## 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.

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

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

## Download data

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

```
# Download the data
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
```

```
--2024-02-17 21:00:39--  https://www.cs.cmu.edu/~deva/data/carseq.npy
Resolving www.cs.cmu.edu (www.cs.cmu.edu)... 128.2.42.95
Connecting to www.cs.cmu.edu (www.cs.cmu.edu)|128.2.42.95|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 254976128 (243M)
Saving to: '/content/carseq.npy'

/content/carseq.npy 100%[===================>] 243.16M  4.40MB/s    in 57s

2024-02-17 21:01:36 (4.27 MB/s) - '/content/carseq.npy' saved [254976128/254976128]

--2024-02-17 21:01:36--  https://www.cs.cmu.edu/~deva/data/girlseq.npy
Resolving www.cs.cmu.edu (www.cs.cmu.edu)... 128.2.42.95
Connecting to www.cs.cmu.edu (www.cs.cmu.edu)|128.2.42.95|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 27648128 (26M)
Saving to: '/content/girlseq.npy'

/content/girlseq.np 100%[===================>]  26.37M  4.27MB/s    in 6.1s

2024-02-17 21:01:42 (4.31 MB/s) - '/content/girlseq.npy' saved [27648128/27648128]
```

## Q2.1: Theory Questions (5 points)

Please refer to the handout for the detailed questions.

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

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

$\frac{\partial \square(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T}$ is called the Jacobian of the Warp. It measures the deriative of the Warp with respect to each of the Parameters that are used in the Warp.

Here, it is a 2x2 matrix since the Warp is Translation. A translation can be described by two parameters p1 and p2.

Taking the derivative of W = [[1 0 p1] , [0 1 p2]] dW/dp = [[ dWx/dp1 dWx/dp2 ], [ dWy/dp1 dWy/dp2 ]]

$\frac{\partial \square(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T}$ = [[1,0], [0,1]]

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

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

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

A is the Steepest Descent Images matrix. It is calculated by multipying the Gradient of the Warped Image with the Jacobian of the Warp

A = $\frac{\partial \square(\mathbf{W}(\mathbf{x};\mathbf{p});\mathbf{x})}{\partial \mathbf{p}^T} \cdot \frac{\partial \square(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T}$

b is the error between the Template and Warped Image. It tells us the pixel-wise difference in intensity values between the two.

===== 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! =====

The coniditions $\mathbf{A}T\mathbf{A}$ that must be met so that a unique solution for $\Delta\mathbf{p}$ is that $\mathbf{A}T\mathbf{A}$ must be invertible (full rank and non-singular determinant)

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

## Q2.2: Lucas-Kanade (20 points)

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

```
def LucasKanade(It, It1, rect, threshold, num_iters, p0=np.zeros(2)):
    """
    :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 template to extract from the image at time t,
                                  where [x1, y1] is the top-left, and [x2, y2] is the bottom-right. Note that coordinates
                                  [floats] that maybe fractional.
    :param[float] threshold     : If change in parameters is less than thresh, terminate the optimization
    :param[int] num_iters       : Maximum number of optimization iterations
    :param[np.array(2, 1)] p0   : Initial translation parameters [p_x0, p_y0] to add to rect, which defaults to [0 0]

    :return[np.array(2, 1)] p   : Final translation parameters [p_x, p_y]
    """

    # Initialize p to p0.
    p = p0
    # ===== your code here! =====

    # Extract rectangle coordinates
    x1, y1, x2, y2 = rect
    # Generate the pixel coordinates inside the rectangle, this is to extract the template from It
    X, Y = np.meshgrid(np.arange(x1, x2 + 1), np.arange(y1, y2 + 1), indexing='xy')
    coords = np.vstack((X.flatten(), Y.flatten()))
    delta_p_norms = []
    # Iterate over num_iters
    for _ in range(num_iters):
        # Warp the template using the current parameters
        warped_coords = coords + p.reshape(2,1)
```

```
        # Extract pixel intensities using interpolation
        It1_rbs = RectBivariateSpline(np.arange(It.shape[0]),np.arange(It.shape[1]) , It1)
        warped_It1 = It1_rbs.ev(warped_coords[1], warped_coords[0])
        # RBS on It as well, to deal with Fractional Coordinates
        It_rbs = RectBivariateSpline(np.arange(It.shape[0]),np.arange(It.shape[1]) , It)
        T = It_rbs.ev(coords[1], coords[0])
        # Compute Image gradients
        gradient_x = It1_rbs.ev(warped_coords[1], warped_coords[0], dx=1)
        gradient_y = It1_rbs.ev(warped_coords[1], warped_coords[0], dy=1)
        # Construct the Gradient of Warped Image matrix
        GW = np.vstack((gradient_y.flatten(), gradient_x.flatten())).T
        # Jacobian of Translation
        J=np.array([[1,0],[0,1]])
        # Compute A
        A = GW @ J
        # Compute the error vector
        b = T - warped_It1
        # Solve for delta_p using least squares
        delta_p, _, _, _ = lstsq(A, b, rcond=None)
        # Update parameters
        p += delta_p
        delta_p_norms.append(np.linalg.norm(delta_p))
        # Check convergence
        if np.linalg.norm(delta_p) < threshold:
          break

        # ===== End of code =====
    return p, delta_p_norms
```

## Debug Q2.2

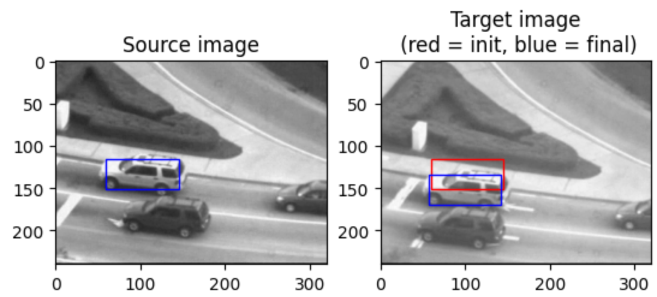
A few tips to debug your implementation:

- 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

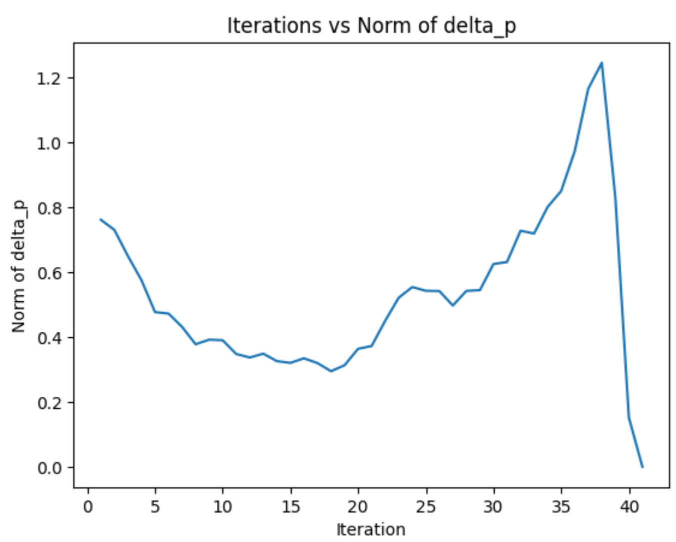
```
def draw_rect(rect,color):
    w = rect[2] - rect[0]
    h = rect[3] - rect[1]
    plt.gca().add_patch(patches.Rectangle((rect[0],rect[1]), w, h, linewidth=1, edgecolor=color, facecolor='none'))

```

```
num_iters = 10000
threshold = 0.01
seq = np.load("/content/carseq.npy")
rect = [59, 116, 145, 151]
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)
plt.imshow(It1, cmap='gray')
plt.title('Target image\n (red = init, blue = final)')
p, delta_p_norms= LucasKanade(It, It1, rect, threshold, num_iters, p0=np.zeros(2))
print(p)
rect_t1 = rect + np.concatenate((p,p))
draw_rect(rect,'r')
draw_rect(rect_t1,'b')
```

```
[-2.34873667 18.5660736 ]
```




```
# Plot iterations vs norm of delta_p
plt.plot(range(1, len(delta_p_norms) + 1), delta_p_norms)
plt.xlabel('Iteration')
plt.ylabel('Norm of delta_p')
plt.title('Iterations vs Norm of delta_p')
plt.show()
```



```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-left and bottom-right corners.
    :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 =[]
    It = seq[:,:,0]

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

        It1 = seq[:,:,i]
        p, _ = LucasKanade(It, It1, rect, threshold, num_iters)
        # Update Template Box
        rect = rect + np.concatenate((p, p))
        rects.append(rect)
        It=seq[:,:,i]

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

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

```python
def visualize_track(seq,rects,frames):
    # 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');
```

```python
seq = np.load("/content/carseq.npy")
rect = [59, 116, 145, 151]

# 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])
```

## Q2.3 (b) - Track Girl Sequence

Same as the car sequence.

```python
# Loads the squence
seq = np.load("/content/girlseq.npy")
rect = [280, 152, 330, 318]

# 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])
```

```python

```

## 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.

```
import time
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from scipy.interpolate import RectBivariateSpline
```

## Download data

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

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

```
--2024-02-18 03:06:35--  https://www.cs.cmu.edu/~deva/data/aerialseq.npy
Resolving www.cs.cmu.edu (www.cs.cmu.edu)... 128.2.42.95
Connecting to www.cs.cmu.edu (www.cs.cmu.edu)|128.2.42.95|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 92160128 (88M)
Saving to: '/content/aerialseq.npy'

/content/aerialseq. 100%[===================>]  87.89M  4.56MB/s    in 22s

2024-02-18 03:06:57 (4.08 MB/s) - '/content/aerialseq.npy' saved [92160128/92160128]

--2024-02-18 03:06:57--  https://www.cs.cmu.edu/~deva/data/antseq.npy
Resolving www.cs.cmu.edu (www.cs.cmu.edu)... 128.2.42.95
Connecting to www.cs.cmu.edu (www.cs.cmu.edu)|128.2.42.95|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 65536128 (62M)
Saving to: '/content/antseq.npy'

/content/antseq.npy 100%[===================>]  62.50M  4.46MB/s    in 14s

2024-02-18 03:07:11 (4.51 MB/s) - '/content/antseq.npy' saved [65536128/65536128]
```

## Q3: Affine Motion Subtraction

## Q3.1: Dominant Motion Estimation (15 points)

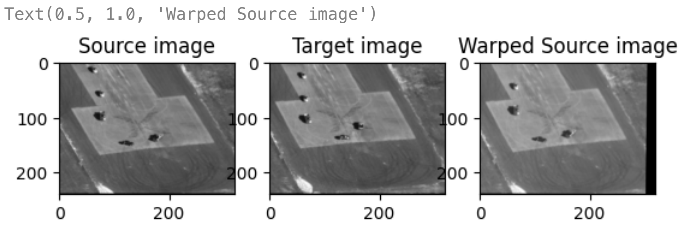
```python
def LucasKanadeAffine_Please(It, It1, threshold, num_iters):
    """
    :param It        : (H, W), current image
    :param It1       : (H, W), next image
    :param threshold : (float), if the length of dp < threshold, terminate the optimization
    :param num_iters : (int), number of iterations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

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

    # ===== your code here! =====

    # -- Get the Shape of Iamge
    H, W = It.shape
    # -- Generate X,Y Coordinate Plane
    X, Y = np.meshgrid(np.arange(W) , np.arange(H), indexing='xy')
    coords = np.vstack((X.flatten(), Y.flatten(), np.ones_like(X.flatten())))

    for i in range(num_iters):
      # -- Warping the Coordinates
      warped_coords = M @ coords
      # -- Create Mask
      x_mask = np.array([True if ((i>=0) and (i<=It.shape[1])) else False for i in warped_coords[0]])
      y_mask =  np.array([True if ((i>=0) and(i<=It.shape[0])) else False for i in warped_coords[1]])
      final_mask= np.logical_and(x_mask, y_mask)
      # -- Interpolate the Values for It1
      It1_rbs = RectBivariateSpline(np.arange(H),np.arange(W), It1)
      warped_It1 = It1_rbs.ev(warped_coords[1], warped_coords[0])
      #Interpolate for It
      It_rbs = RectBivariateSpline(np.arange(H),np.arange(W), It)
      T = It_rbs.ev(coords[1], coords[0])
      # -- Compute Error
      error = T - warped_It1
      # -- Mask the Out of Bound Pixels
      error = error[final_mask]
      # -- Gradients
      gradient_x = It1_rbs.ev(warped_coords[1], warped_coords[0], dy=1)
      gradient_y = It1_rbs.ev(warped_coords[1], warped_coords[0], dx=1)
      # -- Gradient of Warped Image (GW)
      GW = np.vstack((gradient_x, gradient_y)).T
      # -- Jacobian (J)
      J = np.zeros((GW.shape[0], 2, 6))
      J[:, 0, 0]=X.flatten()
      J[:, 0, 1]=Y.flatten()
      J[:, 0, 2]=1
      J[:, 1, 3]=X.flatten()
      J[:, 1, 4]=Y.flatten()
      J[:, 1, 5]=1
      # -- Steepest Descent Images (SDI)
      GW_expanded = GW[:, :, np.newaxis]
      res = GW_expanded * J
      SDI = np.sum(res, axis=1)
      # Removing the Out of Bound Coordinates
      SDI=SDI[final_mask, :]
      # -- Hessian H and Hessian Inverse H_inv
      Hess = SDI.T @ SDI
      H_inv = np.linalg.pinv(Hess)
      # -- delta p
      delta_p = H_inv @ (SDI.T @ error)
      delta_p = delta_p.reshape([2,3])
      val = np.linalg.norm(delta_p)
      M += delta_p
      if val <= threshold:
        break
    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.

```python
import cv2

num_iters = 200
threshold = 0.025
seq = np.load("/content/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_Please(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')
```

```
Text(0.5, 1.0, 'Warped Source image')
```




## Q3.2: Moving Object Detection (10 points)

```python
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 optimization
    :param tolerance : (float), binary threshold of intensity difference when computing the mask
    :return: mask    : (H, W), the mask of the moved object
    """
    mask = np.ones(It.shape, dtype=bool)

    # ===== your code here! =====

    # Estimate M
    M = LucasKanadeAffine_Please(It, It1, threshold, num_iters)
    # Warp It using the estimated motion
    warped_It = cv2.warpAffine(It, M, (It.shape[1], It.shape[0]))
    # Compute absolute intensity difference
    diff = np.abs(warped_It - It1)
    # Threshold the intensity difference to create binary mask
    mask = diff > tolerance
    mask = binary_dilation(mask, iterations=5)
    mask = binary_erosion(mask,iterations=4)

    return mask

```

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

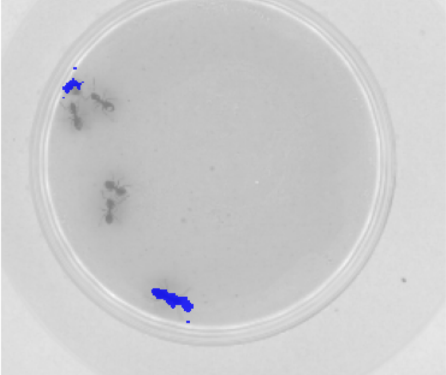
```python
from tqdm import tqdm

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 optimization
    :param tolerance : (float), binary threshold of intensity difference when computing the mask
    :return: masks   : (T, 4) moved objects for each frame
    """
    H, W, N = seq.shape

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

    # ===== your code here! =====
    for i in tqdm(range(1, N)):

        # Estimate dominant motion between current frame and previous frame
        mask = SubtractDominantMotion(seq[:, :, i - 1], seq[:, :, i], num_iters, threshold, tolerance)
        masks.append(mask)

    masks = np.stack(masks, axis=2)
    return masks
```

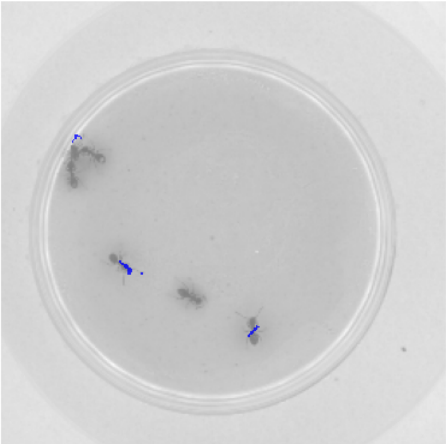
## Q3.3 (a) - Track Ant Sequence

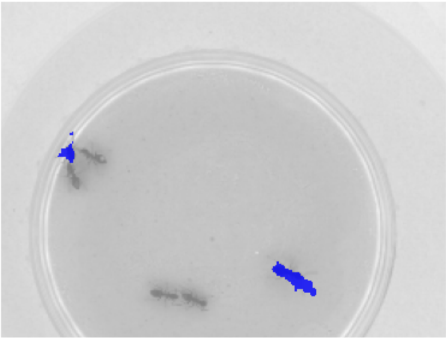
```python
seq = np.load("/content/antseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 1000
threshold = 0.01
tolerance = 0.15

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

```
100%|██████████| 124/124 [02:13<00:00,  1.08s/it]
Ant Sequence takes 133.447735 seconds
```

```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')

```





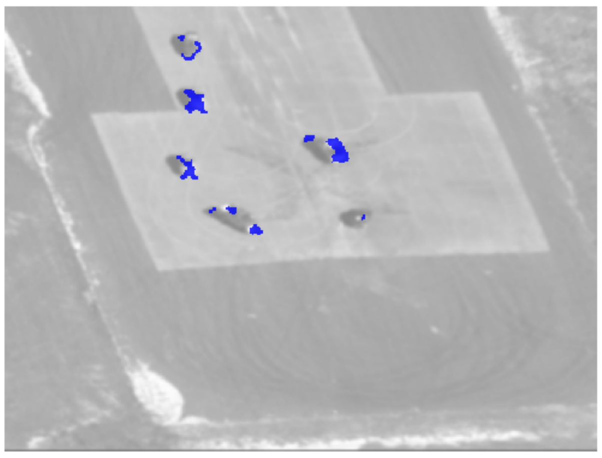


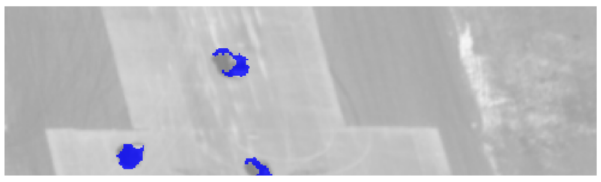
## Q3.3 (b) - Track Aerial Sequence

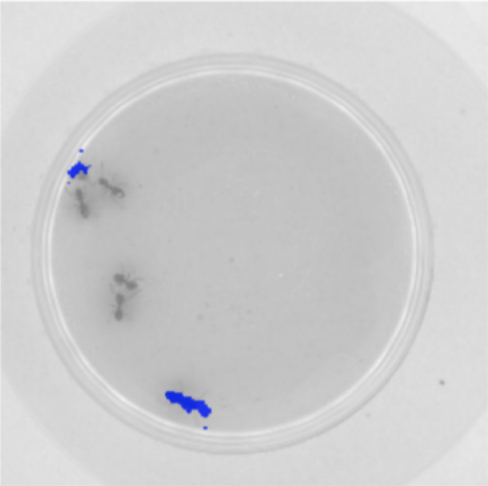
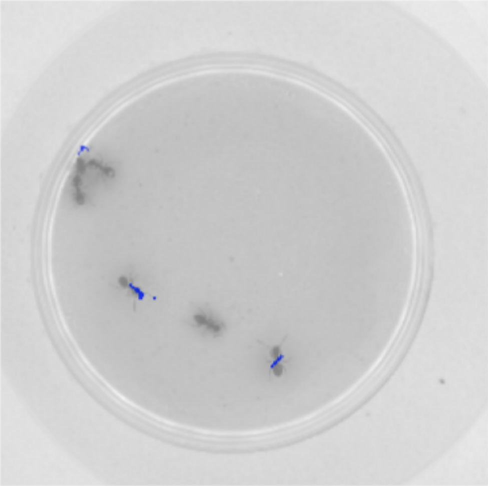
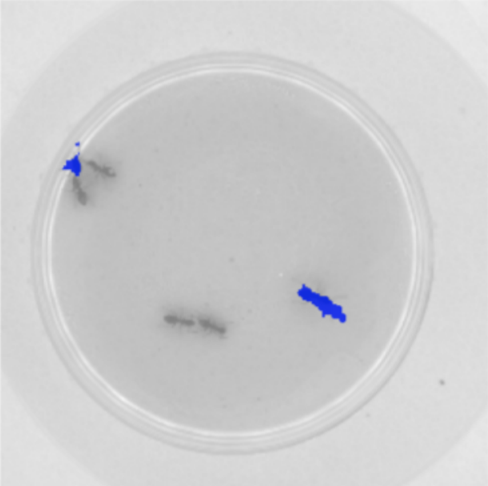
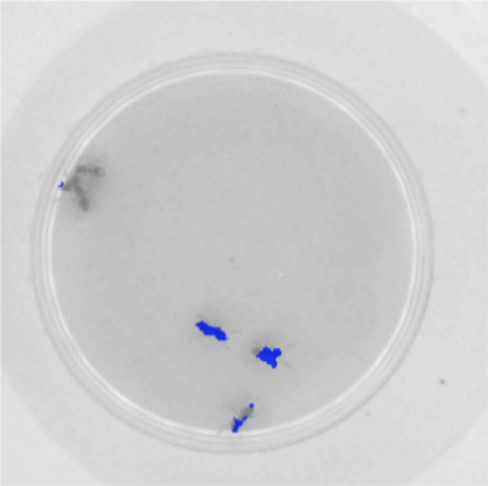
```
seq = np.load("/content/aerialseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 1000
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))
```

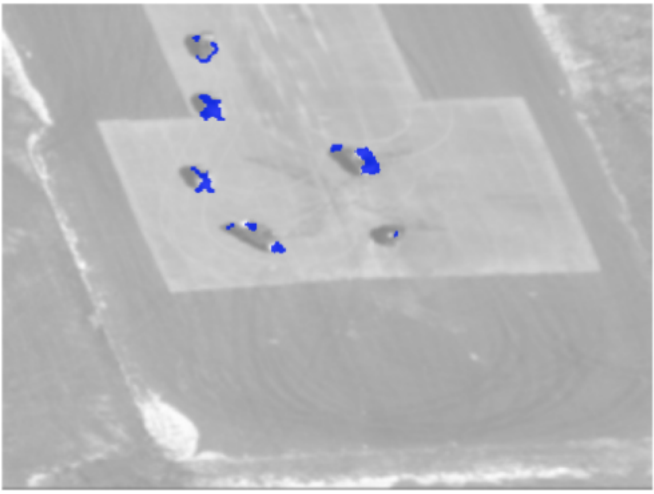
```
100%|██████████| 149/149 [05:39<00:00,  2.28s/it]
Aerial Sequence takes 339.759714 seconds
```

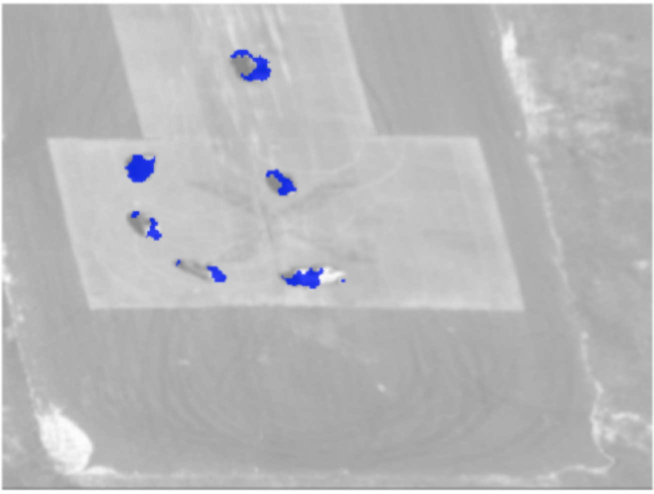
```
frames_to_save = [29, 59, 89, 119]

# 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')
```

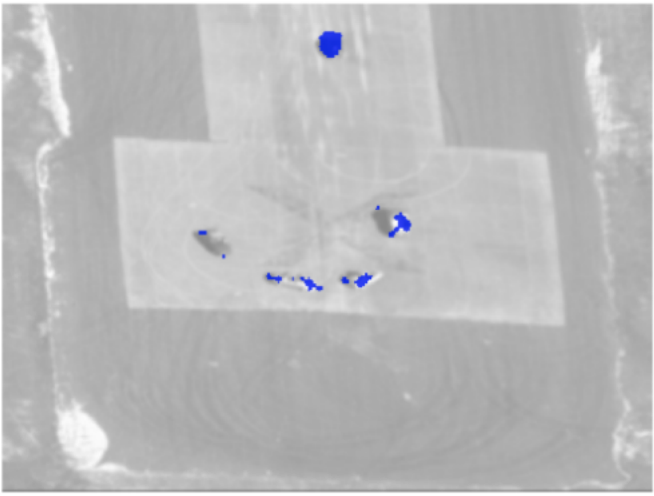


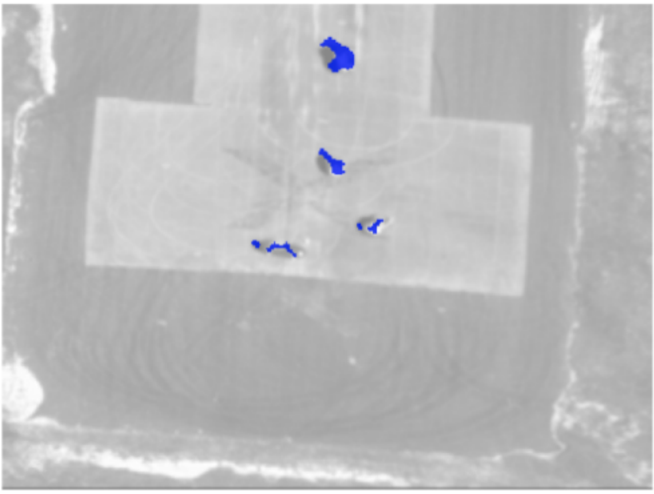










## Initialization

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.

```
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.

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

```
--2024-02-18 04:00:33--  https://www.cs.cmu.edu/~deva/data/aerialseq.npy
Resolving www.cs.cmu.edu (www.cs.cmu.edu)... 128.2.42.95
Connecting to www.cs.cmu.edu (www.cs.cmu.edu)|128.2.42.95|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 92160128 (88M)
Saving to: '/content/aerialseq.npy'

/content/aerialseq. 100%[===================>]  87.89M   911KB/s    in 50s

2024-02-18 04:01:24 (1.77 MB/s) - '/content/aerialseq.npy' saved [92160128/92160128]

--2024-02-18 04:01:24--  https://www.cs.cmu.edu/~deva/data/antseq.npy
Resolving www.cs.cmu.edu (www.cs.cmu.edu)... 128.2.42.95
Connecting to www.cs.cmu.edu (www.cs.cmu.edu)|128.2.42.95|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 65536128 (62M)
Saving to: '/content/antseq.npy'

/content/antseq.npy 100%[===================>]  62.50M  1.37MB/s    in 38s

2024-02-18 04:02:03 (1.63 MB/s) - '/content/antseq.npy' saved [65536128/65536128]
```

## Q4: Efficient Tracking

### Q4.1: Inverse Composition (15 points)

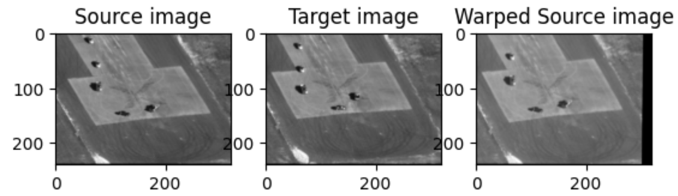
```python
from scipy.interpolate import RectBivariateSpline

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 optimization
    :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! =====

    # -- Get the Shape of Iamge
    H, W = It.shape
    # -- Generate X,Y Coordinate Plane
    X, Y = np.meshgrid(np.arange(W) , np.arange(H), indexing='xy')
    coords = np.vstack((X.flatten(), Y.flatten(), np.ones_like(X.flatten())))
    #Interpolate for It
    It_rbs = RectBivariateSpline(np.arange(H),np.arange(W), It)
    T = It_rbs.ev(coords[1], coords[0])
    #Get Gradients
    gradient_x = It_rbs.ev(coords[1], coords[0], dy=1)
    gradient_y = It_rbs.ev(coords[1], coords[0], dx=1)
    # -- Gradient of Template (GW)
    GW = np.vstack((gradient_x, gradient_y)).T
    # -- Jacobian (J)
    J = np.zeros((GW.shape[0], 2, 6))
    J[:, 0, 0]=X.flatten()
    J[:, 0, 1]=Y.flatten()
    J[:, 0, 2]=1
    J[:, 1, 3]=X.flatten()
    J[:, 1, 4]=Y.flatten()
    J[:, 1, 5]=1
    # -- Steepest Descent Images (SDI)
    GW_expanded = GW[:, :, np.newaxis]
    res = GW_expanded * J
    SDI = np.sum(res, axis=1)
    # -- Hessian H and Hessian Inverse H_inv
    Hess = SDI.T @ SDI
    H_inv = np.linalg.pinv(Hess)

    for i in range(num_iters):
      # -- Warping the Coordinates
      warped_coords = M @ coords
      # -- Create Mask
      x_mask = np.array([True if ((i>=0) and (i<=It.shape[1])) else False for i in warped_coords[0]])
      y_mask =  np.array([True if ((i>=0) and(i<=It.shape[0])) else False for i in warped_coords[1]])
      final_mask= np.logical_and(x_mask, y_mask)
      # -- Interpolate the Values
      #Interpolate on It1
      It1_rbs = RectBivariateSpline(np.arange(H),np.arange(W), It1)
      warped_It1 = It1_rbs.ev(warped_coords[1], warped_coords[0])
      # -- Compute Error
      error = T - warped_It1
      error[~final_mask] = 0
      # -- delta p
      delta_p = H_inv @ (SDI.T @ error)
      delta_p1, delta_p2, delta_p3, delta_p4, delta_p5, delta_p6 = delta_p
      # -- delta m
      delta_M = np.array([[1+ delta_p1, delta_p2, delta_p3], [delta_p4, 1+delta_p5, delta_p6],[0,0,1]])
      delta_m_inv = np.linalg.pinv(delta_M)
      M = M @ delta_M
      val = np.linalg.norm(delta_p)
      if val <= threshold:
        break

    return M

```

## 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.

```python
import cv2

num_iters = 200
threshold = 0.01
seq = np.load("/content/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)
print(M)
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')
```

```
[[ 9.86311293e-01 -1.18960384e-02 -1.04242376e+01]
 [ 2.91335675e-03  1.07828943e+00 -1.89337524e+01]]
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.

```python
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 optimization
    :param tolerance : (float), binary threshold of intensity difference when computing the mask
    :return: mask    : (H, W), the mask of the moved object
    """
    mask = np.ones(It.shape, dtype=bool)

    # ===== your code here! =====

    # Estimate M
    M = InverseCompositionAffine(It, It1, threshold, num_iters)
    # Warp It using the estimated motion
    warped_It = cv2.warpAffine(It, M, (It.shape[1], It.shape[0]))
    # Compute absolute intensity difference
    diff = np.abs(warped_It - It1)
    # Threshold the intensity difference to create binary mask
    mask = diff > tolerance
    mask = binary_dilation(mask, iterations=5)
    mask = binary_erosion(mask,iterations=4)

    return mask

```

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

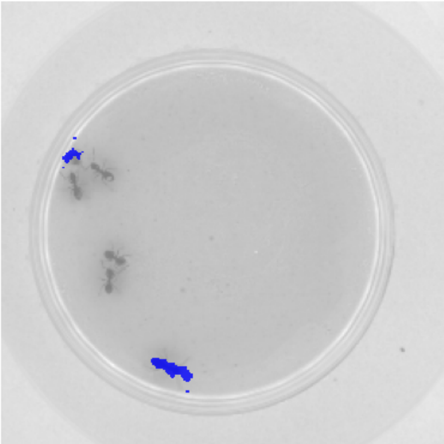
```python
from tqdm import tqdm

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 optimization
    :param tolerance : (float), binary threshold of intensity difference when computing the mask
    :return: masks   : (T, 4) moved objects for each frame
    """
    H, W, N = seq.shape

    masks=[]

    # ===== your code here! =====
    for i in tqdm(range(1, N)):

        # Estimate dominant motion between current frame and previous frame
        mask = SubtractDominantMotion(seq[:, :, i - 1], seq[:, :, i], num_iters, threshold, tolerance)
        masks.append(mask)

    masks = np.stack(masks, axis=2)
    return masks
```

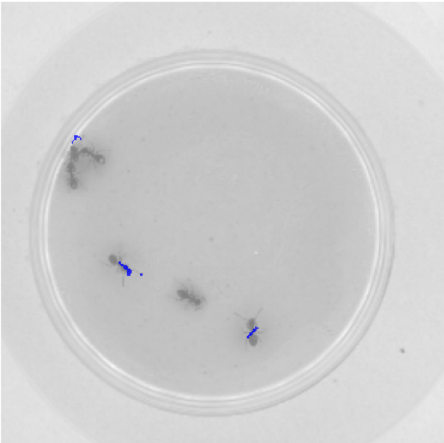
Track the ant sequence with inverse composition method.

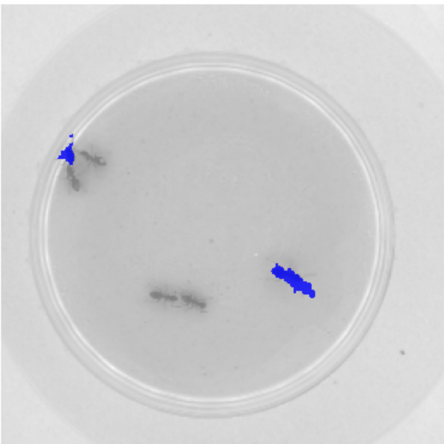
```python
seq = np.load("/content/antseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 1000
threshold = 0.01
tolerance = 0.15

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

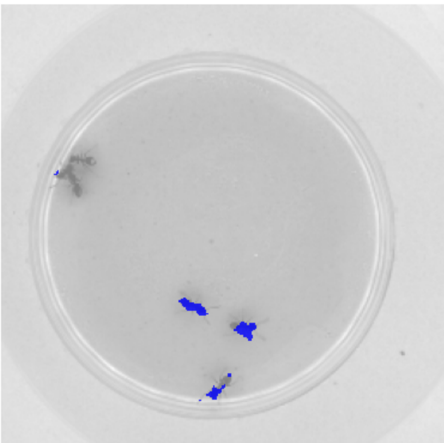
```
100%|██████████| 124/124 [01:18<00:00,  1.58it/s]
Ant Sequence takes 78.469546 seconds
```

```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.

```
seq = np.load("/content/aerialseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 1000
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))
```

```
100%|██████████| 149/149 [03:49<00:00,  1.54s/it]
Ant Sequence takes 229.307912 seconds
```

## 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! =====

| Sequence | Normal Affine | Inverse Composition |
|---|---|---|
| Ant | 139.18s | 76.33s |
| Aerial | 342.66s | 247.19s |

===== 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! =====

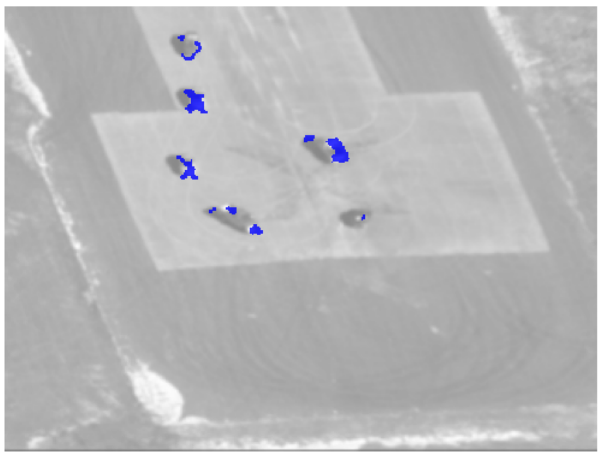
In the original LK Approach, we're updating the warp parameters. We get the optimal delta_p by iteratively updating the warp parameters, and warping the Image It1 at each step until we reach convergence. This involves recalculation of the Jacobian, Gradient, Steepest Descent Images, and Hessian for It1 at each iteration.

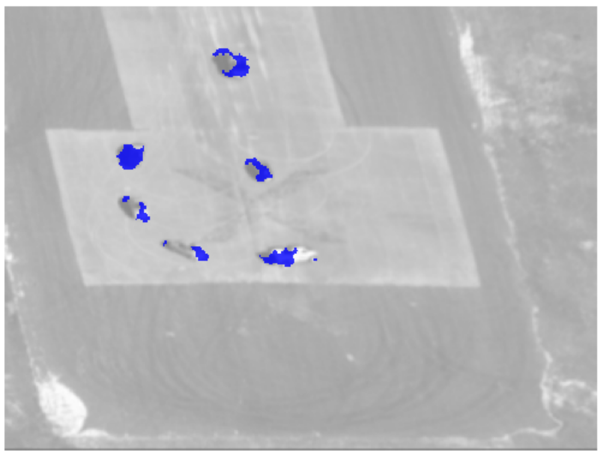
In the Inverse Compositional approach, we're avoiding this recalculation by computing the Jacobian, Gradient, Steepest Descent Images, and Hessian on the original Image It instead.

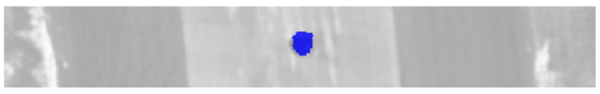
By computing matrices such as the Jacobian (which doesn't depend on p), the Gradient of the Template (which stays constant), and ultimately, the Hessian only once, we are able to drastically reduce the computational time required.

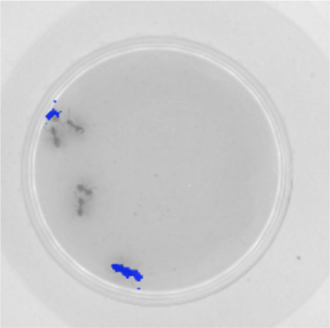
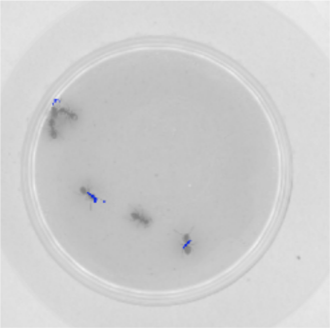
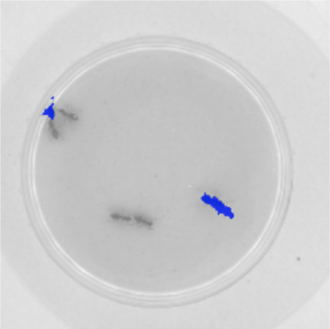
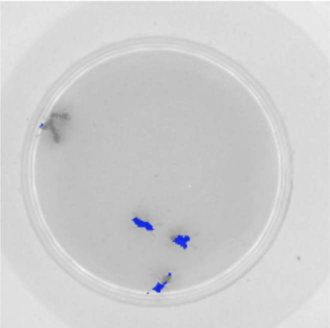
===== end of your answer ====

```
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')
```







```
plt.axis('off')
```