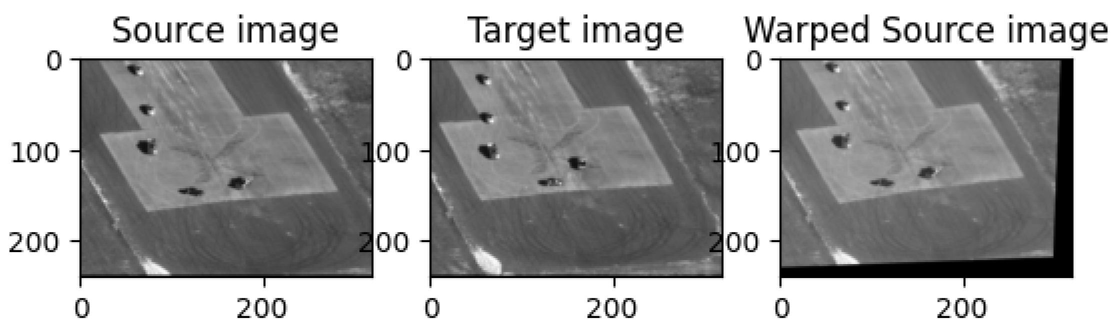
Debug Q3.1

Feel free to use and modify the following snippet to debug your implementation. The snippet simply visualizes the translation resulting from running LK on a single frame. When you warp the source frame using the obtained transformation matrix, it should resemble the target frame.

In [ ]:
```
import cv2

num_iters = 100 # Defaults to 100
threshold = 0.01
seq = np.load("data\\aerialseq.npy")
It = seq[:,:,0]
It1 = seq[:,:,10]

# Source frame
plt.figure()
```

```
plt.subplot(1,3,1)
plt.imshow(It, cmap='gray')
plt.title('Source image')

# Target frame
plt.subplot(1,3,2)
plt.imshow(It1, cmap='gray')
plt.title('Target image')

# Warped source frame
M = LucasKanadeAffine(It, It1, threshold, num_iters)
warped_It = cv2.warpAffine(It, M,(It.shape[1],It.shape[0]))
plt.subplot(1,3,3)
plt.imshow(warped_It, cmap='gray')
plt.title('Warped Source image')

# plt.figure()
# plt.imshow(It1-warped_It)
```

Out[ ]: `Text(0.5, 1.0, 'Warped Source image')`



Q3.2: Moving Object Detection (10 points)

In [ ]:
```
import numpy as np
from scipy.ndimage import binary_erosion
from scipy.ndimage import binary_dilation
from scipy.ndimage import affine_transform
import scipy.ndimage
import cv2

def SubtractDominantMotion(It, It1, num_iters, threshold, tolerance):
    """
    :param It        : (H, W), current image
    :param It1       : (H, W), next image
    :param num_iters : (int), number of iterations for running the optimization
    :param threshold : (float), if the length of dp < threshold, terminate the opti
    :param tolerance : (float), binary threshold of intensity difference when compu
    :return: mask    : (H, W), the mask of the moved object
    """
    mask = np.ones(It.shape, dtype=bool)

    # ===== your code here! =====
    M = LucasKanadeAffine(It, It1, threshold, num_iters)
```

```
    warped_It = cv2.warpAffine(It, M,(It.shape[1],It.shape[0]))
    absDiff = np.abs(It1 - warped_It)
    mask1 = absDiff > tolerance
    mask2 = absDiff < np.max(absDiff)*1.0
    mask = mask1*mask2
    # ===== End of code =====

    return mask
```

Q3.3: Tracking with affine motion (10 points)

In [ ]:
```
from tqdm import tqdm

def TrackSequenceAffineMotion(seq, num_iters, threshold, tolerance):
    """
    :param seq       : (H, W, T), sequence of frames
    :param num_iters : int, number of iterations for running the optimization
    :param threshold : float, if the length of dp < threshold, terminate the optimi
    :param tolerance : (float), binary threshold of intensity difference when compu
    :return: masks   : (T, 4) moved objects for each frame
    """
    H, W, N = seq.shape

    masks = []
    It = seq[:,:,0]

    # ===== your code here! =====
    for i in tqdm(range(1, seq.shape[2])):
        if (i == seq.shape[2]-1):
            It = seq[:,:,i]
            It1 = seq[:,:,i]
        else:
            It = seq[:,:,i]
            It1 = seq[:,:,i+1]
        mask = SubtractDominantMotion(It, It1, num_iters, threshold, tolerance)
        masks.append(mask)
        # print(mask)
    # ===== End of code =====
    masks = np.stack(masks, axis=2)
    return masks
```

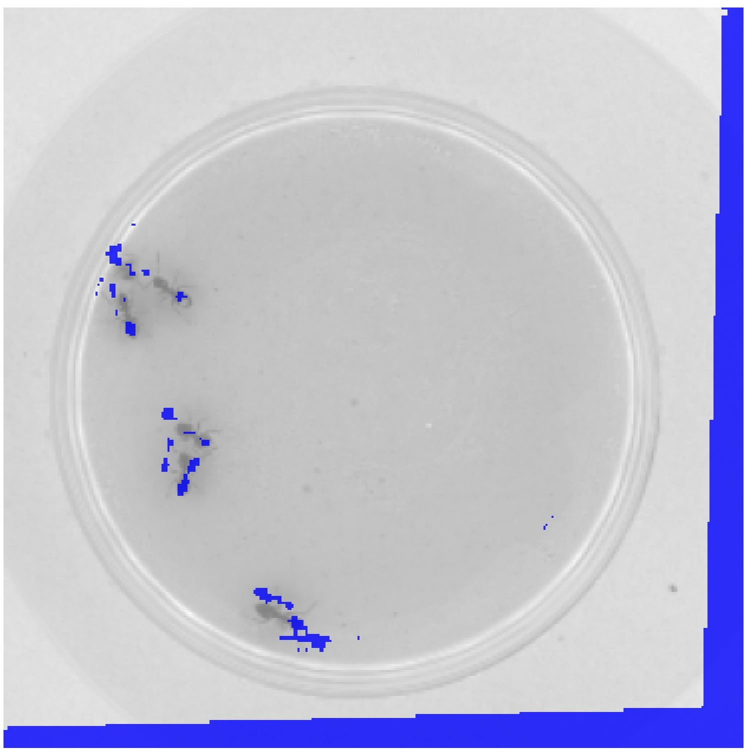
Q3.3 (a) - Track Ant Sequence

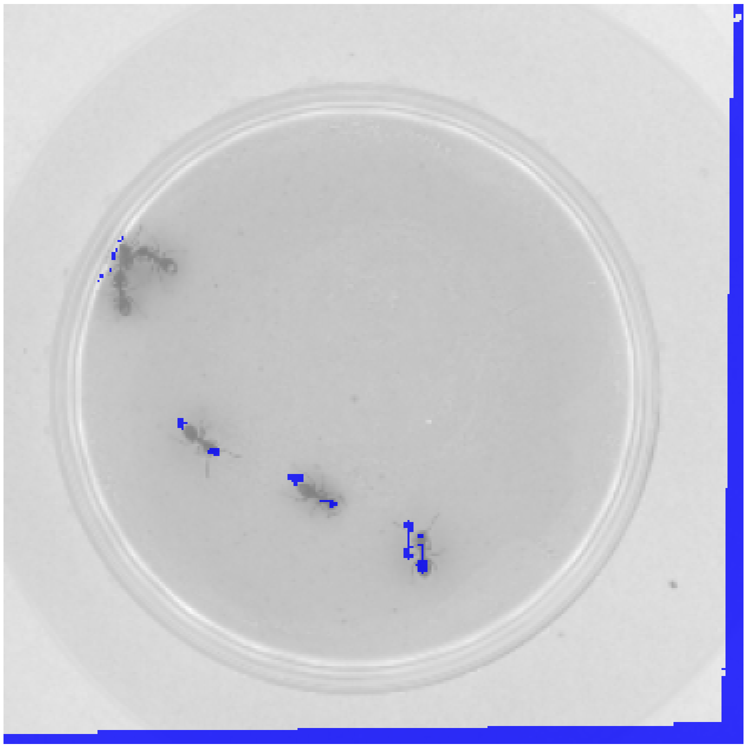
In [ ]:
```
seq = np.load("data\\antseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 20 #
threshold = 0.01
tolerance = 0.2

tic = time.time()
masks = TrackSequenceAffineMotion(seq, num_iters, threshold, tolerance)
```

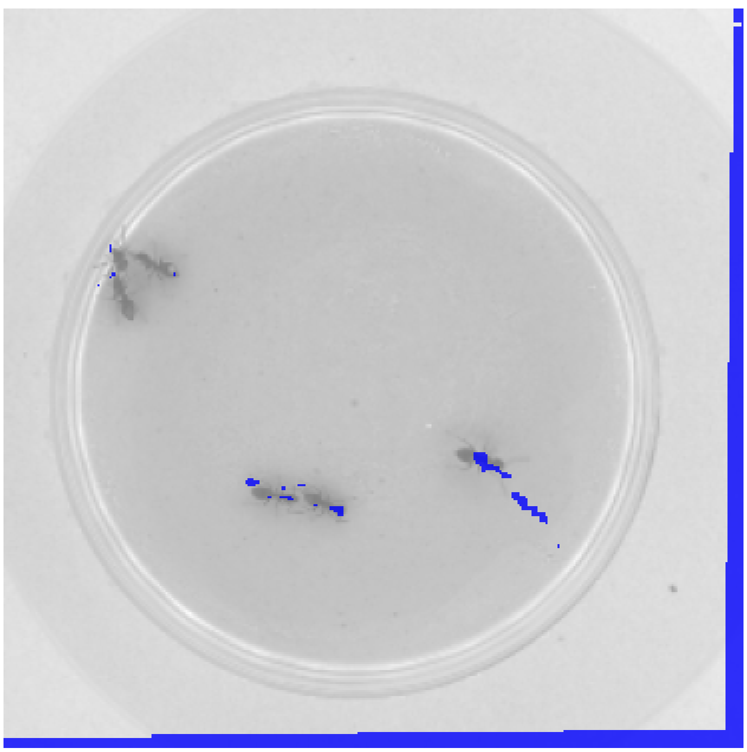
```
toc = time.time()
print('\nAnt Sequence takes %f seconds' % (toc - tic))
```

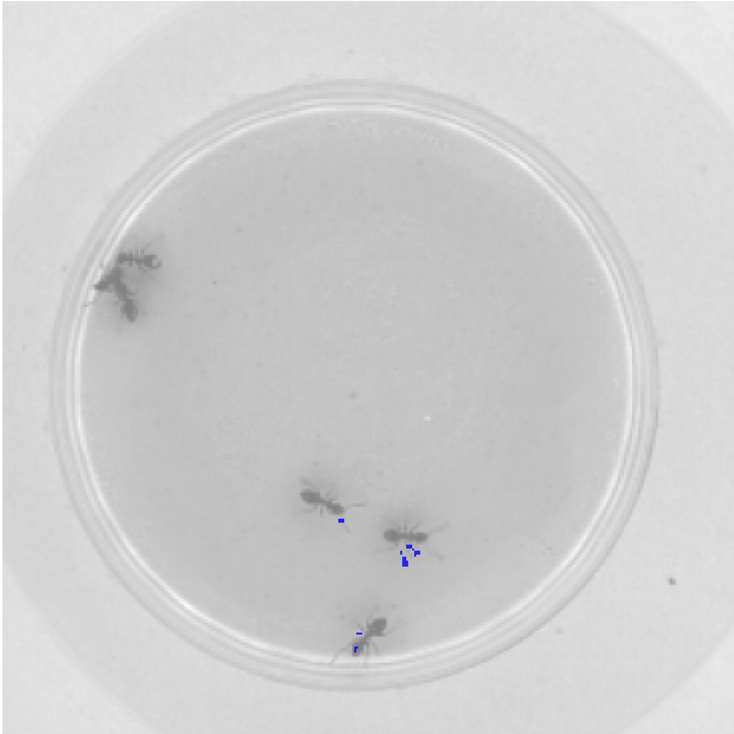
```
  0%|          | 0/124 [00:00<?, ?it/s]
100%|██████████| 124/124 [00:56<00:00,  2.21it/s]
Ant Sequence takes 56.048555 seconds
```

In [ ]:
```
frames_to_save = [29, 59, 89, 119]

# TODO: visualize
for idx in frames_to_save:
    frame = seq[:, :, idx]
    mask = masks[:, :, idx]
    # print(masks[0,0,0])

    plt.figure()
    plt.imshow(frame, cmap="gray", alpha=0.5)
    plt.imshow(np.ma.masked_where(np.invert(mask), mask), cmap='winter', alpha=0.8)
    plt.axis('off')
```







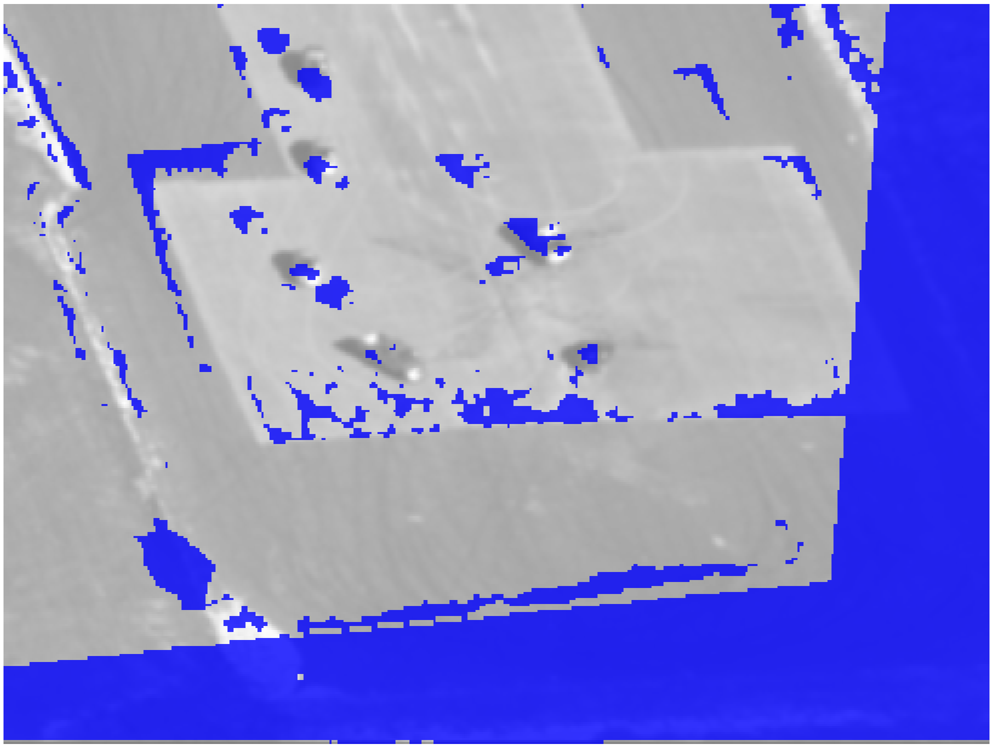


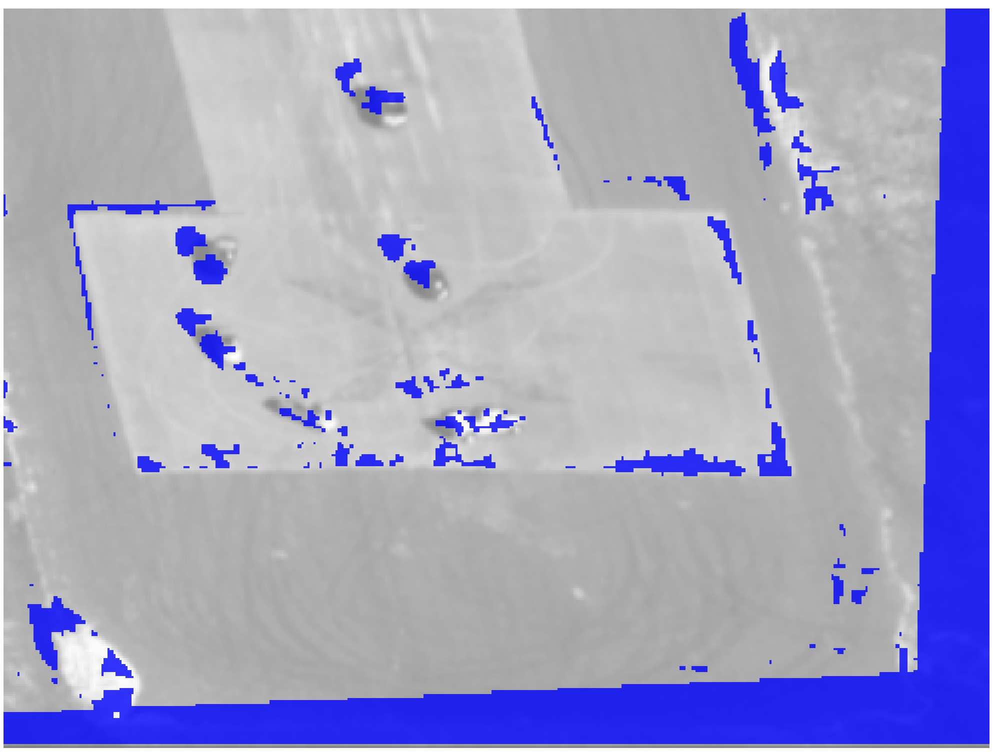
Q3.3 (b) - Track Aerial Sequence

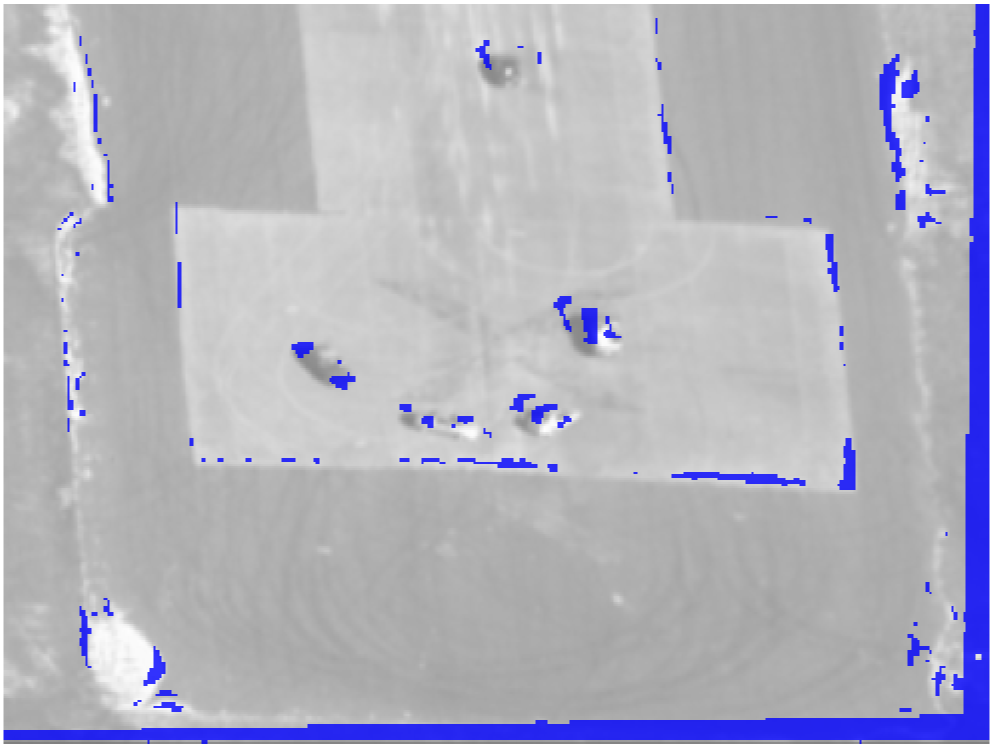
In [ ]:
```
seq = np.load("data\\aerialseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 20
threshold = 0.01
tolerance = 0.2

tic = time.time()
masks = TrackSequenceAffineMotion(seq, num_iters, threshold, tolerance)
toc = time.time()
print('\nAerial Sequence takes %f seconds' % (toc - tic))
```

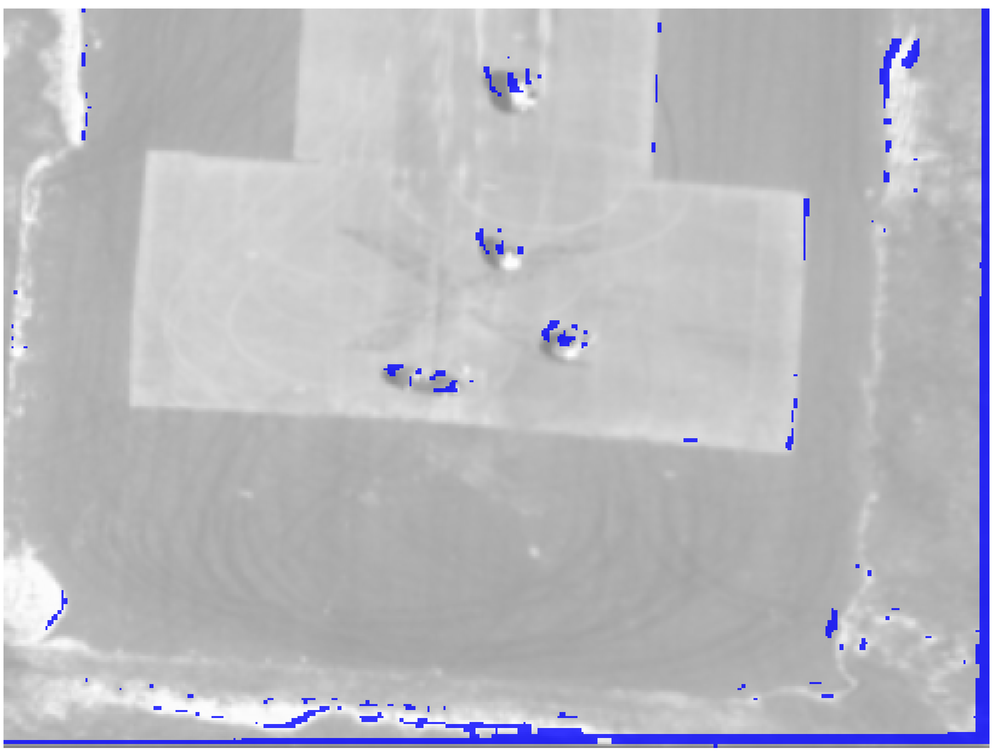
```
  0%|          | 0/149 [00:00<?, ?it/s]
100%|██████████| 149/149 [01:19<00:00,  1.88it/s]
Aerial Sequence takes 79.262299 seconds
```

In [ ]:
```
frames_to_save = [29, 59, 89, 119]

# TODO: visualize
for idx in frames_to_save:
    frame = seq[:, :, idx]
    mask = masks[:, :, idx]

    plt.figure()
    plt.imshow(frame, cmap="gray", alpha=0.5)
    plt.imshow(np.ma.masked_where(np.invert(mask), mask), cmap='winter', alpha=0.8)
    plt.axis('off')
```









Initialization

Run the following code to import the modules you'll need. After your finish the assignment, **remember to run all cells** and save the note book to your local machine as a PDF for gradescope submission.

In [ ]:
```
import time
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
```

Download data

In this section we will download the data and setup the paths.

In [ ]:
```
# Download the data
# if not os.path.exists('/content/aerialseq.npy'):
#     !wget https://www.cs.cmu.edu/~deva/data/aerialseq.npy -O /content/aerialseq.n
# if not os.path.exists('/content/antseq.npy'):
#     !wget https://www.cs.cmu.edu/~deva/data/antseq.npy -O /content/antseq.npy
```

Q4: Efficient Tracking

Q4.1: Inverse Composition (15 points)

In [ ]:
```
from scipy.interpolate import RectBivariateSpline

def InverseCompositionAffine(It, It1, threshold, num_iters):
    """
    :param It        : (H, W), current image
    :param It1       : (H, W), next image
    :param threshold : (float), if the length of dp < threshold, terminate the opti
    :param num_iters : (int), number of iterations for running the optimization

    :return: M       : (2, 3) The affine transform matrix
    """
    # Initial M
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])

    # ===== your code here! =====
    i = 0
    delta_p_length = threshold
    p = np.zeros((6,1))
```

```
    H = It.shape[0]
    W = It.shape[1]

    xVals = np.arange(0,W,1)
    yVals = np.arange(0,H,1)

    p1 = p[0,0]
    p2 = p[1,0]
    p3 = p[2,0]
    p4 = p[3,0]
    p5 = p[4,0]
    p6 = p[5,0]
    M = np.array([[1 + p1, p2, p3], [p4, 1 + p5, p6], [0, 0, 1]]) # (3,3) and there
    Minv = np.linalg.inv(M)

    yGrid, xGrid = np.meshgrid(yVals,xVals,indexing="ij")
    xGrid_flat = xGrid.ravel()
    yGrid_flat = yGrid.ravel()
    oneGrid_flat = np.zeros_like(xGrid_flat) + 1
    points_flat = np.vstack((xGrid_flat,yGrid_flat,oneGrid_flat)) # [x_list;y_list;

    points_flat_tf = M@points_flat
    xGrid_tf_flat = points_flat_tf[0,:]
    yGrid_tf_flat = points_flat_tf[1,:]
    xGrid_tf = np.reshape(xGrid_tf_flat,(H,W))
    yGrid_tf = np.reshape(yGrid_tf_flat,(H,W))

    rbs_TW = RectBivariateSpline(yVals,xVals,It)
    TW = rbs_TW.ev(xGrid_tf,yGrid_tf)

    T_t = np.copy(It)
    T_t_flat = T_t.ravel()

    gradientT_X = np.gradient(T_t,axis=1)
    gradientT_Y = np.gradient(T_t,axis=0)
    gradientT_X_flat = gradientT_X.ravel()
    gradientT_Y_flat = gradientT_Y.ravel()
    gradientT_all = np.vstack((gradientT_X_flat,gradientT_Y_flat))

    xVals = np.arange(0,W,1)
    yVals = np.arange(0,H,1)

    yGrid, xGrid = np.meshgrid(yVals,xVals,indexing="ij")
    xGrid_flat = xGrid.ravel()
    yGrid_flat = yGrid.ravel()

    maskX = np.logical_and(xGrid_tf_flat<W,xGrid_tf_flat>=0)
    maskY = np.logical_and(yGrid_tf_flat<H,yGrid_tf_flat>=0)
    maskAll = maskX*maskY

    N = xGrid_flat.size

    gradientX_zeros = np.copy(gradientT_X_flat)
    gradientX_zeros[~maskAll] = 0
    gradientY_zeros = np.copy(gradientT_Y_flat)
    gradientY_zeros[~maskAll] = 0
```

```
    A = np.zeros((N,6))
    A[:,0] = gradientX_zeros*xGrid_flat
    A[:,1] = gradientY_zeros*xGrid_flat
    A[:,2] = gradientX_zeros*yGrid_flat
    A[:,3] = gradientY_zeros*yGrid_flat
    A[:,4] = gradientX_zeros*1
    A[:,5] = gradientY_zeros*1


    while (i < num_iters) and (delta_p_length >= threshold):

        points_flat_tf = M@points_flat
        xGrid_tf_flat = points_flat_tf[0,:]
        yGrid_tf_flat = points_flat_tf[1,:]
        xGrid_tf = np.reshape(xGrid_tf_flat,(H,W))
        yGrid_tf = np.reshape(yGrid_tf_flat,(H,W))
        TW = rbs_TW.ev(xGrid_tf,yGrid_tf)
        TW_flat = TW.ravel()

        b = (T_t_flat - TW_flat)
        # print(A.shape)
        # print(b.shape)

        delta_p = np.linalg.lstsq(A,b,rcond=None)[0]
        p = p + delta_p
        delta_p_length = np.square(np.linalg.norm(delta_p,ord=2))

        p1 = p[0,0]
        p2 = p[1,0]
        p3 = p[2,0]
        p4 = p[3,0]
        p5 = p[4,0]
        p6 = p[5,0]
        M = np.array([[1 + p1, p2, p3], [p4, 1 + p5, p6], [0, 0, 1]]) # (3,3) and t
        i = i + 1

    # Adjust M to be (2,3) again
    newM = M[0:2,0:3]
    # ===== End of code =====
    return newM
```

Debug Q4.1

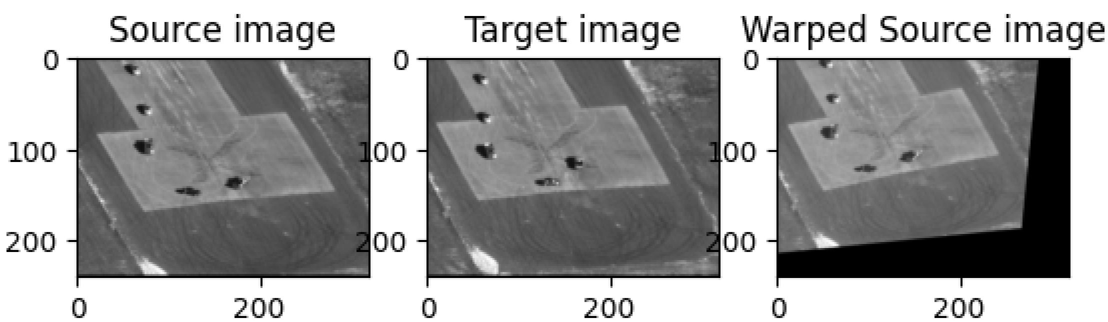
Feel free to use and modify the following snippet to debug your implementation. The snippet simply visualizes the translation resulting from running LK on a single frame. When you warp the source frame using the obtained transformation matrix, it should resemble the target frame.

In [ ]:

```
import cv2

num_iters = 100
threshold = 0.01
```

```
seq = np.load("data\\aerialseq.npy")
It = seq[:,:,0]
It1 = seq[:,:,10]

# Source frame
plt.figure()
plt.subplot(1,3,1)
plt.imshow(It, cmap='gray')
plt.title('Source image')

# Target frame
plt.subplot(1,3,2)
plt.imshow(It1, cmap='gray')
plt.title('Target image')

# Warped source frame
M = InverseCompositionAffine(It, It1, threshold, num_iters)
warped_It = cv2.warpAffine(It, M,(It.shape[1],It.shape[0]))
plt.subplot(1,3,3)
plt.imshow(warped_It, cmap='gray')
plt.title('Warped Source image')
```

Out[ ]: Text(0.5, 1.0, 'Warped Source image')



Q4.2 Tracking with Inverse Composition (10 points)

Re-use your impplementation in Q3.2 for subtract dominant motion. Just make sure to use InverseCompositionAffine within.

In [ ]:
```
import numpy as np
from scipy.ndimage import binary_erosion
from scipy.ndimage import binary_dilation
from scipy.ndimage import affine_transform
import scipy.ndimage
import cv2

def SubtractDominantMotion(It, It1, num_iters, threshold, tolerance):
    """
    :param It         : (H, W), current image
    :param It1        : (H, W), next image
    :param num_iters : (int), number of iterations for running the optimization
    :param threshold : (float), if the length of dp < threshold, terminate the opti
    :param tolerance : (float), binary threshold of intensity difference when compu
```

```
    :return: mask    : (H, W), the mask of the moved object
    """
    mask = np.ones(It.shape, dtype=bool)

    mask = np.ones(It.shape, dtype=bool)

    # ===== your code here! =====
    M = InverseCompositionAffine(It, It1, threshold, num_iters)
    warped_It = cv2.warpAffine(It, M,(It.shape[1],It.shape[0]))
    absDiff = np.abs(It1 - warped_It)
    mask1 = absDiff > tolerance
    mask2 = absDiff < np.max(absDiff)*1.0
    mask = mask1*mask2
    # ===== End of code =====

    return mask
```

Re-use your implementation in Q3.3 for sequence tracking.

In [ ]:

```
from tqdm import tqdm

def TrackSequenceAffineMotion(seq, num_iters, threshold, tolerance):
    """
    :param seq       : (H, W, T), sequence of frames
    :param num_iters : int, number of iterations for running the optimization
    :param threshold : float, if the length of dp < threshold, terminate the optimi
    :param tolerance : (float), binary threshold of intensity difference when compu
    :return: masks   : (T, 4) moved objects for each frame
    """
    H, W, N = seq.shape

    masks = []
    It = seq[:,:,0]

    # ===== your code here! =====
    for i in tqdm(range(1, seq.shape[2])):
        if (i == seq.shape[2]-1):
            It = seq[:,:,i]
            It1 = seq[:,:,i]
        else:
            It = seq[:,:,i]
            It1 = seq[:,:,i+1]
        mask = SubtractDominantMotion(It, It1, num_iters, threshold, tolerance)
        masks.append(mask)
        # print(mask)
    # ===== End of code =====
    masks = np.stack(masks, axis=2)
    return masks
```

Track the ant sequence with inverse composition method.

In [ ]:

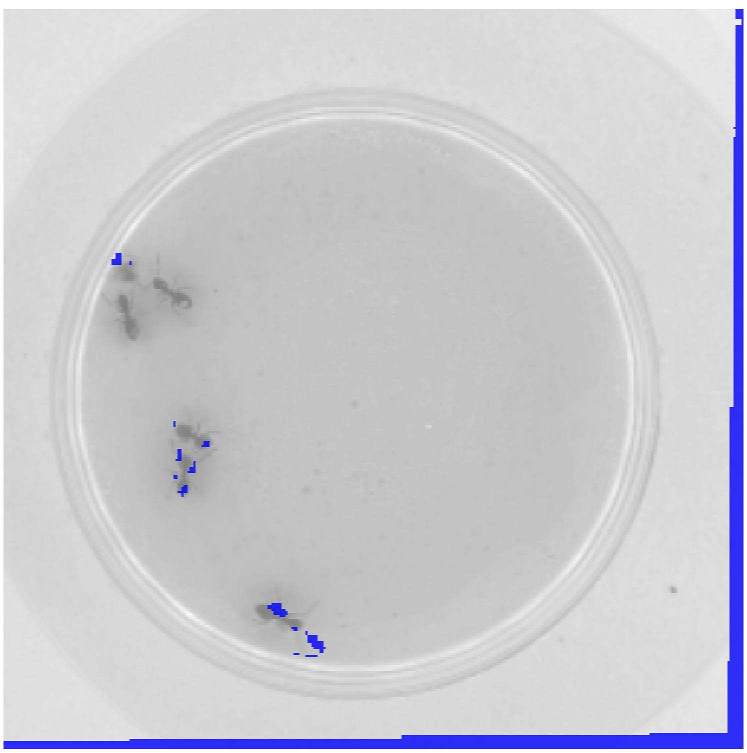
```
seq = np.load("data\\antseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 20
```

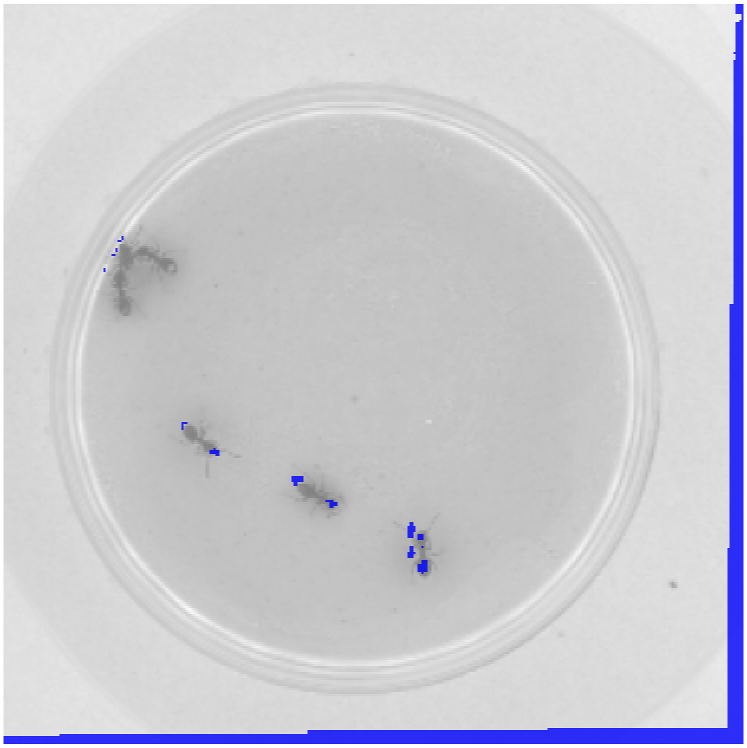
```python
threshold = 0.01
tolerance = 0.2

tic = time.time()
masks = TrackSequenceAffineMotion(seq, num_iters, threshold, tolerance)
toc = time.time()
print('\nAnt Sequence takes %f seconds' % (toc - tic))
```

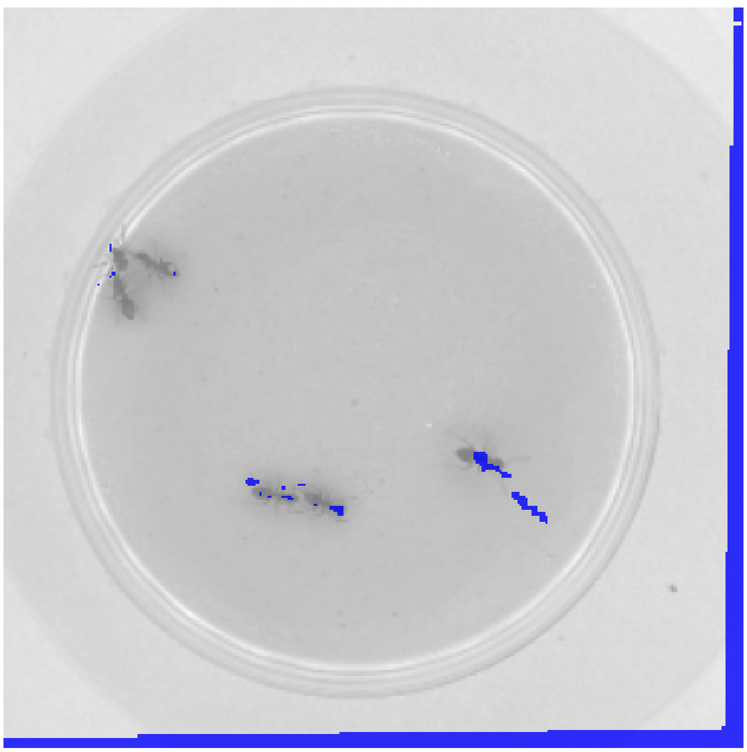
```
  0%|          | 0/124 [00:00<?, ?it/s]
100%|██████████| 124/124 [00:54<00:00,  2.28it/s]
Ant Sequence takes 54.456346 seconds
```

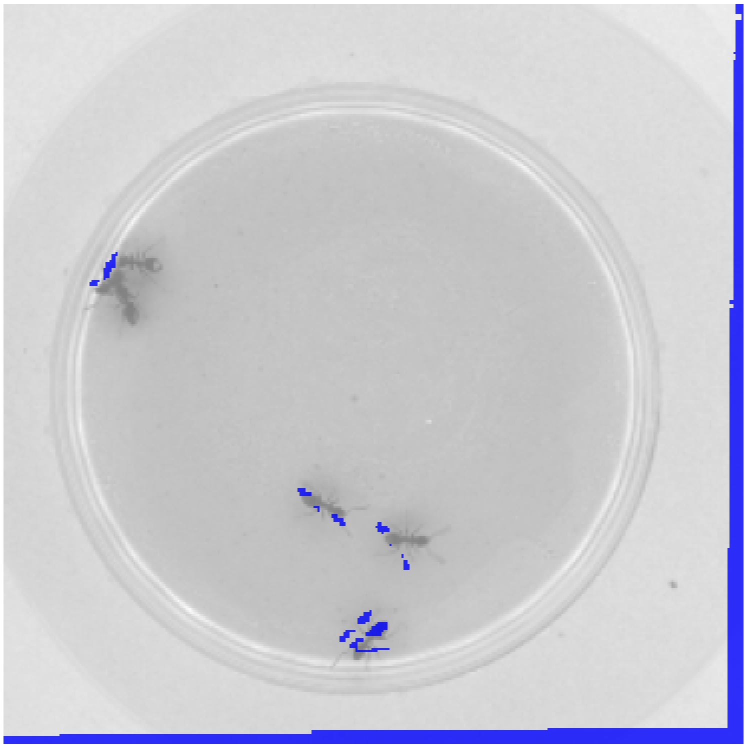
In [ ]:

```python
frames_to_save = [29, 59, 89, 119]

# TODO: visualize
for idx in frames_to_save:
    frame = seq[:, :, idx]
    mask = masks[:, :, idx]

    plt.figure()
    plt.imshow(frame, cmap="gray", alpha=0.5)
    plt.imshow(np.ma.masked_where(np.invert(mask), mask), cmap='winter', alpha=0.8)
    plt.axis('off')
```







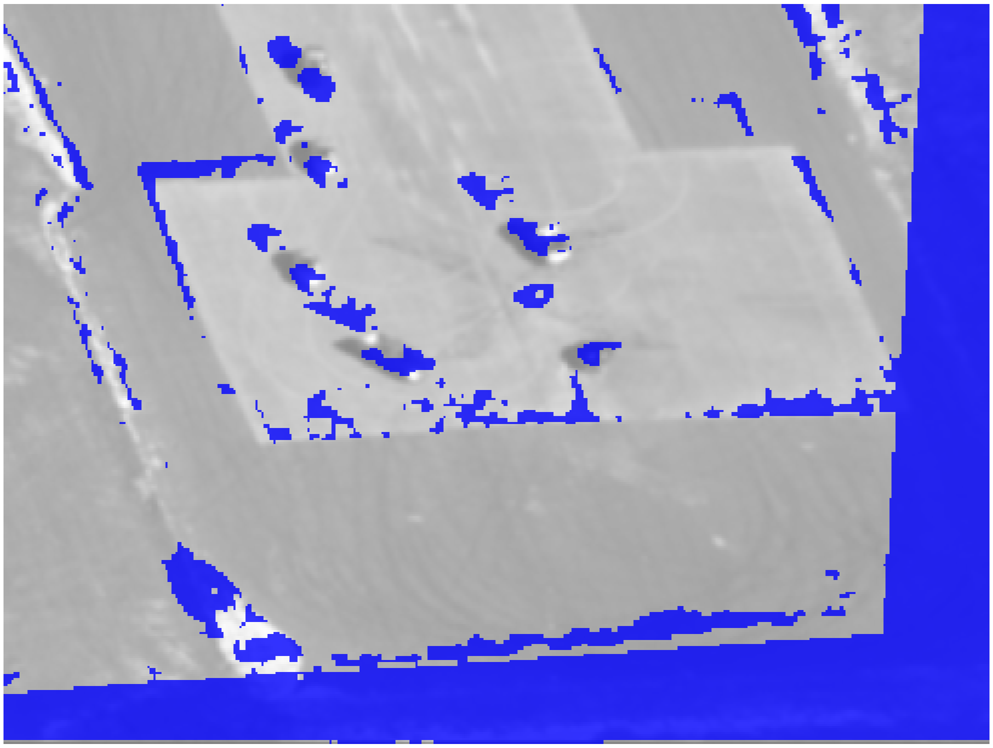


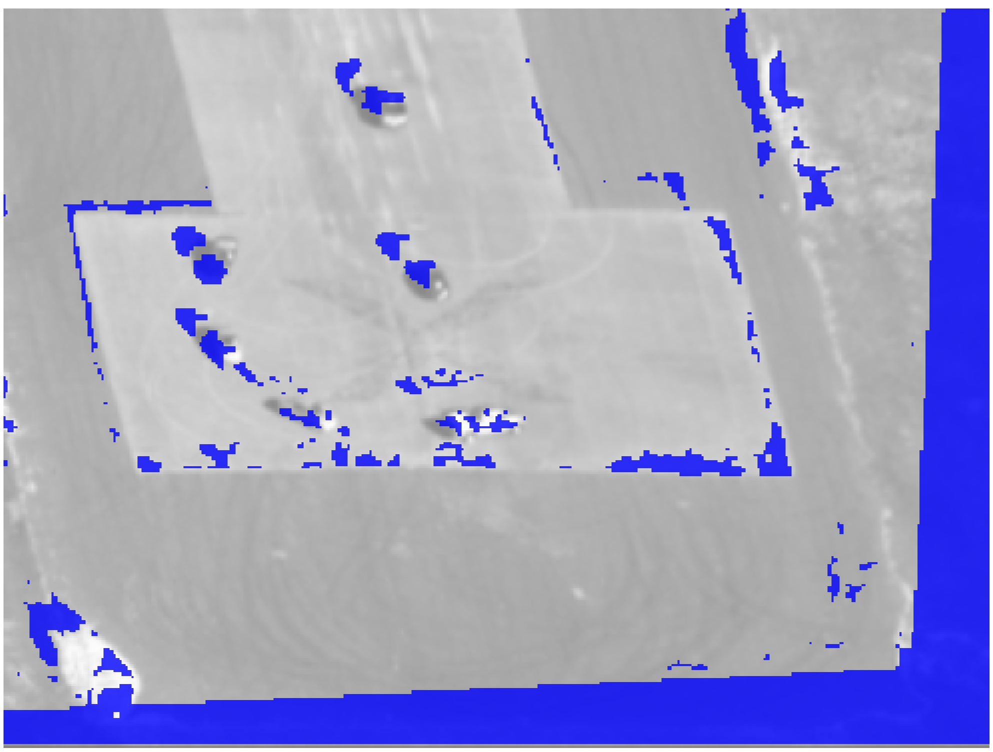
Track the aerial sequence with inverse composition method.

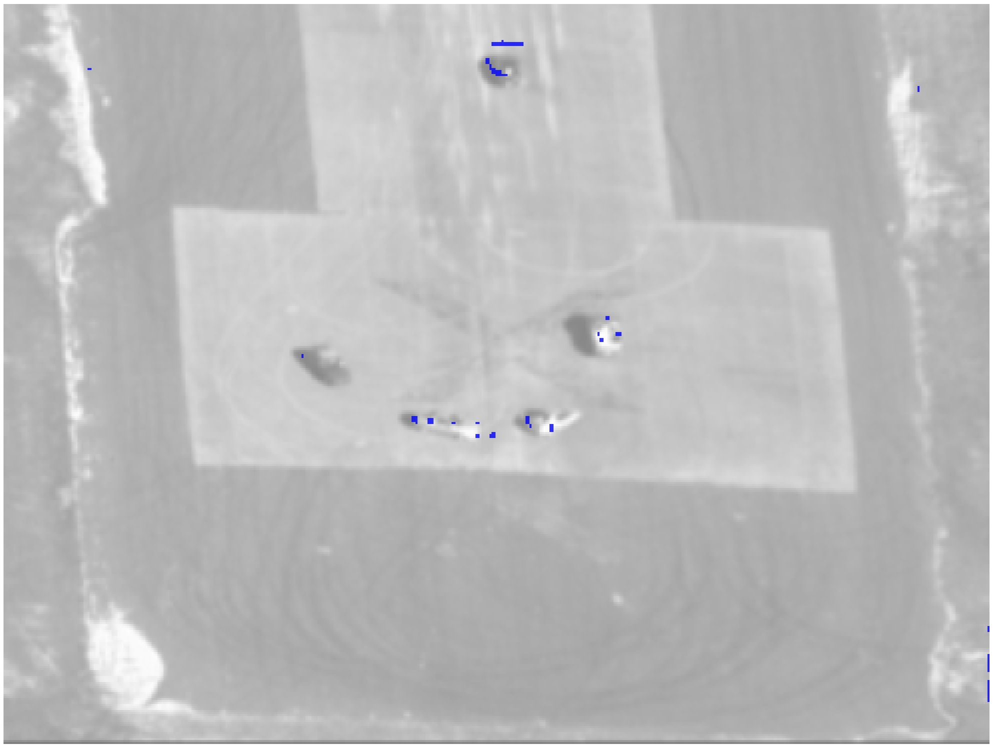
In [ ]:
```
seq = np.load("data\\aerialseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 20
threshold = 0.01
tolerance = 0.2

tic = time.time()
masks = TrackSequenceAffineMotion(seq, num_iters, threshold, tolerance)
toc = time.time()
print('\nAerial Sequence takes %f seconds' % (toc - tic))
```

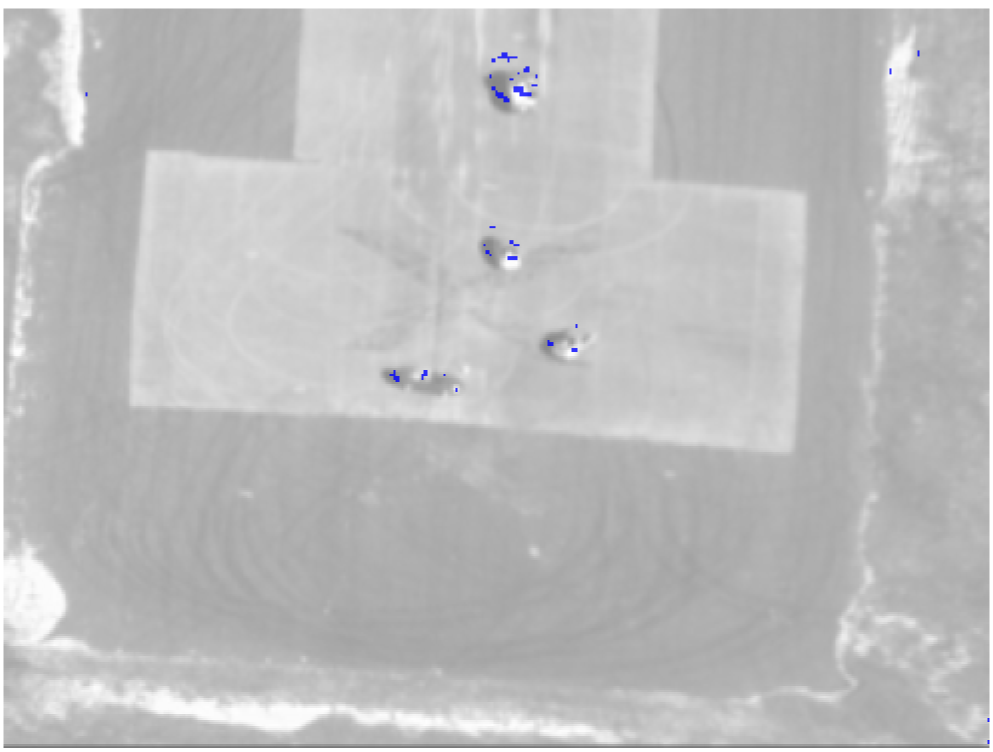
```
  0%|          | 0/149 [00:00<?, ?it/s]
100%|██████████| 149/149 [01:07<00:00,  2.19it/s]
Aerial Sequence takes 68.004451 seconds
```

In [ ]:
```
frames_to_save = [29, 59, 89, 119]

# TODO: visualize
for idx in frames_to_save:
    frame = seq[:, :, idx]
    mask = masks[:, :, idx]

    plt.figure()
    plt.imshow(frame, cmap="gray", alpha=0.5)
    plt.imshow(np.ma.masked_where(np.invert(mask), mask), cmap='winter', alpha=0.8)
    plt.axis('off')
```









Q4.2.1 Compare the runtime of the algorithm using inverse composition (as described in this section) with its runtime without inverse composition (as detailed in the previous section) in the context of the ant and aerial sequences:

===== your answer here! =====

The ant sequence normally took 56 seconds but was decreased to around 53 seconds using the inverse composition method.

The aerial sequence normally took 79 seconds but was decreased to around 66 seconds using the inverse composition method.

===== end of your answer ====

Q4.2.2 In your own words, please describe briefly why the inverse compositional approach is more computationally efficient than the classical approach:

===== your answer here! =====

You don't have to recompute the Hessian over and over again. For my computation strategy that's equivalent to not having to compute the A matrix over and over again.

===== end of your answer ====