Initialization

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

In [18]:
```
import time
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
```

Download data

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

In [19]:
```
# 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
```

Q4: Efficient Tracking

Q4.1: Inverse Composition (15 points)

In [20]:
```
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, termi
nate the optimization
    :param num_iters : (int), number of iterations for running the opt
imization

    :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! =====


    # Image dimensions
    H, W = It.shape
    Y, X = np.meshgrid(np.arange(W), np.arange(H), indexing='xy')

    # Spline interpolations for the images
    spline_It = RectBivariateSpline(np.arange(H), np.arange(W), It)
    spline_It1 = RectBivariateSpline(np.arange(H), np.arange(W), It1)

    coords = np.vstack([X.flatten(), Y.flatten(), np.ones_like(X.flatt
en())])
    template = It
    temp_grad_x = np.gradient(template, axis=1).flatten()
    temp_grad_y = np.gradient(template, axis=0).flatten()
    steepest_descent = np.vstack([temp_grad_x.flatten() * X.flatten(),
                                  temp_grad_x.flatten() * Y.flatten(),
                                  temp_grad_x.flatten(),
                                  temp_grad_y.flatten() * X.flatten(),
                                  temp_grad_y.flatten() * Y.flatten(),
                                  temp_grad_y.flatten()]).T
    inverse_hessian = np.linalg.inv(steepest_descent.T @ steepest_desc
ent)


    for _ in range(num_iters):
        new_coords = M @ coords
        mask = ((new_coords[0,:] >= 0) &
                (new_coords[0,:] < It.shape[1]) &
                (new_coords[1,:] >= 0) &
                (new_coords[1,:] < It.shape[0]))
        It1_warp = spline_It1.ev(new_coords[1], new_coords[0]).reshape
(It.shape)
        error = It1_warp - template
        dp = inverse_hessian @ steepest_descent.T @ error.flatten()
```

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



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

In [21]:
```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! =====
    It_spline = RectBivariateSpline(np.arange(It.shape[0]), np.arange(It.shape[1]), It)
    It1_spline = RectBivariateSpline(np.arange(It1.shape[0]), np.arange(It1.shape[1]), It1)
    x_grid, y_grid = np.meshgrid(np.arange(It.shape[1]), np.arange(It.shape[0]))
    old_coords = np.vstack([x_grid.flatten(), y_grid.flatten(), np.ones_like(x_grid.flatten())])
    template = It
    temp_grad_x = np.gradient(template, axis=1).flatten()
    temp_grad_y = np.gradient(template, axis=0).flatten()
    steepest_descent = np.vstack([temp_grad_x.flatten() * x_grid.flatten(),
                                  temp_grad_x.flatten() * y_grid.flatten(),
                                  temp_grad_x.flatten(),
                                  temp_grad_y.flatten() * x_grid.flatten(),
                                  temp_grad_y.flatten() * y_grid.flatten(),
                                  temp_grad_y.flatten()]).T
    inverse_hessian = np.linalg.inv(steepest_descent.T @ steepest_descent)

    for _ in range(num_iters):
        new_coords = M @ old_coords
        mask = ((new_coords[0,:] >= 0) &
                (new_coords[0,:] < It.shape[1]) &
                (new_coords[1,:] >= 0) &
                (new_coords[1,:] < It.shape[0]))
        It1_warp = It1_spline.ev(new_coords[1], new_coords[0]).reshape(It.shape)
        error = It1_warp - template
        dp = inverse_hessian @ steepest_descent.T @ error.flatten()
        if np.linalg.norm(dp) < threshold:
            break
        dM = np.array([[1 + dp[0], dp[1], dp[2]], [dp[3], 1 + dp[4], dp[5]], [0, 0, 1]])
        M = M @ np.linalg.inv(dM)
```

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

In [21]:

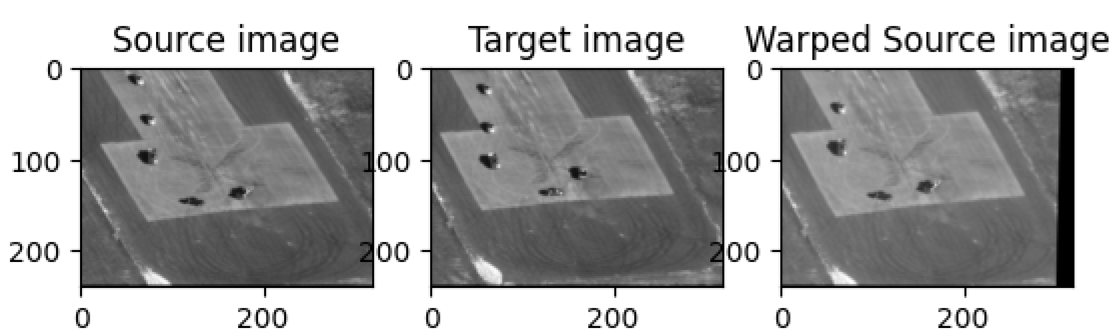
Debug Q4.1

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

In [22]:

```
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')
```

Out[22]: Text(0.5, 1.0, 'Warped Source image')



Q4.2 Tracking with Inverse Composition (10 points)

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

In [23]:
```
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 opt
imization
    :param threshold : (float), if the length of dp < threshold, termi
nate the optimization
    :param tolerance : (float), binary threshold of intensity differen
ce 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)
    imH, imW = It.shape

    It_warped = affine_transform(It, -M, offset=0.0, output_shape=Non
e, order=1)
    diff = np.absolute(It_warped - It)
    mask[diff > tolerance] = 0
    mask[diff < tolerance] = 1

    mask = binary_erosion(mask)
    mask = binary_dilation(mask, iterations=1)
    # ===== End of code =====

    return mask
```

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

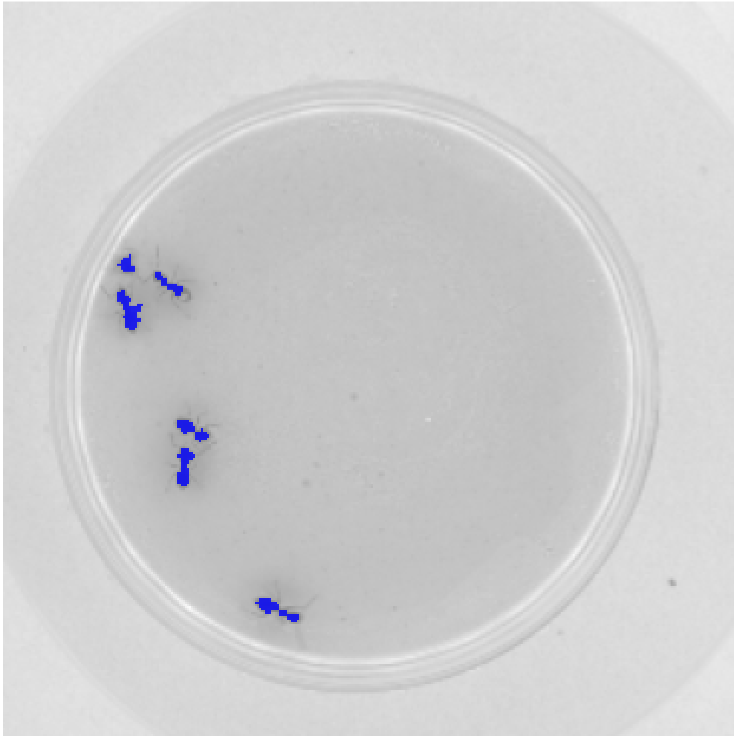
In [24]:
```
from tqdm import tqdm

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 optim
ization
    :param threshold : float, if the length of dp < threshold, termina
te the optimization
    :param tolerance : (float), binary threshold of intensity differen
ce when computing the mask
    :return: masks   : (T, 4) moved objects for each frame
    """
    H, W, N = seq.shape

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

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

        # Compute the mask for moving objects
        mask= SubtractDominantMotion(It, It1, num_iters, threshold, to
lerance)
        masks.append(mask)

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

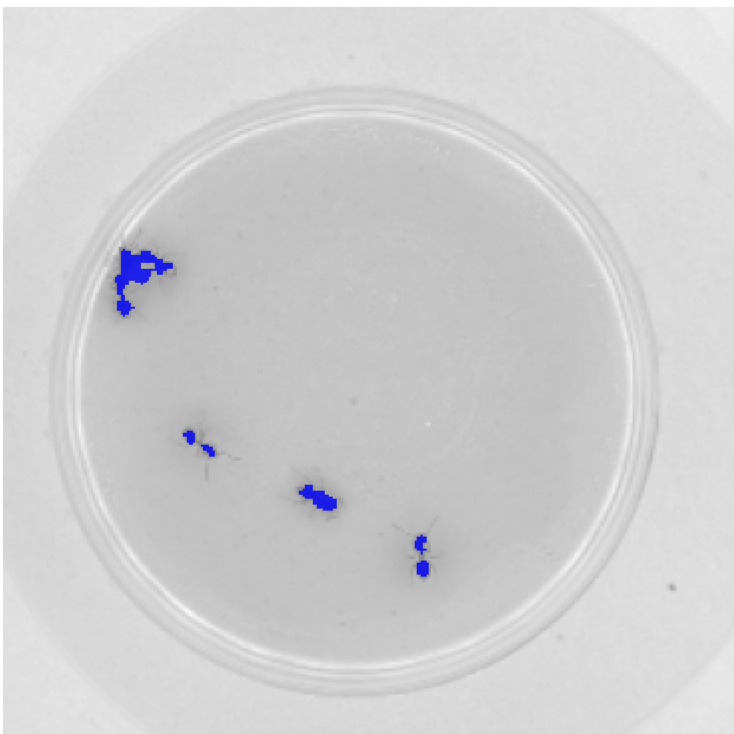
Track the ant sequence with inverse composition method.

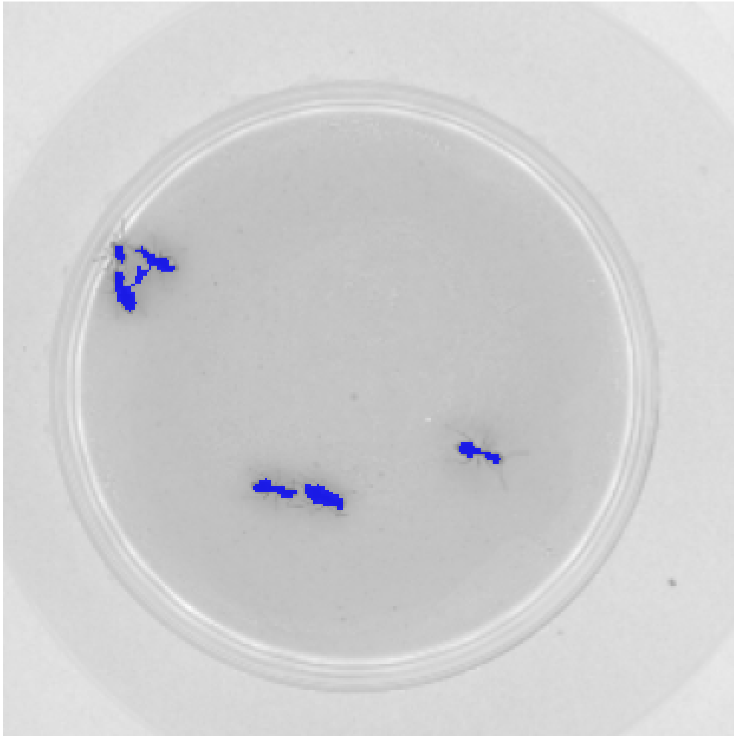
In [25]:
```
seq = np.load("/content/antseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 1000
threshold = 0.01
tolerance = 0.3

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

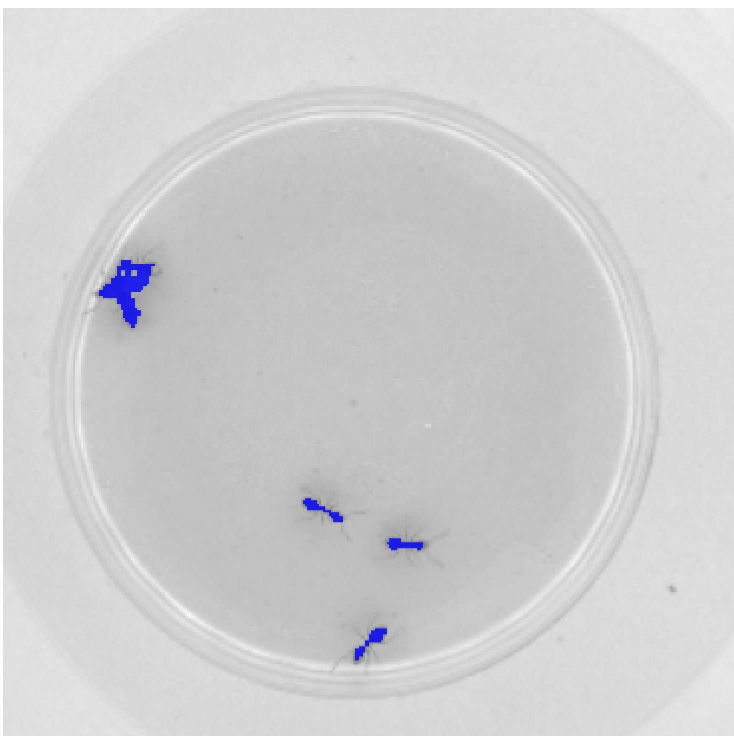
```
100%|██████████| 124/124 [00:29<00:00,  4.20it/s]

Ant Sequence takes 29.581000 seconds
```

In [26]:
```
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='winte
r', alpha=0.8)
    plt.axis('off')
```









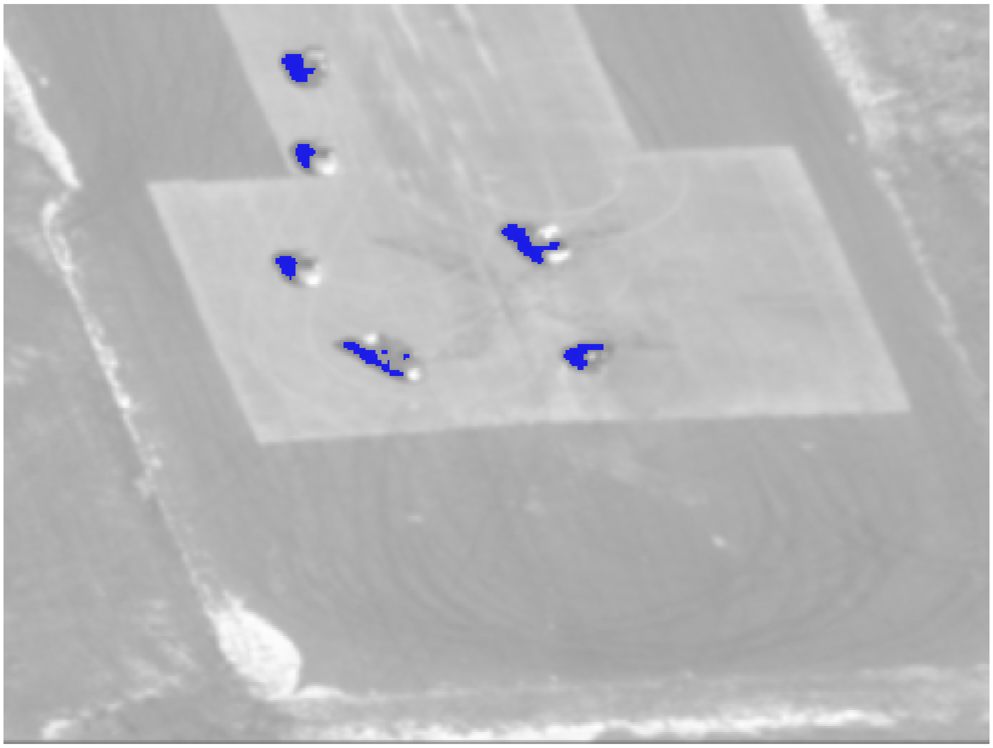
Track the aerial sequence with inverse composition method.

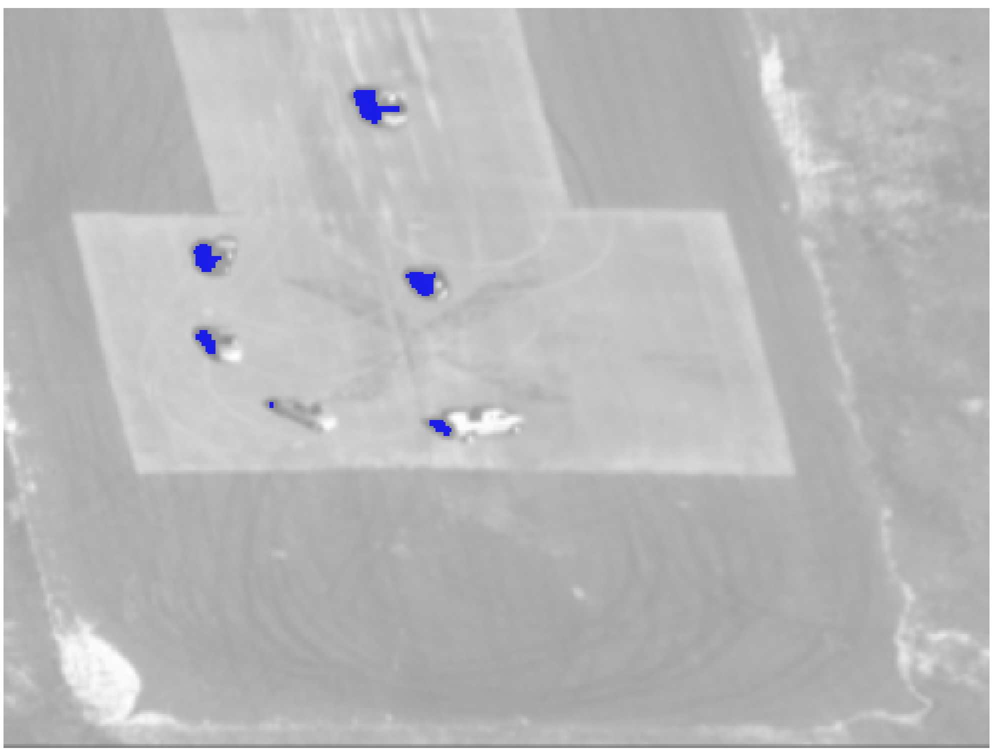
In [27]:

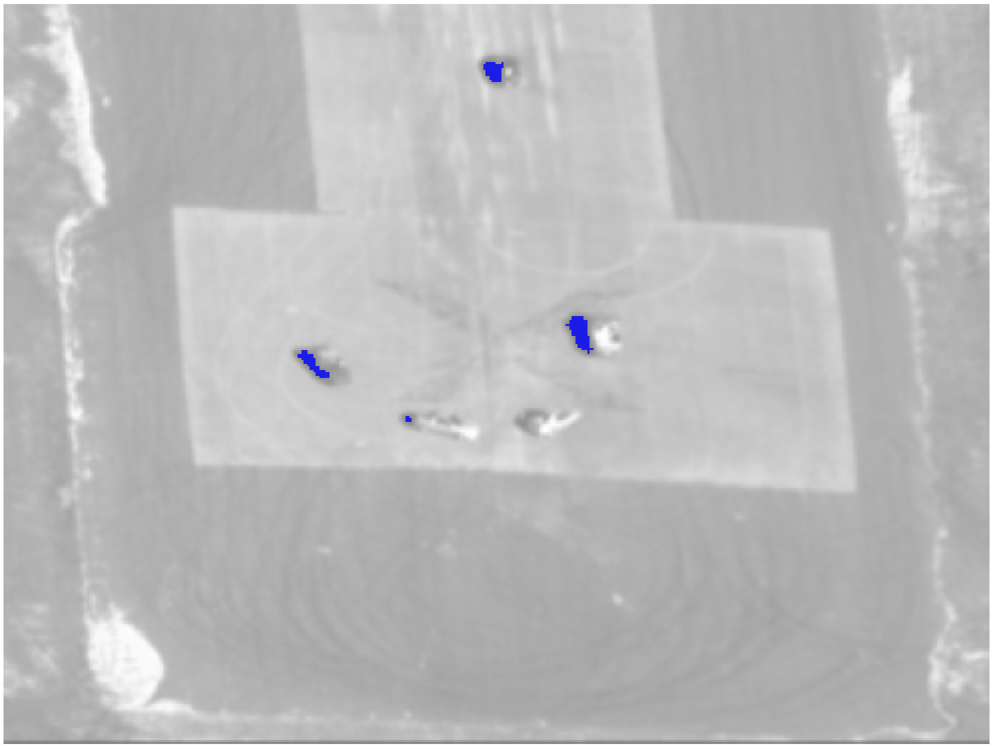
```
seq = np.load("/content/aerialseq.npy")

# NOTE: feel free to play with these parameters
num_iters = 1000
threshold = 0.01
tolerance = 0.3

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

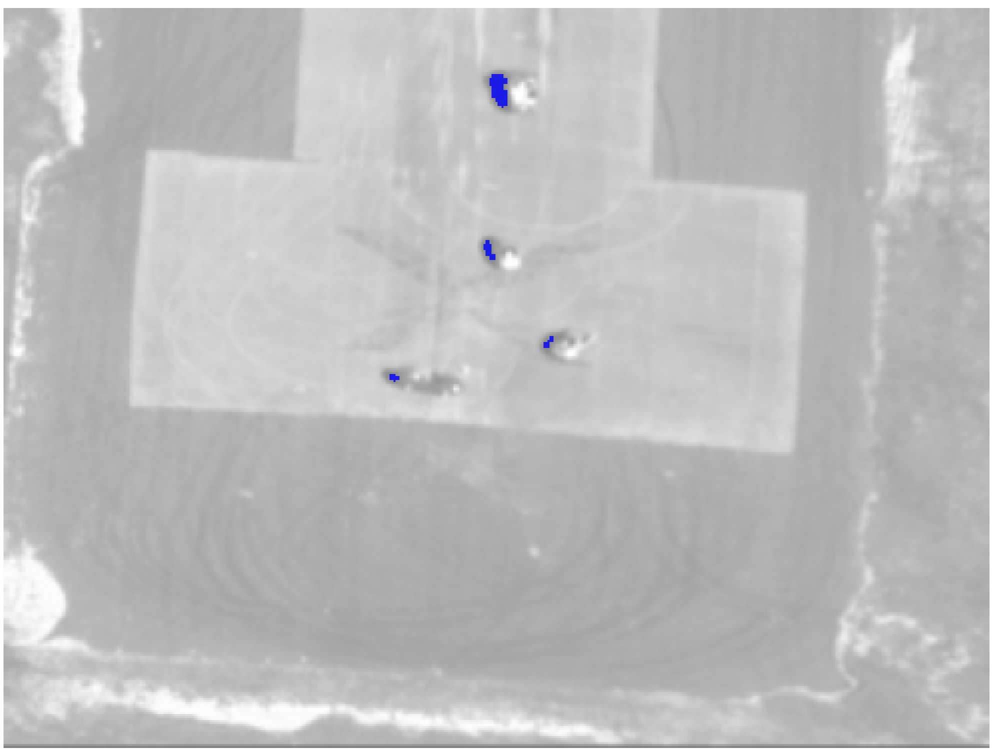
```
100%|██████████| 149/149 [01:10<00:00,  2.12it/s]

Ant Sequence takes 70.378632 seconds
```

In [17]:
```
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='winte
r', alpha=0.8)
    plt.axis('off')
```









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

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

Sequence	LK Algorithm	Inverse Composition Algorithm
ant	92.856568 s	29.014318 s
aerial	237.973585 s	72.448923 s

===== 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! =====

The inverse composition algorithm computes the hessian before the loop while the LK classic algorithm does the computation in the loop. By finding the gradients of the template and forming the hession without the warping the inverse method becomes siginicantly more efficient. ===== end of your answer ====

```
In [ ]:
```