LucasKanade

February 21, 2024

1 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.

[74]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import scipy.signal
```

2 Download data

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

[75]:
```
# 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
```

3 Q2.1: Theory Questions (5 points)

Please refer to the handout for the detailed questions.

3.1 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! =====

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

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

3.2 Q2.1.2: What is A and b?

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

$$\mathbf{A} = \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial \mathcal{W}(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T} \mathbf{b} = -\mathcal{I}_{t+1}\left(\mathcal{W}(\mathbf{x};\mathbf{p})\right) + \mathcal{T}(\mathbf{x}) \tag{2}$$

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

3.3 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! =====

$\mathbf{A}^T\mathbf{A}$ must be invertible, which is to say $\mathbf{A}^T\mathbf{A}$ should be non-singuler and positive definite.

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

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

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

[103]:
```python
from scipy.interpolate import RectBivariateSpline
from numpy.linalg import lstsq

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! =====
    # Hint: Iterate over num_iters and for each iteration, construct a linear␣
 ↪system (Ax=b) that solves for a x=delta_p update
```

```python
    # Construct [A] by computing image gradients at (possibly fractional) pixel␣
↪locations.
    # We suggest using RectBivariateSpline from scipy.interpolate to␣
↪interpolate pixel values at fractional pixel locations
    # 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 'xy' versus 'ij' indexing:
    # https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.
↪RectBivariateSpline.ev.html#scipy.interpolate.RectBivariateSpline.ev
    # https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html

    # Step1: generate x and y coordinates
    x1, y1, x2, y2 = rect
    x_range = int(x2 - x1)
    y_range = int(y2 - y1)
    x = np.arange(0, x_range + 1) + x1
    y = np.arange(0, y_range + 1) + y1
    X, Y = np.meshgrid(x, y)

    # Step2: gradients calculation
    f = np.array([1, 0, -1], ndmin=2)  # Filter for gradient calculation
    xGradient = scipy.signal.convolve2d(It1, f, mode='same')
    yGradient = scipy.signal.convolve2d(It1, f.T, mode='same')

    # Step3: interpolators for the images and gradients
    spline_It = RectBivariateSpline(np.arange(It.shape[0]), np.arange(It.
↪shape[1]), It)
    spline_It1 = RectBivariateSpline(np.arange(It.shape[0]), np.arange(It.
↪shape[1]), It1)
    spline_x = RectBivariateSpline(np.arange(It.shape[0]), np.arange(It.
↪shape[1]), xGradient)
    spline_y = RectBivariateSpline(np.arange(It.shape[0]), np.arange(It.
↪shape[1]), yGradient)

    T = spline_It.ev(Y, X)

    # Step4: Start the for loop iteration
    for _ in range(num_iters):
        X_warp = X + p[0]
        Y_warp = Y + p[1]

        warped_It1 = spline_It1.ev(Y_warp, X_warp)
```

```
        It1_x = spline_x.ev(Y_warp, X_warp)
        It1_y = spline_y.ev(Y_warp, X_warp)

        A = np.stack((It1_x.ravel(), It1_y.ravel())).T
        b = (T - warped_It1).ravel()

        delta_p, _, _, _ = np.linalg.lstsq(A, b, rcond=None)
        p += delta_p

        if np.linalg.norm(delta_p) < threshold:
            break

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

4.1 Debug Q2.2

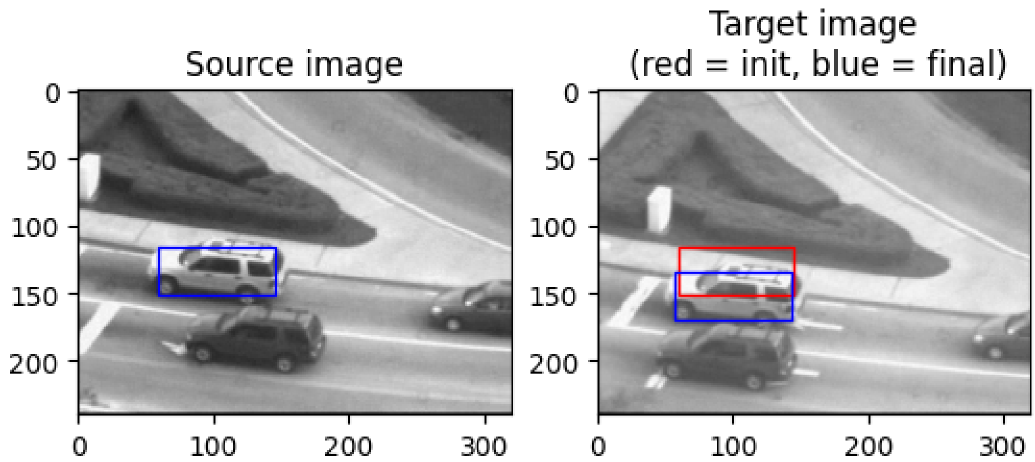
A few tips to debug your implementation: - Feel free to use and modify the following snippet to debug your implementation. The snippet simply visualizes the translation resulting from running LK on a single frame. You should be able to see a slight shift in the template.

- You may also want to visualize the image gradients you compute within your LK implementation
- Plot iterations vs the norm of delta_p

[104]:
```
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'))
```

[105]:
```
num_iters = 100
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)
```

```python
plt.imshow(It1, cmap='gray')
plt.title('Target image\n (red = init, blue = final)')
p = LucasKanade(It, It1, rect, threshold, num_iters, p0=np.zeros(2))
rect_t1 = rect + np.concatenate((p,p))
draw_rect(rect,'r')
draw_rect(rect_t1,'b')
```



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

[106]:
```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! =====
        # TODO: add your code track the object of interest in the sequence
        if i == 0:
          p = np.zeros(2)
```

```
        else:
          p = np.array([rects[i-1][0]-rect[0], rects[i-1][1]-rect[1]])

        p = LucasKanade(It, seq[:,:,i], rect, threshold, num_iters, p)
        new_rect = [rect[0] + p[0], rect[1] + p[1], rect[2] + p[0], rect[3] +␣
↪p[1]]
        rects.append(new_rect)
        # ===== 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
```

4.2.1 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.

[107]:
```
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');
```

[108]:
```
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])
```

4.2.2 Q2.3 (b) - Track Girl Sequence

Same as the car sequence.

[109]:
```
# 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])
```

LucasKanadeAffine

February 21, 2024

1 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.

[24]:
```
import time
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import scipy.signal
```

2 Download data

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

[25]:
```
# 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
```

3 Q3: Affine Motion Subtraction

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

[26]:
```
from scipy.interpolate import RectBivariateSpline
from scipy.ndimage import affine_transform

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␣
 ↪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! =====
    p = np.zeros(6)

    # Step1: create meshgrid
    H, W = It.shape
    X, Y = np.meshgrid(np.arange(W), np.arange(H))
    X_flat = X.ravel()
    Y_flat = Y.ravel()

    # Step2: interpolators
    spline_It = RectBivariateSpline(np.arange(H), np.arange(W), It)
    spline_It1 = RectBivariateSpline(np.arange(H), np.arange(W), It1)

    for _ in range(num_iters):
        M = np.array([[1+p[0], p[1], p[2]], [p[3], 1+p[4], p[5]]])
        warped_It1 = affine_transform(It1, M[:2, :2], offset=M[:2, 2],
↪output_shape=(H, W), order=1)

        It1_x = scipy.signal.convolve2d(warped_It1, np.array([[-1, 0, 1]]),
↪mode='same')
        It1_y = scipy.signal.convolve2d(warped_It1, np.array([[-1], [0], [1]]),
↪mode='same')

        # A matrix and p1 to p6
        A = np.zeros((H*W, 6))
        A[:, 0] = X_flat * It1_x.ravel()
        A[:, 1] = Y_flat * It1_x.ravel()
        A[:, 2] = It1_x.ravel()
        A[:, 3] = X_flat * It1_y.ravel()
        A[:, 4] = Y_flat * It1_y.ravel()
        A[:, 5] = It1_y.ravel()

        # b
        b = spline_It.ev(Y, X).ravel() - warped_It1.ravel()

        delta_p, _, _, _ = np.linalg.lstsq(A, b, rcond=None)
        p += delta_p

        if np.linalg.norm(delta_p) < threshold:
            break
```

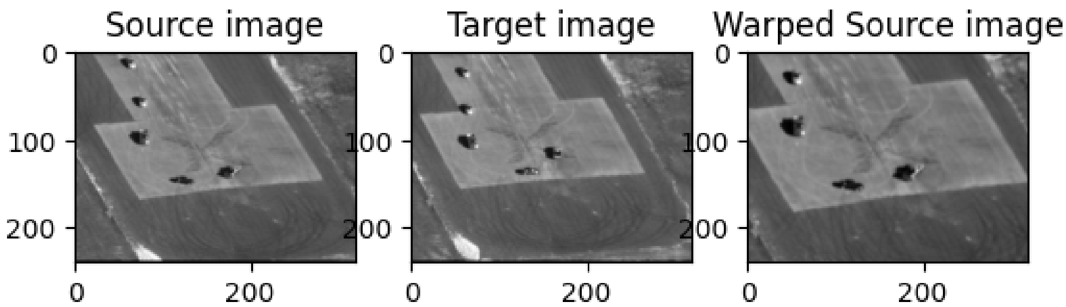
```
    M = np.array([[1+p[0], p[1], p[2]], [p[3], 1+p[4], p[5]]])

    # ===== End of code =====
    return M
```

3.2 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.

[27]:
```
import cv2

num_iters = 100
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 = 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')
```

[27]:
```
Text(0.5, 1.0, 'Warped Source image')
```



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

[28]:
```
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! =====
    # M = LucasKanadeAffine(It, It1, threshold, num_iters)
    # M_cv2 = M[:2, :]
    # transformed_It = cv2.warpAffine(It, M_cv2, (It.shape[1], It.shape[0]))

    # difference = It1 - transformed_It
    # high_difference_idx = np.abs(difference) > tolerance

    # mask[high_difference_idx] = True

    # mask = binary_erosion(mask, structure=np.ones((2, 1)), iterations=1)
```

```python
    # mask = binary_dilation(mask, iterations=1)

    M = LucasKanadeAffine(It, It1, threshold, num_iters)
    warped_It = cv2.warpAffine(It, M,(It.shape[1],It.shape[0]))
    diff_image = np.abs(warped_It - It1)
    mask = diff_image > tolerance
    mask = binary_erosion(mask, structure=np.ones((2, 1)))
    mask = binary_dilation(mask, iterations = 1)

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

    return mask
```

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

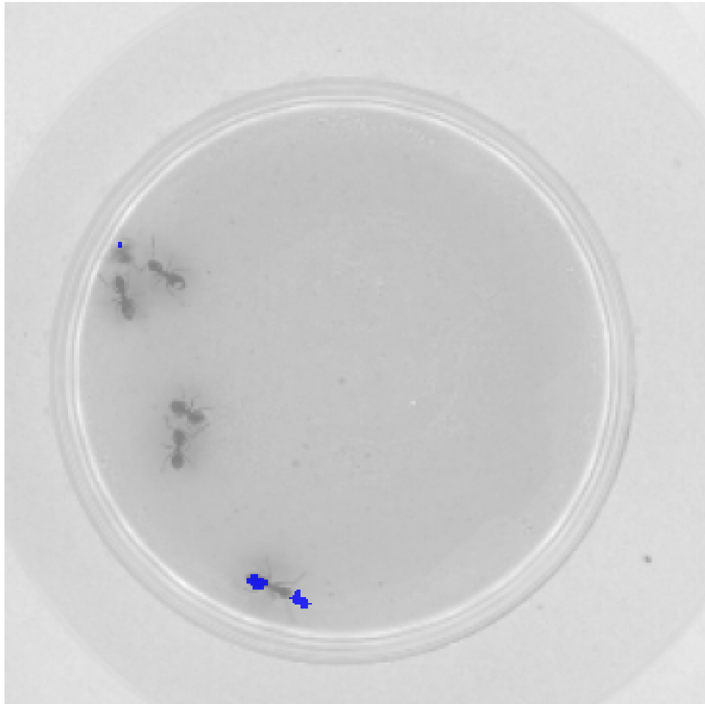
[29]:
```python
from tqdm import tqdm

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

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

    # ===== your code here! =====
    for i in tqdm(range(1, seq.shape[2])):
        It = seq[:, :, i-1]
        It1 = seq[:, :, i]

        mask = SubtractDominantMotion(It, It1, num_iters, threshold, tolerance)
        masks.append(mask)

    # ===== End of code =====
    masks = np.stack(masks, axis=2)
    return masks
```

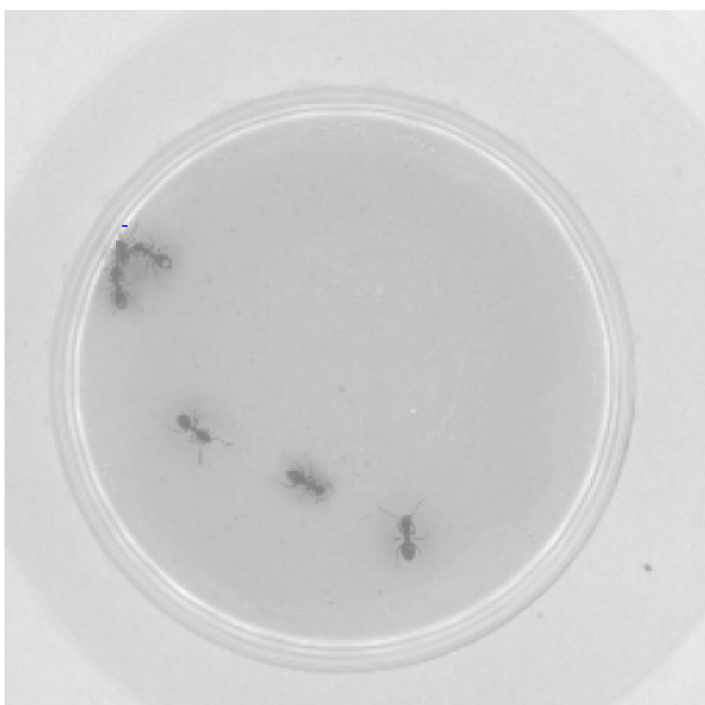
4.2 Q3.3 (a) - Track Ant Sequence

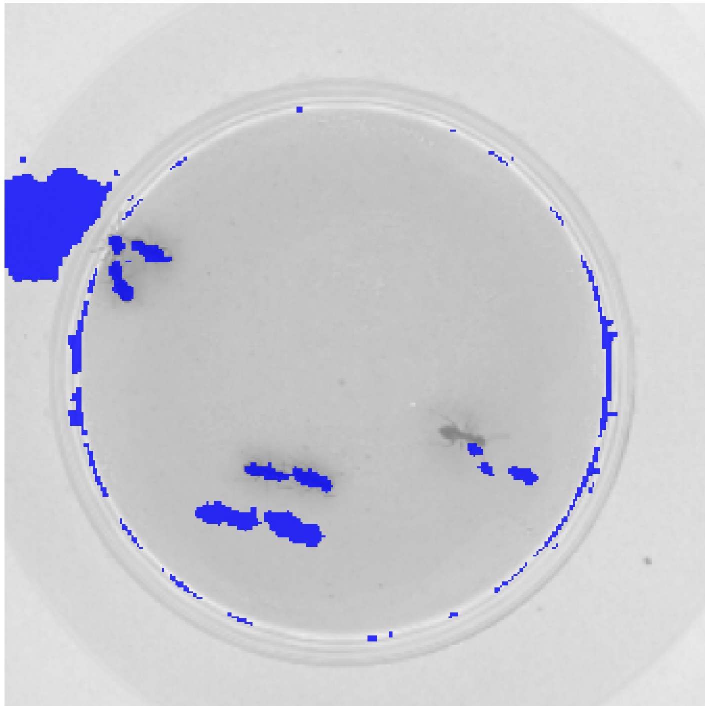
[30]:
```
seq = np.load("/content/antseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 100
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))
```

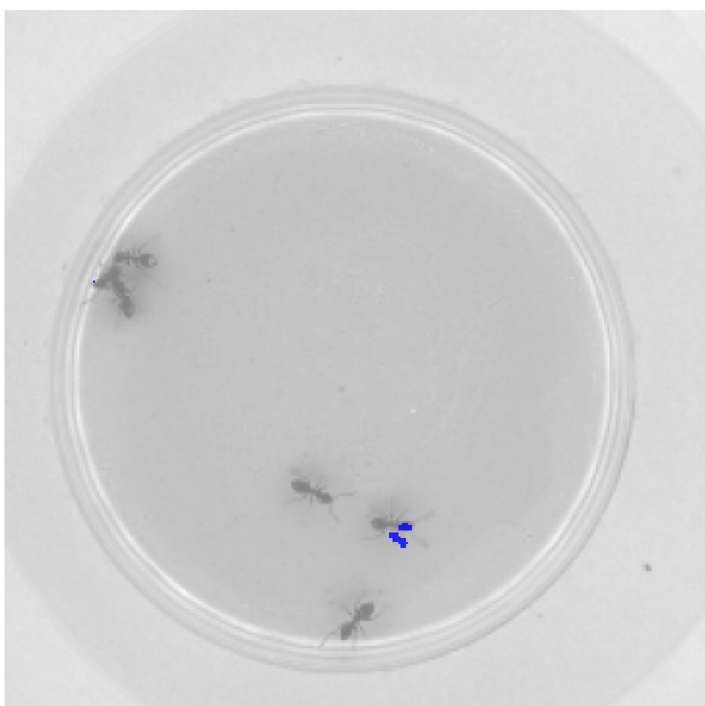
```
100%|          | 124/124 [08:16<00:00,  4.01s/it]

Ant Sequence takes 496.878964 seconds
```

[31]:
```
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')
```







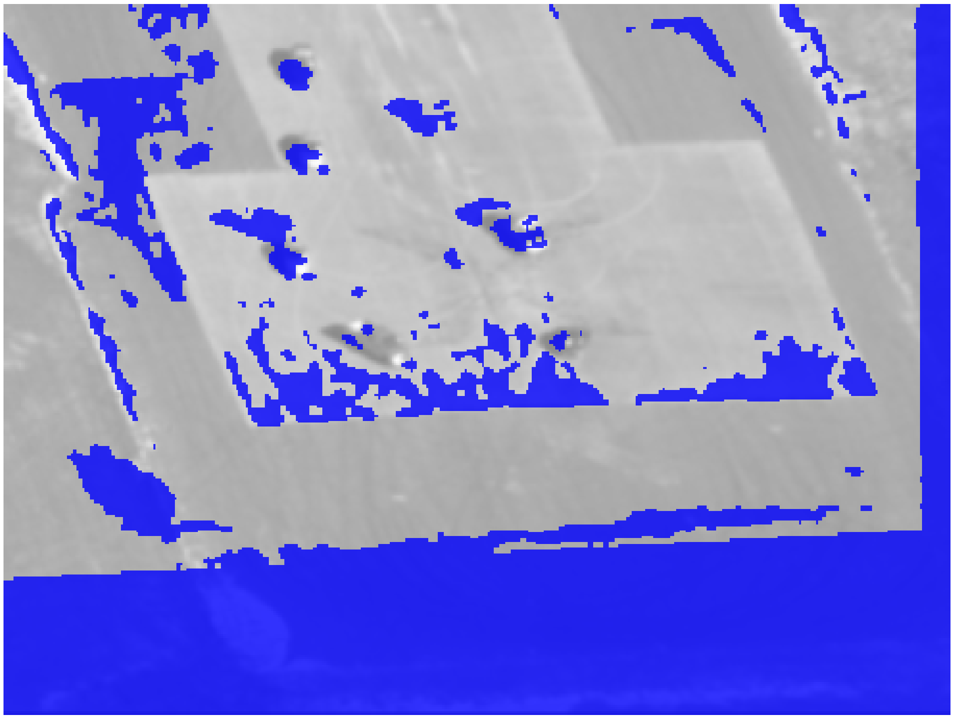


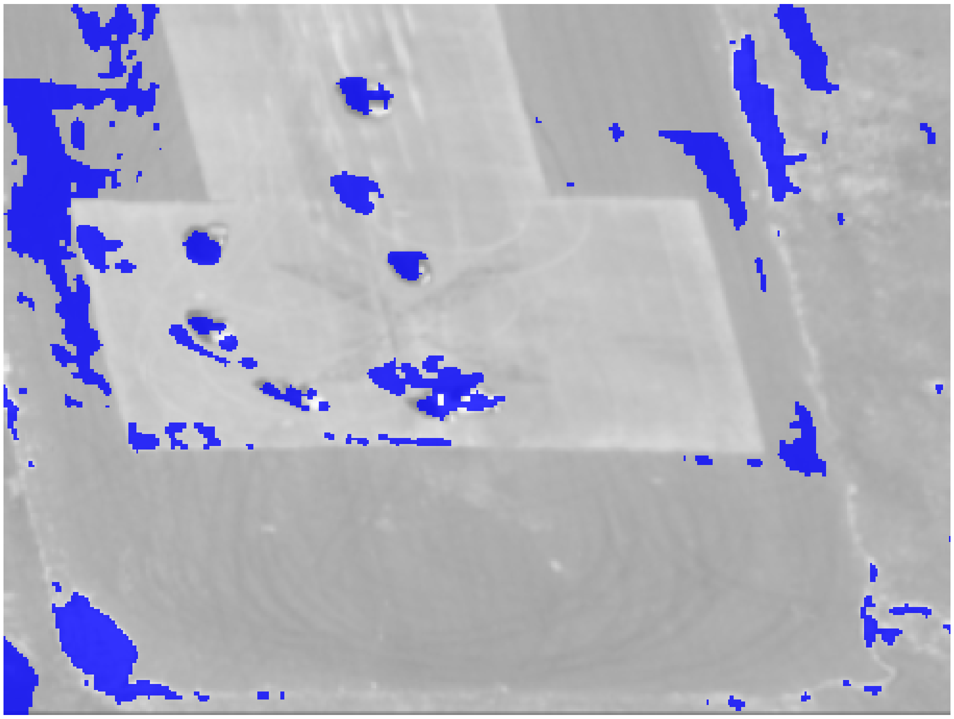
4.2.1 Q3.3 (b) - Track Aerial Sequence

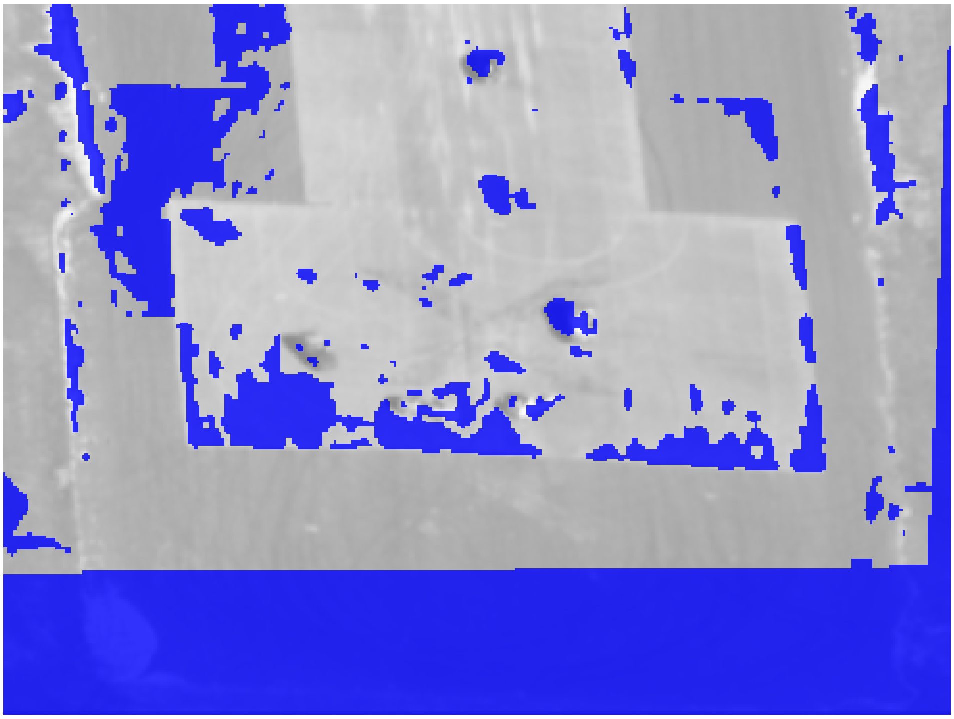
[32]:
```
seq = np.load("/content/aerialseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 100
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))
```

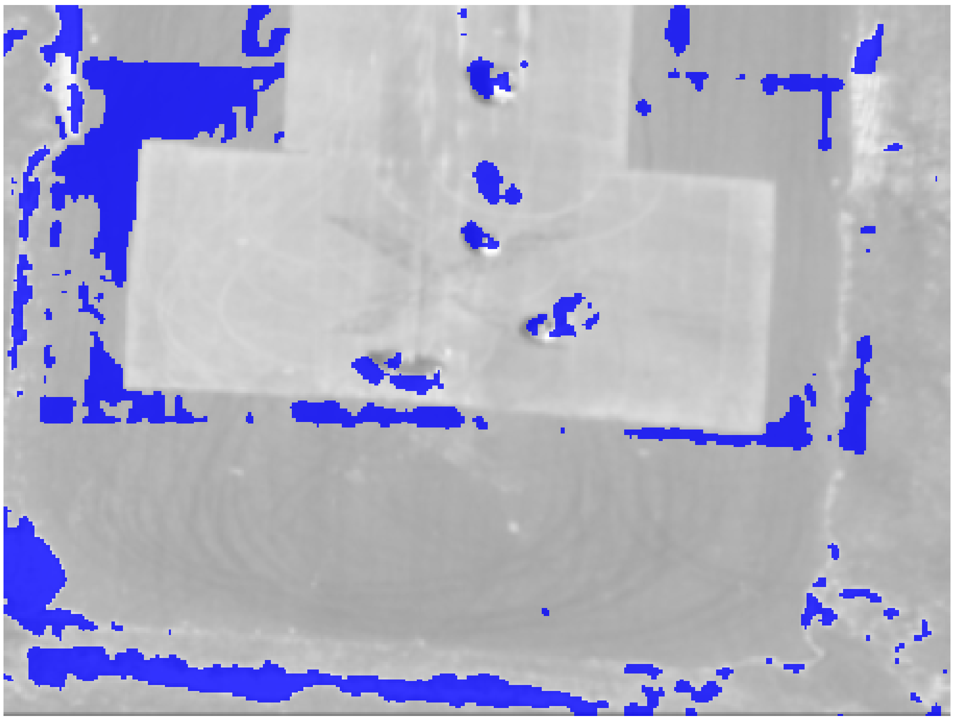
```
100%|      | 149/149 [23:57<00:00,  9.65s/it]


Ant Sequence takes 1437.214954 seconds
```

[33]:
```
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')
```









LucasKanadeEfficient

February 21, 2024

1 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.

[72]:
```
import time
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
```

2 Download data

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

[73]:
```
# 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
```

3 Q4: Efficient Tracking

3.1 Q4.1: Inverse Composition (15 points)

[109]:
```
from scipy.interpolate import RectBivariateSpline
from scipy.ndimage import affine_transform
from numpy.linalg import lstsq
import scipy.signal

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! =====
    H, W = It.shape
    x = np.arange(0, W)
    y = np.arange(0, H)
    X, Y = np.meshgrid(x, y)

    It_spline = RectBivariateSpline(np.arange(H), np.arange(W), It)
    It1_spline = RectBivariateSpline(np.arange(H), np.arange(W), It1)

    interped_gx = It_spline.ev(Y, X, dx=0, dy=1).flatten()
    interped_gy = It_spline.ev(Y, X, dx=1, dy=0).flatten()

    A = np.zeros((H*W, 6))
    X_flat, Y_flat = X.ravel(), Y.ravel()
    A[:, 0] = interped_gx * X_flat
    A[:, 1] = interped_gx * Y_flat
    A[:, 2] = interped_gx
    A[:, 3] = interped_gy * X_flat
    A[:, 4] = interped_gy * Y_flat
    A[:, 5] = interped_gy

    p = M.flatten()

    for _ in range(num_iters):
        X_warp = p[0] * X + p[1] * Y + p[2]
        Y_warp = p[3] * X + p[4] * Y + p[5]
        valid = (X_warp >= 0) & (X_warp < W) & (Y_warp >= 0) & (Y_warp < H)
        X_warp, Y_warp = X_warp[valid], Y_warp[valid]

        interped_I = It1_spline.ev(Y_warp, X_warp)

        A_valid = A[valid.flatten()]
        b = interped_I.flatten() - It[valid].flatten()

        delta_p, _, _, _ = lstsq(A_valid, b, rcond=None)

        if np.linalg.norm(delta_p) < threshold:
            break
```

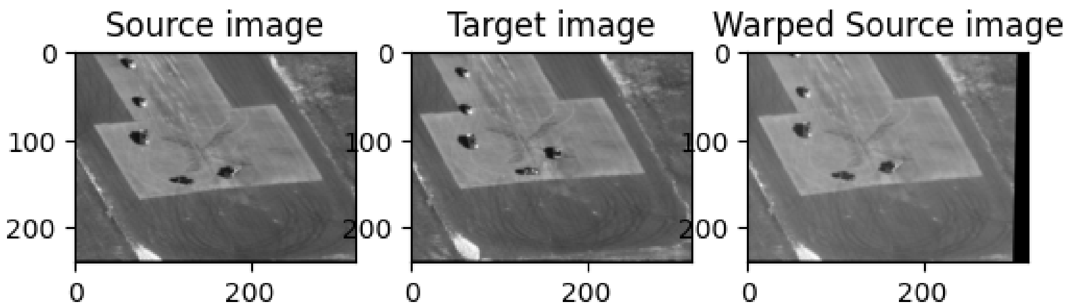
```python
        delta_M = np.array([[1 + delta_p[0], delta_p[1], delta_p[2]],
                            [delta_p[3], 1 + delta_p[4], delta_p[5]],
                            [0, 0, 1]])
        M = M @ np.linalg.inv(delta_M)
        p = M[:2, :].flatten()

    M = M[:2, :]
    # ===== End of code =====
    return M
```

3.2 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.

[110]:
```python
import cv2

num_iters = 100
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)
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')
```

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



3.3 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.

[111]:
```
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! =====
    M = InverseCompositionAffine(It, It1, threshold, num_iters)
    warped_It = cv2.warpAffine(It, M,(It.shape[1],It.shape[0]))
    diff_image = np.abs(warped_It - It1)
    mask = diff_image > tolerance
    mask = binary_erosion(mask, structure=np.ones((2, 1)))
    mask = binary_dilation(mask, iterations = 1)
    # ===== End of code =====

    return mask
```

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

[112]:
```
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

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

    # ===== your code here! =====
    for i in tqdm(range(1, seq.shape[2])):
        It = seq[:, :, i-1]
        It1 = seq[:, :, i]

        mask = SubtractDominantMotion(It, It1, num_iters, threshold, tolerance)
        masks.append(mask)

    # ===== End of code =====
    masks = np.stack(masks, axis=2)
    return masks
```

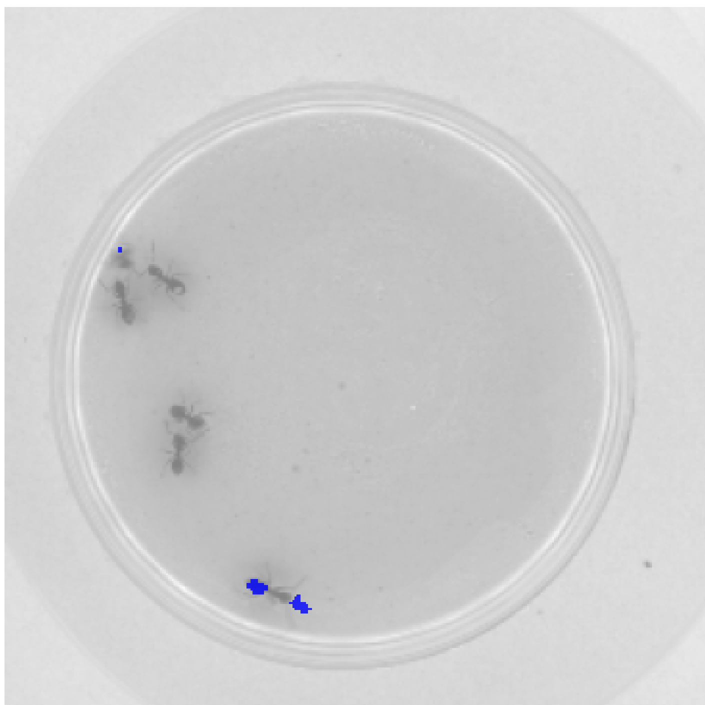
Track the ant sequence with inverse composition method.

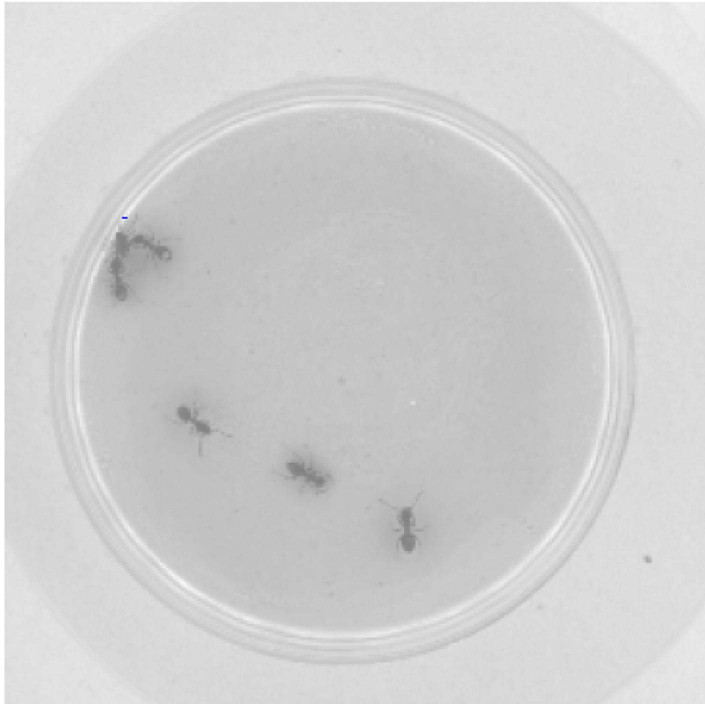
[113]:
```
seq = np.load("/content/antseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 100
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))
```

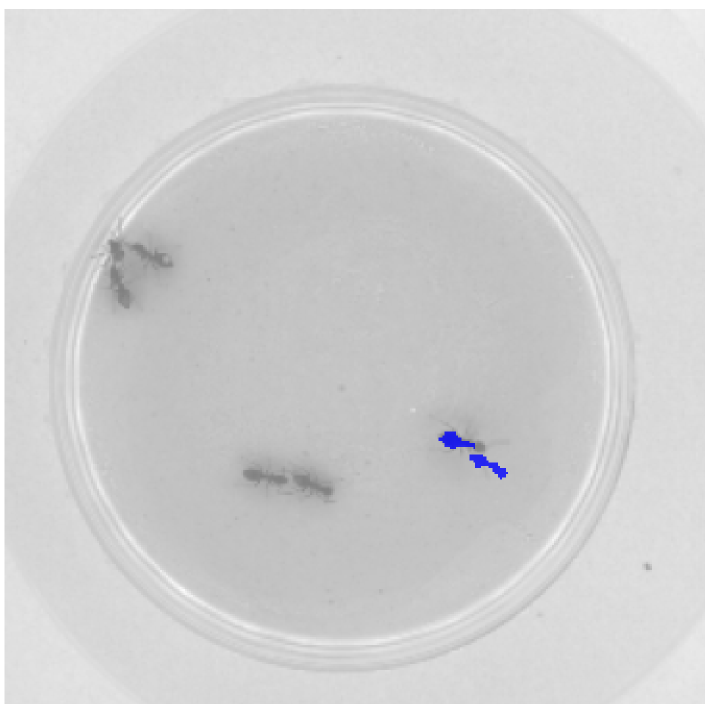
```
100%|      | 124/124 [00:42<00:00,  2.94it/s]
```

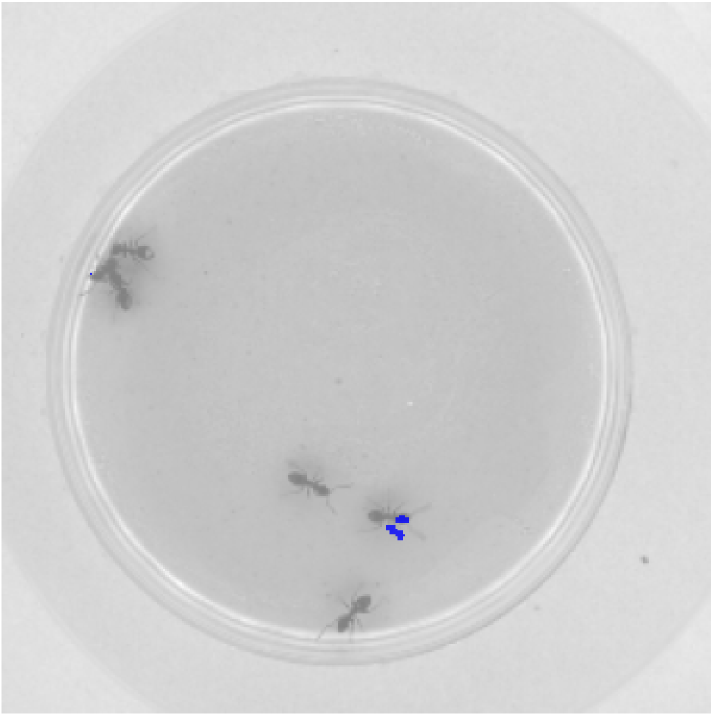
Ant Sequence takes 42.346663 seconds

[114]:
```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')
```









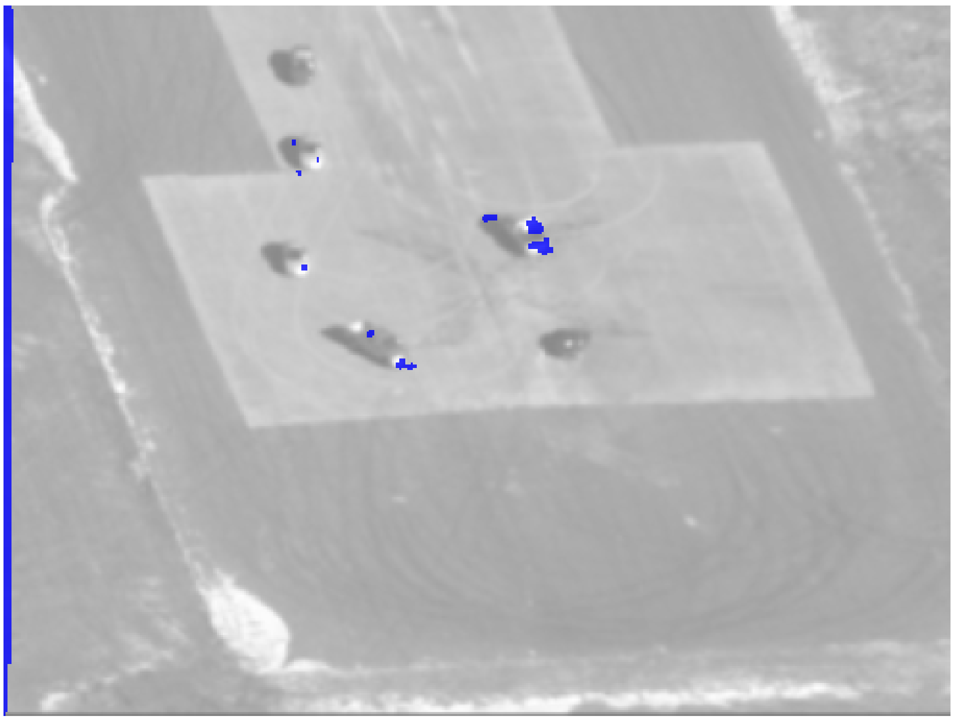
Track the aerial sequence with inverse composition method.

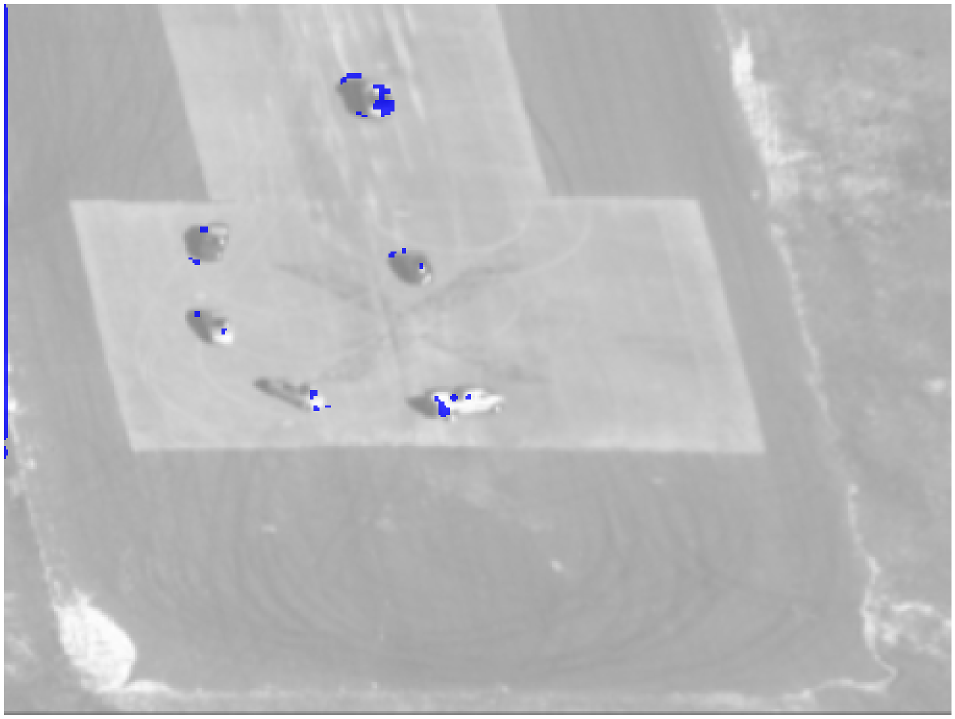
[115]:
```
seq = np.load("/content/aerialseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 100
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))
```

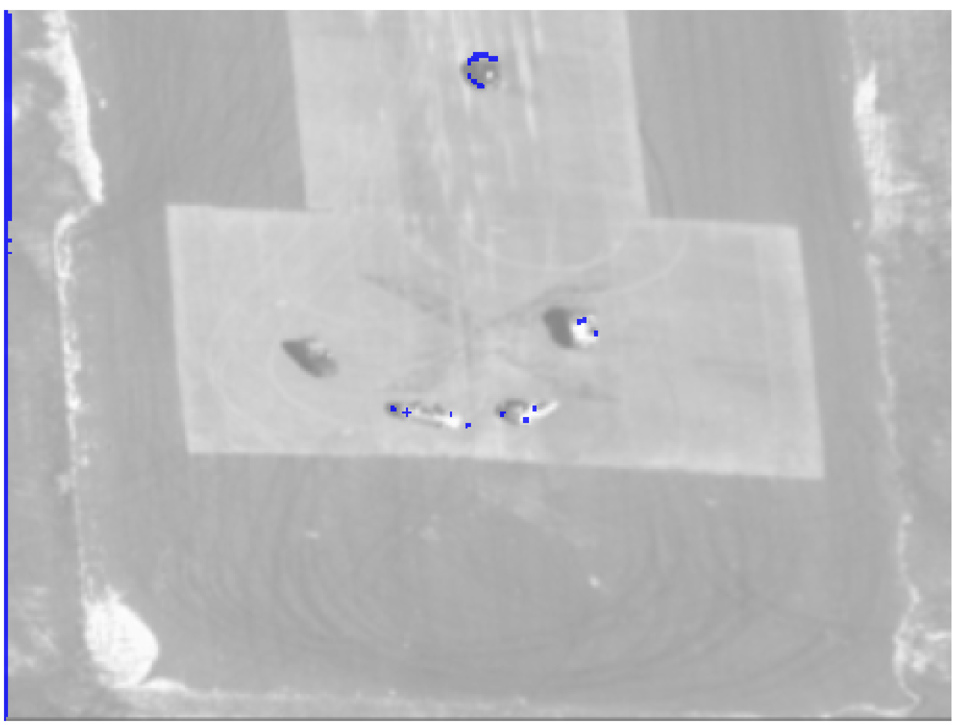
```
100%|          | 149/149 [02:00<00:00,  1.23it/s]

Ant Sequence takes 121.023149 seconds
```

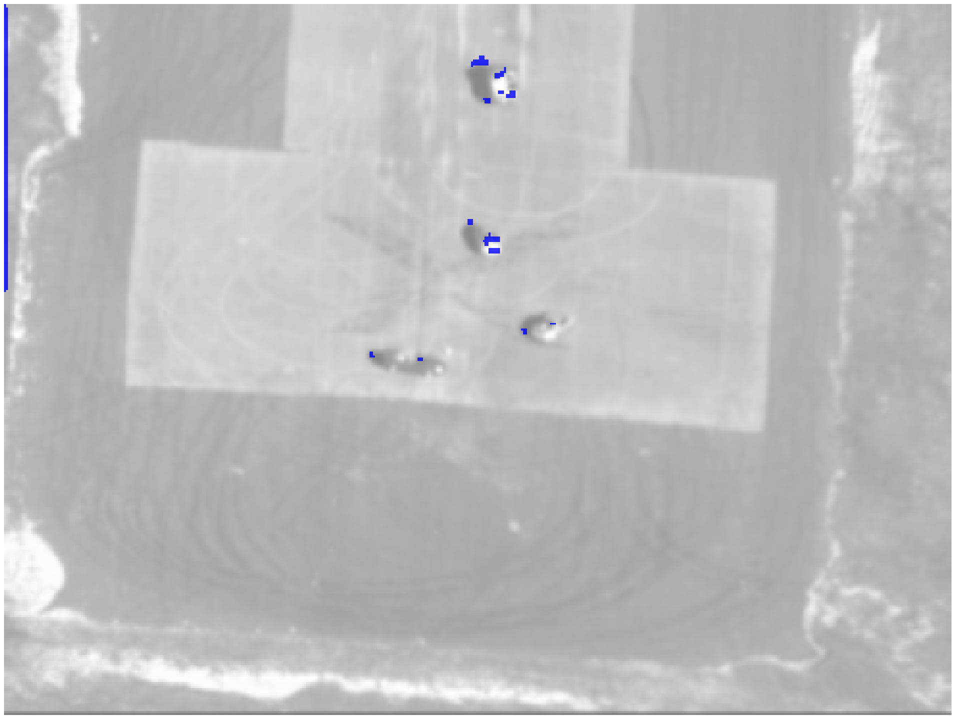
[116]:
```
frames_to_save = [29, 59, 89, 119]

# TODO: visualize
for idx in frames_to_save:
```

```python
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')
```









3.4 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! =====

For Ant Sequence Tracking, algorithm without inverse composition takes 1437.214954 seconds, algorithm using inverse composition takes 42.346663 seconds.

For Aerial Sequence Tracking, algorithm without inverse composition takes 496.878964 seconds, algorithm using inverse composition takes 121.023149 seconds.

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

3.5 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 inverse compositional approach, the gradient of the template image and the Hessian matrix is pre-computated since that the template image stays fixed, and the warp update is composed with the current warp estimate in the inverse direction.

Also, in inverse approach, updating the warp is typically simpler and can be computated faster, because they avoid the repeated computation of derivatives and matrix inversions on the changing warped image.

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