LucasKanade

February 15, 2025

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.

[1]:
```
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.

[2]:
```
# 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
```

```
--2025-02-16 00:29:51--  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   107KB/s    in 23m 10s

2025-02-16 00:53:02 (179 KB/s) - '/content/carseq.npy' saved
[254976128/254976128]

--2025-02-16 00:53:02--  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   592KB/s    in 48s

2025-02-16 00:53:51 (560 KB/s) - ‘/content/girlseq.npy’ saved
[27648128/27648128]
```

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! =====

The warp is a simple translation, each component of $\mathbf{W}$ is linear with respect to the corresponding parameter. Therefore, we have:

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

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

3.2 Q2.1.2: What is A and b?

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

A: The matrix

$$A$$

is defined as the Jacobian (with respect to the parameters) of the image intensity at the warped coordinates. Formally,

$$A = \nabla I_{t+1}(\mathbf{W}(\mathbf{x};\mathbf{p})) \cdot \frac{\partial \mathbf{W}(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T}.$$

Since

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

this simplifies to:

$$A = \nabla I_{t+1}(\mathbf{W}(\mathbf{x};\mathbf{p})).$$

For a template containing

$$D$$

pixels,

$$A$$

is a

$$D \times 2$$

matrix where each row corresponds to the spatial image gradients (in

$$x$$

and

$$y$$

) evaluated at the warped positions:

$$A = \begin{bmatrix} \nabla I_{t+1}(\mathbf{W}(\mathbf{x}_1; \mathbf{p})) \\ \nabla I_{t+1}(\mathbf{W}(\mathbf{x}_2; \mathbf{p})) \\ \vdots \\ \nabla I_{t+1}(\mathbf{W}(\mathbf{x}_D; \mathbf{p})) \end{bmatrix}.$$

b: The vector

$$b$$

represents the residual (or error) between the template

$$T_t$$

and the warped image

$$I_{t+1}$$

at the current estimate

$$\mathbf{p}$$

. It is given by:

$$b = T_t(\mathbf{x}) - I_{t+1}(\mathbf{W}(\mathbf{x}; \mathbf{p})),$$

where the subtraction is performed pixel-wise over all

$$D$$

pixels in the template.

===== 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! =====

In the least-squares formulation, we solve:

$$\arg\min_{\Delta\mathbf{p}} \|A\Delta\mathbf{p} - b\|_2^2,$$

which leads to the normal equations:

$$A^T A\Delta\mathbf{p} = A^T b.$$

A unique solution for

$$\Delta\mathbf{p}$$

exists if and only if

$$\mathbf{A}^T\mathbf{A}$$

is invertible. This requires that:

1. **Full Rank:** The matrix

   $$A$$

   must have full column rank in this 2D case, meaning that its two columns (corresponding to the

   $$x$$

   and

   $$y$$

   gradients) are linearly independent.

2. **Positive Definiteness:** As a consequence,

   $$\mathbf{A}^T\mathbf{A}$$

   is a

   $$2 \times 2$$

   positive definite matrix, ensuring it is nonsingular. This means that the image patch must have sufficient texture or variation in intensity in both directions; otherwise,

   $$\mathbf{A}^T\mathbf{A}$$

   may be singular.

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

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

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

[3]:
```
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
    # 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
    x1, y1, x2, y2 = rect
    # Determine width and height of the template (inclusive of both endpoints)
    width = int(round(x2 - x1)) + 1
    height = int(round(y2 - y1)) + 1

    # Create coordinate grids for the template region
```

```
    # x coordinates span from x1 to x2 and y coordinates from y1 to y2
    x = np.linspace(x1, x2, width)
    y = np.linspace(y1, y2, height)
    # Meshgrid produces X (columns) and Y (rows) arrays, each of shape (height, width)
    X, Y = np.meshgrid(x, y)

    # Extract the template T from image It using interpolation
    It_spline = RectBivariateSpline(np.arange(It.shape[0]), np.arange(It.shape[1]), It)
    T = It_spline.ev(Y, X)

    # Create an interpolation spline for image It1
    It1_spline = RectBivariateSpline(np.arange(It1.shape[0]), np.arange(It1.shape[1]), It1)

    for _ in range(num_iters):
        # Warp the template coordinates using the current parameters p
        # p[0] is the x offset and p[1] is the y offset
        X_warp = X + p[0]
        Y_warp = Y + p[1]

        # Evaluate the warped image at these coordinates
        Iw = It1_spline.ev(Y_warp, X_warp)

        # Compute the error image (difference between warped image and the template)
        error = Iw - T

        # Compute gradients of It1 at the warped coordinates.
        # Note: In RectBivariateSpline, the first argument corresponds to the y-coordinate (rows)
        # and the second corresponds to the x-coordinate (columns).
        # For the derivative with respect to x (columns), use dy=1 (and dx=0);
        # for y (rows), use dx=1 (and dy=0).
        Ix = It1_spline.ev(Y_warp, X_warp, dx=0, dy=1)
        Iy = It1_spline.ev(Y_warp, X_warp, dx=1, dy=0)

        # Build the matrix A by stacking the gradients at each pixel.
        # Each row of A is [Ix, Iy] at a pixel, and A has shape (D, 2)
        A = np.vstack((Ix.flatten(), Iy.flatten())).T

        # Solve for the update delta_p in the equation A * delta_p = -error.
        delta_p, _, _, _ = lstsq(A, -error.flatten(), rcond=None)

        # Update the parameters p
        p = p + delta_p
```

```
        # Check for convergence: if the norm of delta_p is below the threshold,␣
↪break out of the loop.
        if np.linalg.norm(delta_p) < threshold:
            break

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

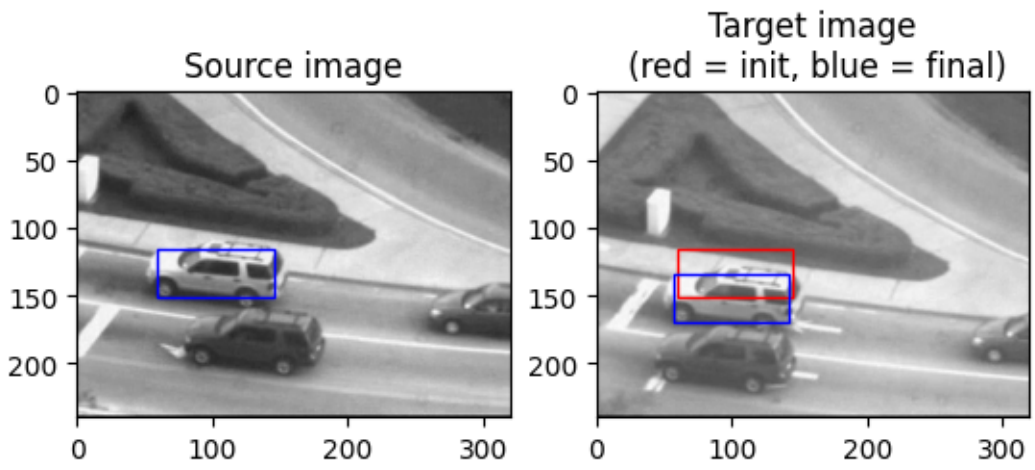
4.1 Debug Q2.2

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

[4]:
```
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'))
```

[5]:
```
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)
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)

[12]:
```
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:
            current_rect = np.array(rect).flatten()
            rects.append(current_rect.copy())
        else:
            It_prev = seq[:, :, i-1]
            It_curr = seq[:, :, i]
            p = LucasKanade(It_prev, It_curr, current_rect, threshold, num_iters, p0=np.zeros(2))
            current_rect = current_rect + np.array([p[0], p[1], p[0], p[1]])
```

```
            rects.append(current_rect.copy())
        # ===== 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.

[13]:
```
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');
```

[14]:
```
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.

[15]:
```
# 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 15, 2025

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.

[1]:
```
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.

[ ]:
```
# 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)

[2]:
```
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␣
 ↪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
    # Create a grid of (x,y) coordinates for the entire image (template)
    X, Y = np.meshgrid(np.arange(W), np.arange(H))
    X_flat = X.flatten()
    Y_flat = Y.flatten()

    # Create splines for interpolation on It and It1
    It_spline = RectBivariateSpline(np.arange(H), np.arange(W), It)
    It1_spline = RectBivariateSpline(np.arange(H), np.arange(W), It1)

    # Initialize the affine parameters: p = [p1, p2, p3, p4, p5, p6]
    # The affine warp is: W(x; p) = [ (1+p1)*x + p2*y + p3; p4*x + (1+p5)*y +
↪p6 ]
    p_vec = np.zeros(6)

    for _ in range(num_iters):
        # Construct the current affine warp matrix from p_vec
        M = np.array([[1 + p_vec[0], p_vec[1], p_vec[2]],
                      [p_vec[3],     1 + p_vec[4], p_vec[5]]])

        # Warp the coordinates of the template using M
        X_warp = M[0, 0] * X_flat + M[0, 1] * Y_flat + M[0, 2]
        Y_warp = M[1, 0] * X_flat + M[1, 1] * Y_flat + M[1, 2]

        # Determine the valid coordinates that fall within the bounds of It1
        valid_idx = (X_warp >= 0) & (X_warp <= W - 1) & (Y_warp >= 0) & (Y_warp
↪<= H - 1)
        if np.sum(valid_idx) == 0:
            break

        # Select only the valid points
        X_valid = X_flat[valid_idx]
        Y_valid = Y_flat[valid_idx]
        X_warp_valid = X_warp[valid_idx]
        Y_warp_valid = Y_warp[valid_idx]

        # Evaluate the warped image It1 at the valid warped coordinates
        I1_warp = It1_spline.ev(Y_warp_valid, X_warp_valid)
        # Evaluate the template image It at the valid coordinates (using
↪interpolation)
        T_valid = It_spline.ev(Y_valid, X_valid)
```

```
        # Compute the error between the template and the warped image
        error = T_valid - I1_warp  # shape (num_valid,)

        # Compute the gradients of It1 at the warped coordinates
        I1_dx = It1_spline.ev(Y_warp_valid, X_warp_valid, dx=0, dy=1)
        I1_dy = It1_spline.ev(Y_warp_valid, X_warp_valid, dx=1, dy=0)

        # Compute the Jacobian of the warp with respect to the affine␣
↪parameters for each valid pixel.
        # For a pixel (x, y), the Jacobian is:
        # [ x, y, 1, 0, 0, 0 ]
        # [ 0, 0, 0, x, y, 1 ]
        # Therefore, the steepest descent images are:
        # [ I1_dx*x, I1_dx*y, I1_dx, I1_dy*x, I1_dy*y, I1_dy ]
        A = np.vstack((I1_dx * X_valid,
                       I1_dx * Y_valid,
                       I1_dx,
                       I1_dy * X_valid,
                       I1_dy * Y_valid,
                       I1_dy)).T  # Shape: (num_valid, 6)

        # Solve for the parameter update dp such that A dp = error in a␣
↪least-squares sense
        dp, _, _, _ = np.linalg.lstsq(A, error, rcond=None)

        # Update the affine parameters
        p_vec = p_vec + dp

        # Check for convergence
        if np.linalg.norm(dp) < threshold:
            # Update M one last time before breaking out
            M = np.array([[1 + p_vec[0], p_vec[1], p_vec[2]],
                          [p_vec[3],     1 + p_vec[4], p_vec[5]]])
            break
    # ===== End of code =====

    return M
```

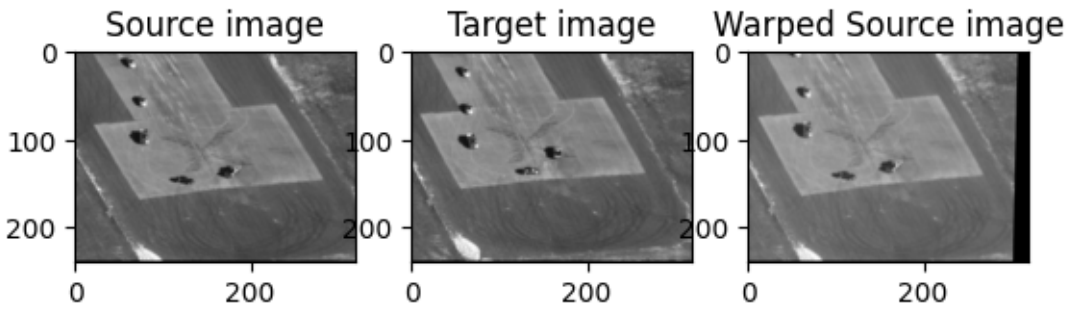
3.2 Debug Q3.1

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.

[3]:
```
import cv2
```

```
num_iters = 100
threshold = 0.01
seq = np.load("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')
```

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



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

[4]:
```
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! =====
        # Compute the dominant affine transformation matrix M using␣
 ↪LucasKanadeAffine
    M = LucasKanadeAffine(It, It1, threshold, num_iters)

    # Warp image It to align with It1 using the computed affine transformation␣
 ↪M.
    # Note: cv2.warpAffine expects the size in (width, height) order.
    H, W = It.shape
    It_warped = cv2.warpAffine(It, M, (W, H))

    # Compute the absolute difference between the warped image and It1.
    diff = np.abs(It1 - It_warped)

    # Threshold the difference to create a binary mask of moving regions.
    mask = diff > tolerance

    # Optionally, refine the mask using morphological operations:
    # Apply binary erosion followed by dilation to remove noise.
    mask = binary_dilation(binary_erosion(mask))
    # ===== End of code =====

    return mask
```

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

[5]:
```
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]

    # ===== your code here! =====
    masks = []
    for i in tqdm(range(1, seq.shape[2])):
        It_current = seq[:, :, i-1]
        It_next = seq[:, :, i]
        current_mask = SubtractDominantMotion(It_current, It_next, num_iters, threshold, tolerance)
        masks.append(current_mask)
    # ===== End of code =====
    masks = np.stack(masks, axis=2)
    return masks
```

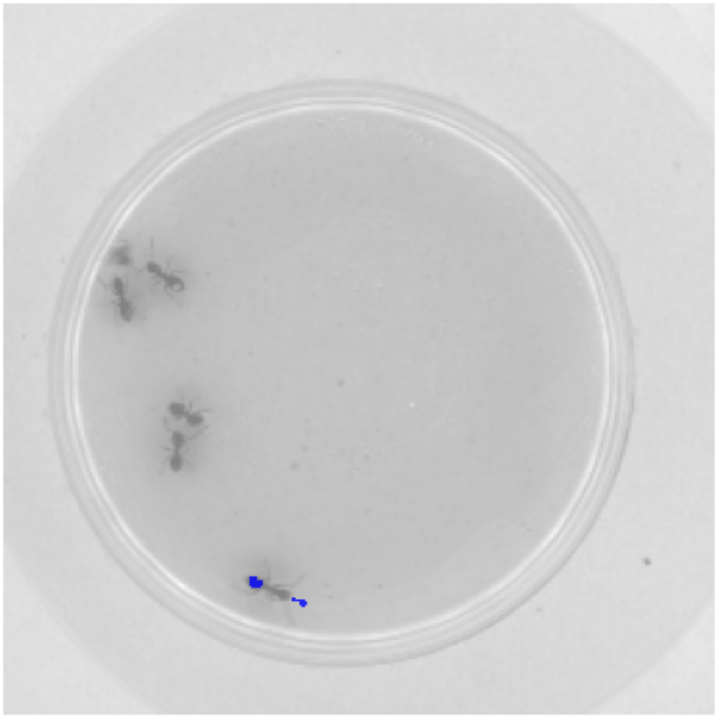
4.2 Q3.3 (a) - Track Ant Sequence

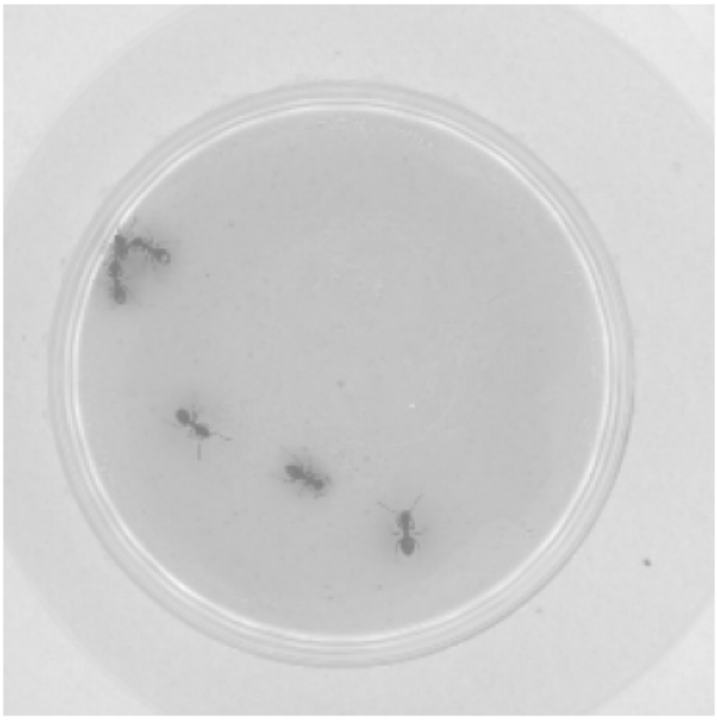
[6]:
```
seq = np.load("antseq.npy")

# 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))
```

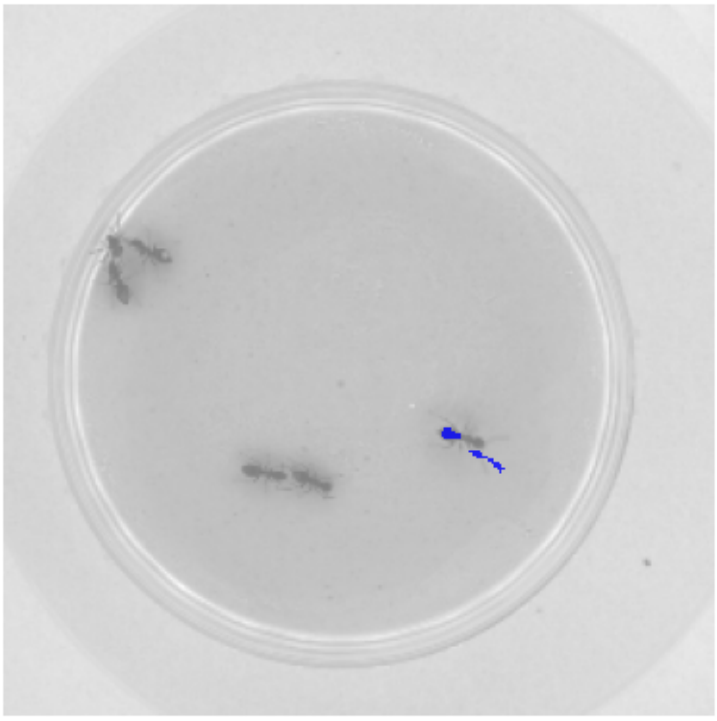
```
100%|      | 124/124 [00:35<00:00,  3.46it/s]


Ant Sequence takes 35.958883 seconds
```

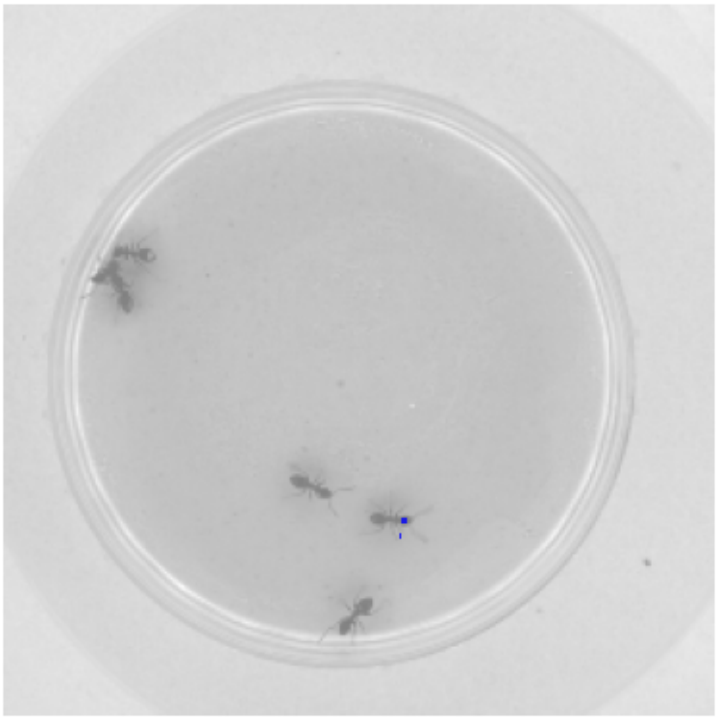
[7]:
```
frames_to_save = [29, 59, 89, 119]

# 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')
```









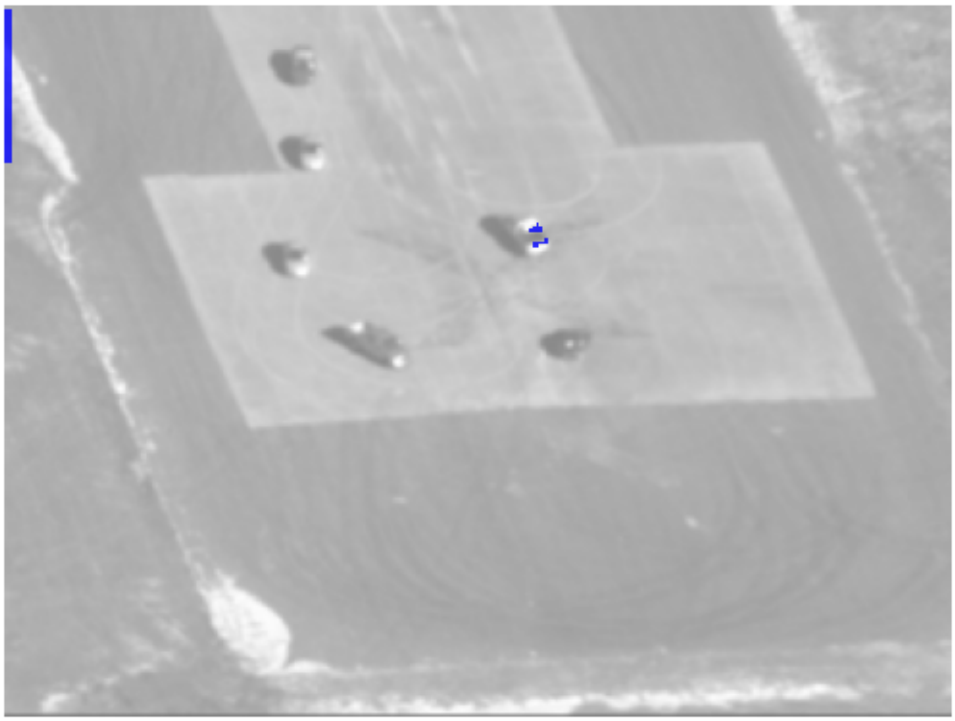
4.2.1 Q3.3 (b) - Track Aerial Sequence

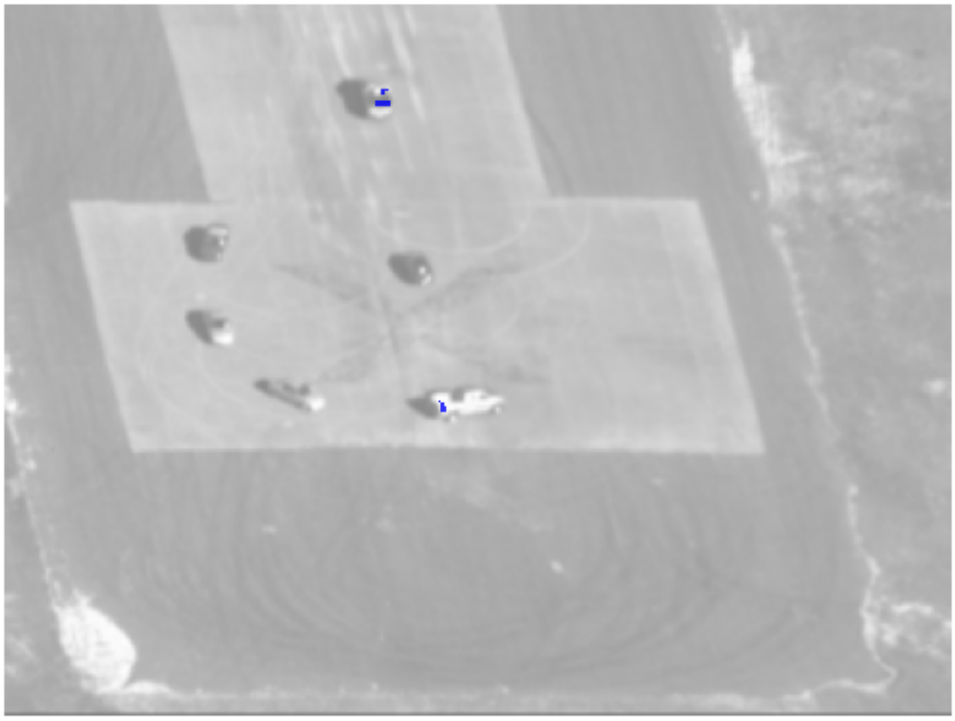
[8]:
```
seq = np.load("aerialseq.npy")

# 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))
```

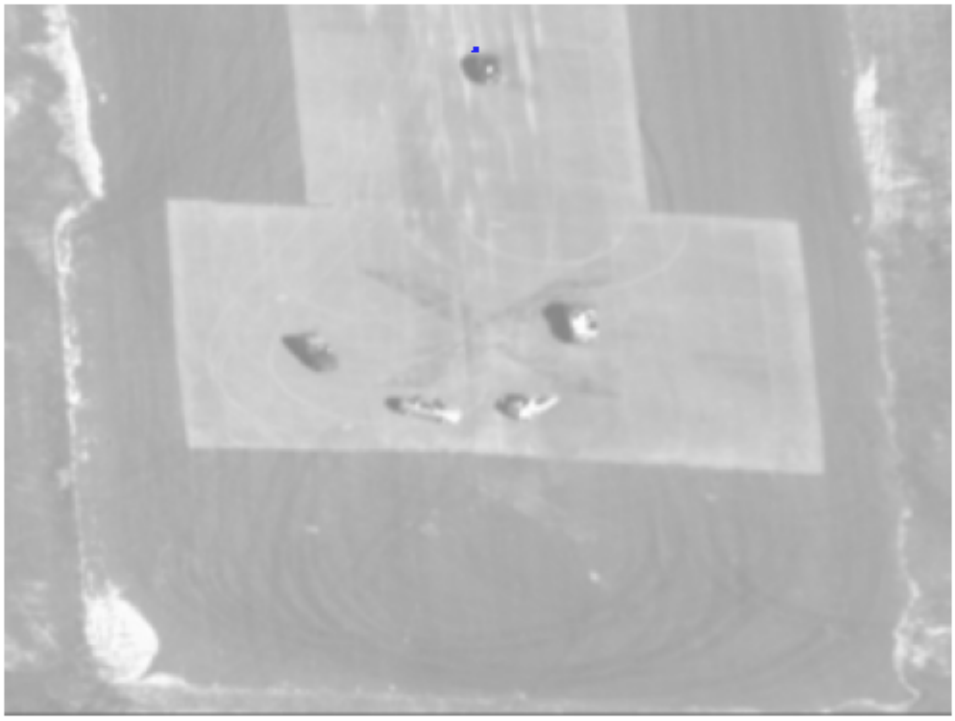
```
100%|      | 149/149 [01:22<00:00,  1.80it/s]

Aerial Sequence takes 82.882023 seconds
```

[9]:
```
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 15, 2025

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.

[4]:
```
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.

[ ]:
```
# 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)

[18]:
```
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! =====
    H, W = It.shape
    # Create a grid of (x, y) coordinates for the entire template image It
    X, Y = np.meshgrid(np.arange(W), np.arange(H))
    X_flat = X.flatten()
    Y_flat = Y.flatten()
    T = It.flatten()  # Template image (vectorized)

    # Compute the gradients of the template (It). Note: np.gradient returns [T/
↪ y,  T/ x]
    T_dy, T_dx = np.gradient(It)
    T_dx_flat = T_dx.flatten()
    T_dy_flat = T_dy.flatten()

    # Pre-compute the steepest descent images using the Jacobian at the␣
↪identity warp.
    # For an affine warp parameterized as:
    #   x' = (1 + p1)*x + p2*y + p3,
    #   y' = p4*x + (1 + p5)*y + p6,
    # the Jacobian at p=0 is:
    #   [ x,   y,   1,   0,   0,   0 ]
    #   [ 0,   0,   0,   x,   y,   1 ]
    # Therefore, for each pixel, the steepest descent images are:
    #   [T_dx*x, T_dx*y, T_dx, T_dy*x, T_dy*y, T_dy]
    SD = np.vstack((T_dx_flat * X_flat,
                    T_dx_flat * Y_flat,
                    T_dx_flat,
                    T_dy_flat * X_flat,
                    T_dy_flat * Y_flat,
                    T_dy_flat)).T  # Shape: (n_pixels, 6)

    # Compute the Hessian matrix (which is constant) and its inverse
    H_matrix = SD.T @ SD  # (6 x 6)
    H_inv = np.linalg.inv(H_matrix)

    # Create an interpolation spline for the current image It1
    It1_spline = RectBivariateSpline(np.arange(H), np.arange(W), It1)

    # Initialize the affine warp M as the 2x3 identity matrix
    M = np.array([[1.0, 0.0, 0.0],
                  [0.0, 1.0, 0.0]])
```

```
    for _ in range(num_iters):
        # Warp the template coordinates using the current affine warp M
        X_warp = M[0, 0] * X_flat + M[0, 1] * Y_flat + M[0, 2]
        Y_warp = M[1, 0] * X_flat + M[1, 1] * Y_flat + M[1, 2]

        # Determine the valid pixels that fall within the bounds of It1
        valid_idx = (X_warp >= 0) & (X_warp <= W - 1) & (Y_warp >= 0) & (Y_warp␣
↪<= H - 1)
        if np.sum(valid_idx) == 0:
            break  # Exit if no pixels are valid

        # Select only the valid pixels for computing the error
        T_valid = T[valid_idx]
        SD_valid = SD[valid_idx]
        X_warp_valid = X_warp[valid_idx]
        Y_warp_valid = Y_warp[valid_idx]

        # Interpolate the warped image It1 at the valid warped coordinates
        I_warp_valid = It1_spline.ev(Y_warp_valid, X_warp_valid)

        # Compute the error between the warped current image and the template.
        # In the inverse compositional formulation, we linearize around the␣
↪template,
        # so the error is: error = I(W(x; p)) - T(x)
        error = I_warp_valid - T_valid

        # Compute the parameter update dp using the pre-computed Hessian and␣
↪the steepest descent images
        dp = H_inv @ (SD_valid.T @ error)

        # Check for convergence: if the norm of dp is below the threshold,␣
↪terminate the loop.
        if np.linalg.norm(dp) < threshold:
            break

        # Form the incremental warp ΔM corresponding to dp:
        # ΔM = [ [1+dp1,    dp2,    dp3],
        #        [  dp4,  1+dp5,    dp6],
        #        [   0,      0,      1 ] ]
        delta_M = np.array([[1 + dp[0], dp[1], dp[2]],
                            [dp[3], 1 + dp[4], dp[5]],
                            [0, 0, 1]])

        # Update the warp: M ← M * (ΔM)⁻¹  (inverse composition update)
        delta_M_inv = np.linalg.inv(delta_M)
        M_extended = np.vstack((M, np.array([0, 0, 1])))
        M_updated = M_extended @ delta_M_inv
```

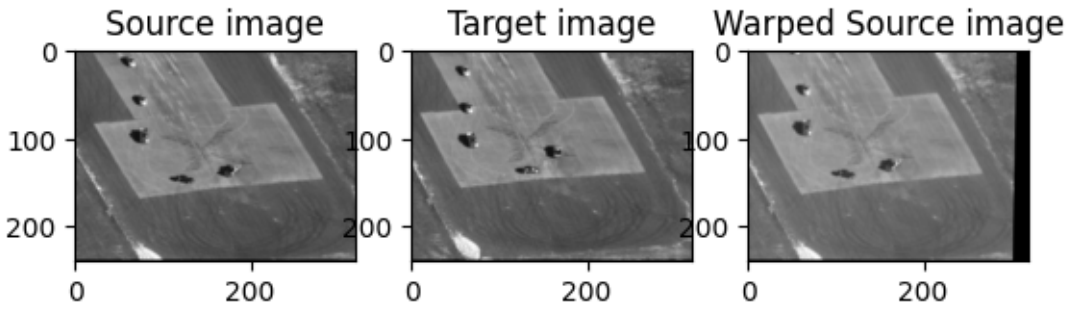
```
        M = M_updated[0:2, :]
    # ===== End of code =====
    return M
```

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.

[19]:
```
import cv2

num_iters = 100
threshold = 0.01
seq = np.load("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')
```

[19]:
```
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.

[20]:
```
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! =====
        # Compute the dominant affine transformation matrix M using␣
 ↪LucasKanadeAffine
    M = InverseCompositionAffine(It, It1, threshold, num_iters)

    # Warp image It to align with It1 using the computed affine transformation␣
 ↪M.
    # Note: cv2.warpAffine expects the size in (width, height) order.
    H, W = It.shape
    It_warped = cv2.warpAffine(It, M, (W, H))

    # Compute the absolute difference between the warped image and It1.
    diff = np.abs(It1 - It_warped)

    # Threshold the difference to create a binary mask of moving regions.
    mask = diff > tolerance

    # Optionally, refine the mask using morphological operations:
    # Apply binary erosion followed by dilation to remove noise.
    mask = binary_dilation(binary_erosion(mask))
```

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

    return mask
```

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

[21]:
```
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]

    # ===== your code here! =====
    masks = []
    for i in tqdm(range(1, seq.shape[2])):
        It_current = seq[:, :, i-1]
        It_next = seq[:, :, i]
        current_mask = SubtractDominantMotion(It_current, It_next, num_iters,␣
 ↪threshold, tolerance)
        masks.append(current_mask)
    # ===== End of code =====
    masks = np.stack(masks, axis=2)
    return masks
```

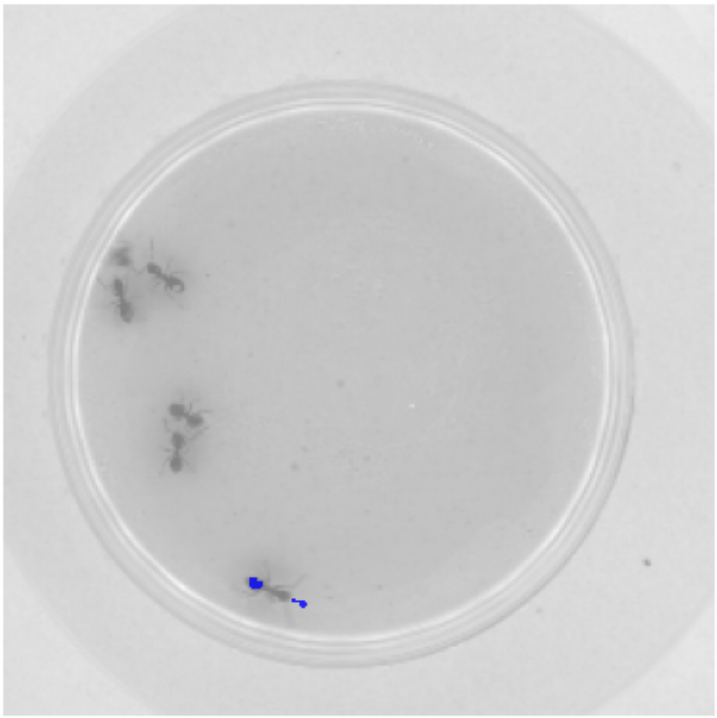
Track the ant sequence with inverse composition method.

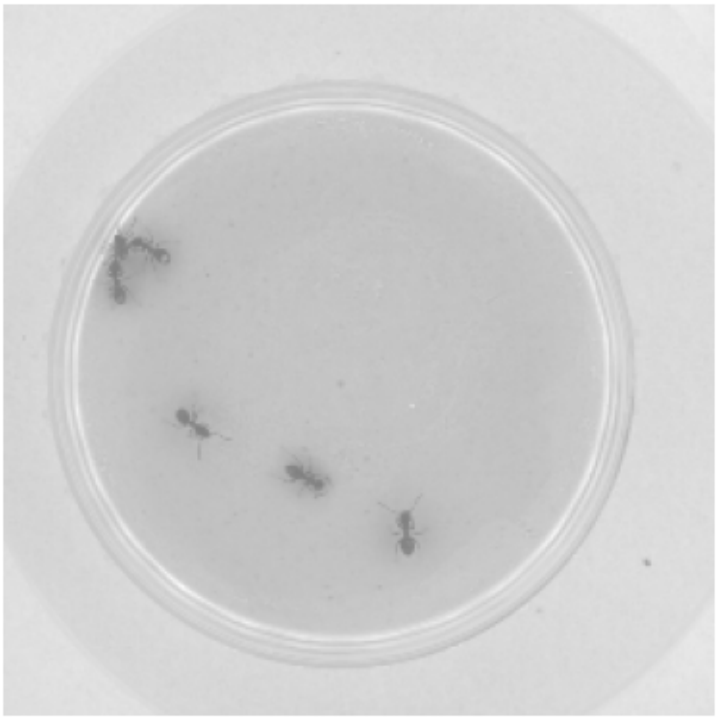
[22]:
```
seq = np.load("antseq.npy")

# 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()
```

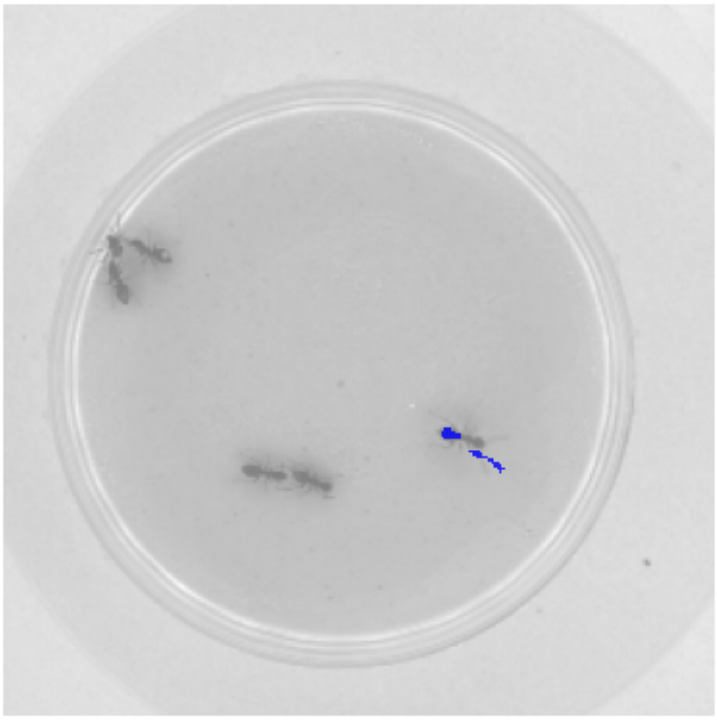
```
print('\nAnt Sequence takes %f seconds' % (toc - tic))
```

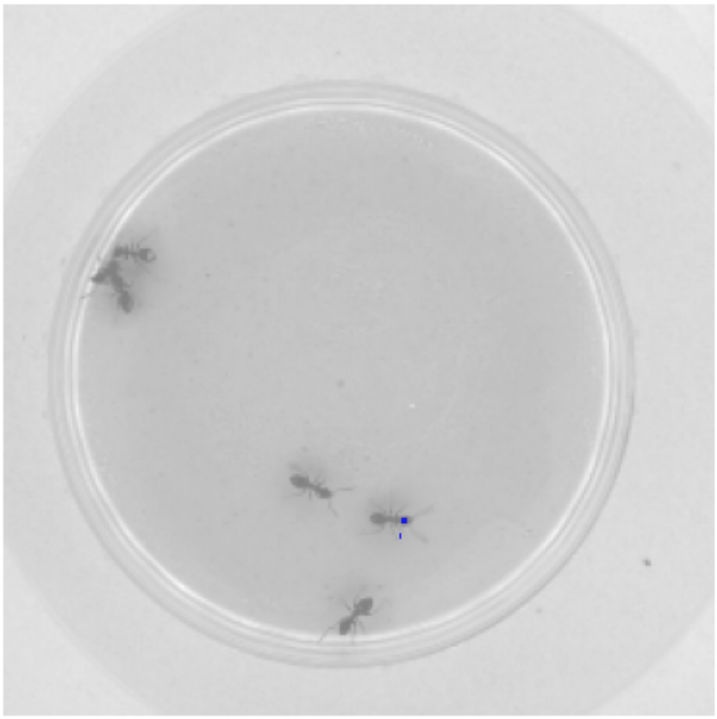
```
100%|      | 124/124 [00:09<00:00, 12.92it/s]

Ant Sequence takes 9.664032 seconds
```

[23]:
```
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')
```









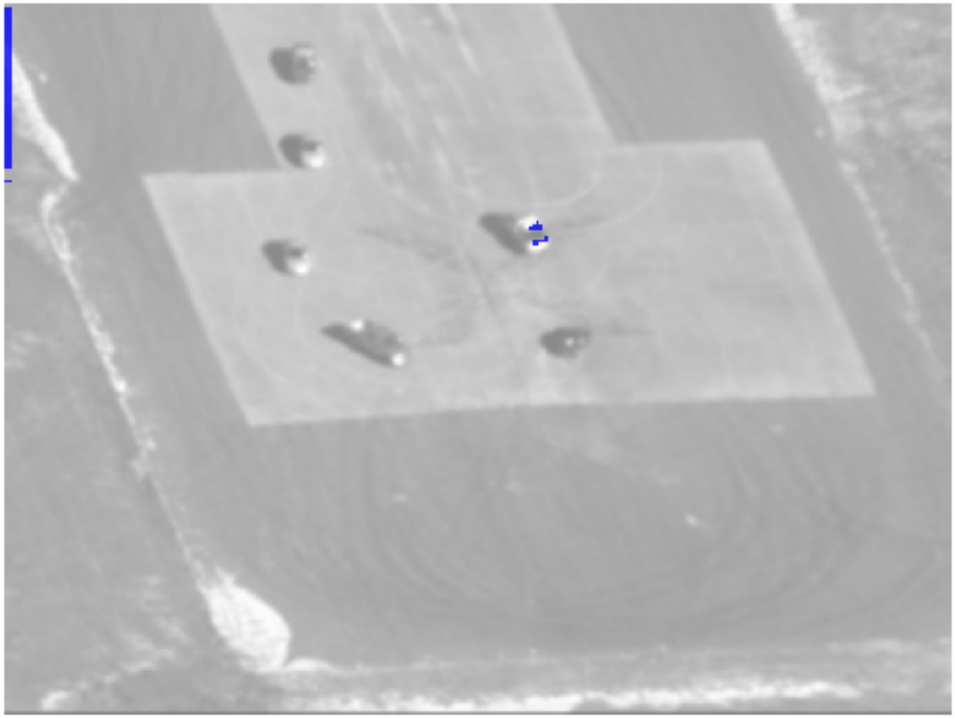
Track the aerial sequence with inverse composition method.

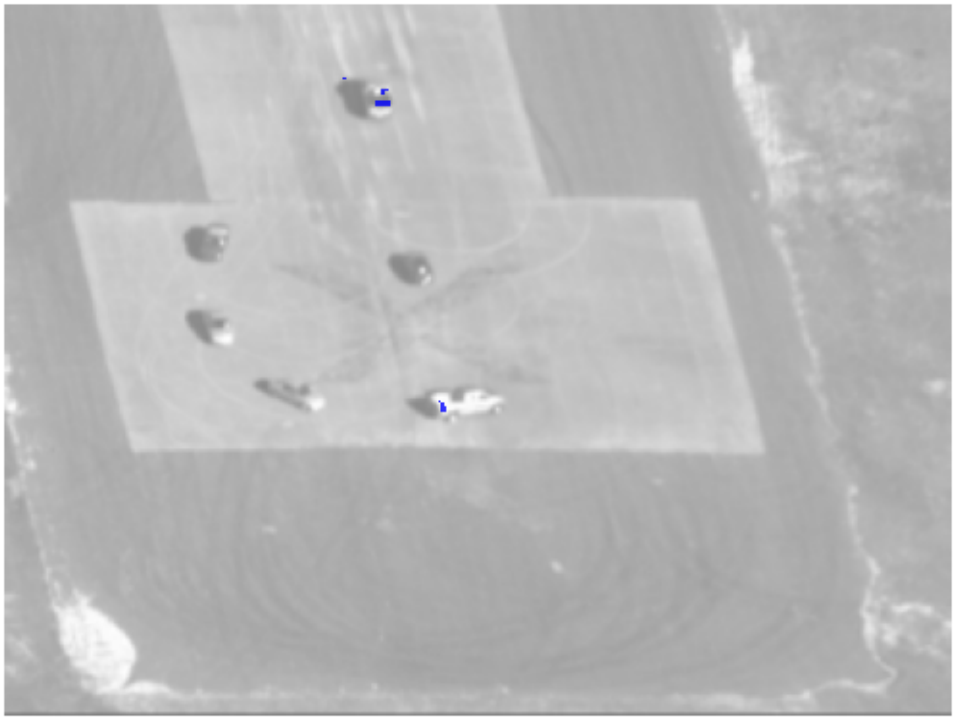
[24]:
```
seq = np.load("aerialseq.npy")

# 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))
```

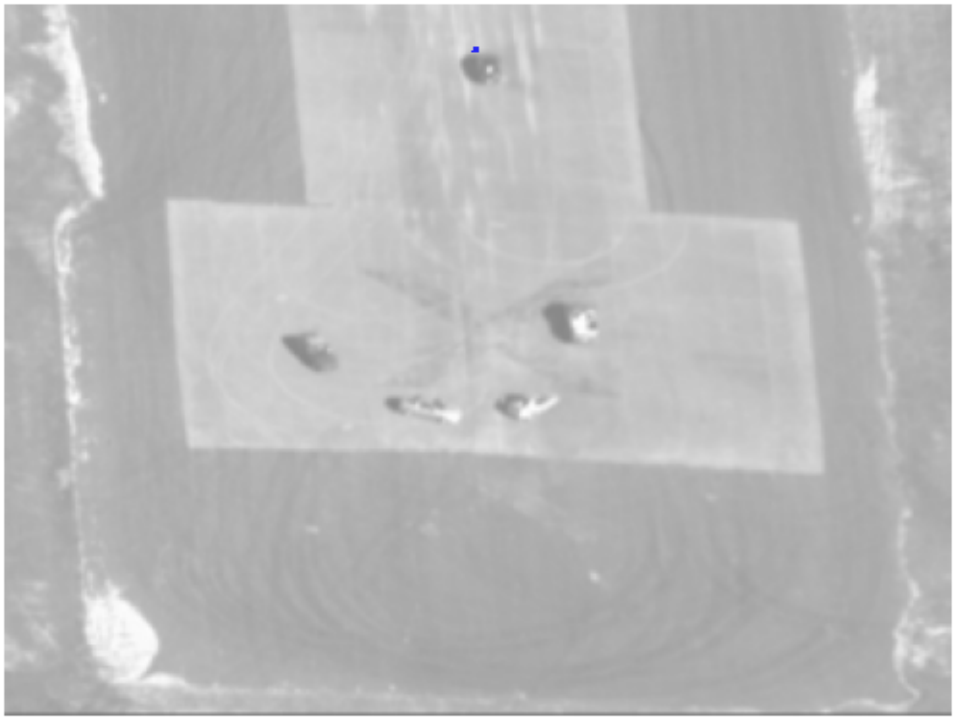
```
100%|      | 149/149 [00:24<00:00,  6.13it/s]

Ant Sequence takes 24.411206 seconds
```

[25]:
```
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')
```







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! ===== 1. **W/o Inverse Composition:** - Ant Sequence: 35.96s - Aerial Sequence: 82.88s

2. **Inverse Composition:**

- Ant Sequence: 9.66s
- Aerial Sequence: 24.11s

The inverse composition algorithm executed the processing of antseq.npy and aerialseq.npy roughly 350% faster on average.

===== 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! ===== The major reason for this speed up is the precomputation. By precomputing the template gradients, steepest descent images, and Hessian matrix because the linearizatio is about the fixed template, we don't have to recompute them on every iteration and

can focus only on the things that changed in the image, greatly speeding up the runtime. ===== end of your answer ====