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
```

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
```

```
--2025-09-24 02:23:55--  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  3.41MB/s    in 72s

2025-09-24 02:25:08 (3.37 MB/s) - ‘/content/carseq.npy’ saved [254976128/254976128]

--2025-09-24 02:25:08--  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  3.59MB/s    in 7.4s

2025-09-24 02:25:16 (3.56 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 $\frac{\partial \mathbf{W}(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T}$? (**Hint**: It should be a 2x2 matrix)

===== your answer here! =====
We know that W(x; p) is pure translational.
$W(x;p) = x + p$
W(x; p) =

$$\begin{bmatrix} x + p_x \\ y + p_y \end{bmatrix}$$

The derivate is a jacobian matrix defined as:

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

$$\begin{bmatrix} \frac{\partial (x+p_x)}{\partial p_x} & \frac{\partial (x+p_x)}{\partial p_y} \\ \frac{\partial (y+p_y)}{\partial p_x} & \frac{\partial (y+p_y)}{\partial p_y} \end{bmatrix}$$

=

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

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

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

===== your answer here! =====
A is a (D, 2) matrix.

A = $\frac{\partial I_{t+1}(x')}{\partial x'^T} \frac{\partial W(x;p)}{\partial p^T}$

b is a (D, 1) vector.

b = $I_t(x) - I_{t+1}(x')$
b = $I_t(x) - I_{t+1}(W(x;p))$

===== 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! =====
$\mathbf{A}^T\mathbf{A}$ must be invertible which means the image gradients should have sufficient variation in both x and y directions.

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

Q2.2: Lucas-Kanade (20 points)

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

```
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.RectBiv
    # https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html

    height, width = It.shape
    x1, y1, x2, y2 = rect[0], rect[1], rect[2], rect[3]
    top_left = [x1, y1]
    top_right = [x2, y1]
    bottom_left = [x1, y2]
    bottom_right = [x2, y2]

    spline_t = RectBivariateSpline(np.arange(height), np.arange(width), It)
    spline_t1 = RectBivariateSpline(np.arange(height), np.arange(width), It1)
```

```
    rows = np.arange(x1, x2 + 1)
    col = np.arange(y1, y2 + 1)
    x, y = np.meshgrid(rows, col)
    template_patch = spline_t.ev(y, x)

    for i in range(num_iters):
      wx = x + p[0]
      wy = y + p[1]

      if (wx.min() < 0 or wy.min() < 0 or wx.max() >= width or wy.max() >= height):
          break

      imaget1_warp = spline_t1.ev(wy, wx)

      error = (template_patch - imaget1_warp)
      error = error.flatten()
      gradient_x = spline_t1.ev(wy, wx, dx = 0, dy = 1).flatten()
      gradient_y = spline_t1.ev(wy, wx, dx = 1, dy = 0).flatten()

      A = np.vstack((gradient_x, gradient_y)).T
      delta_p, residuals, rank, s = lstsq(A, error, rcond=None)

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


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

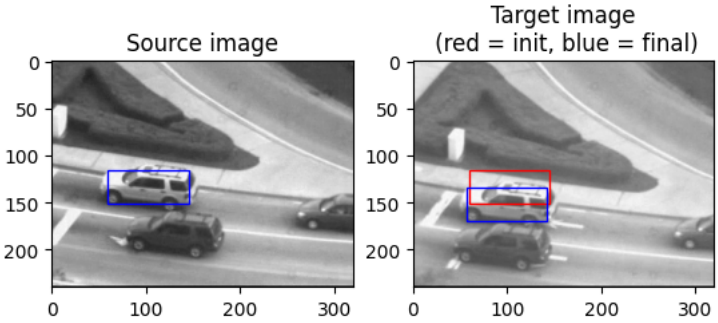
Debug Q2.2

A few tips to debug your implementation:

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

```
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 = 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')
```



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

```
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]

    initial_frame = seq[:, :, 0]
    initial_rect = rect.copy()
    epsilon = 1.5

    rects.append(rect)
    p0 = np.zeros(2)

    # Iterate over the car sequence and track the car
    for i in range(1, seq.shape[2]):

        # ===== your code here! =====
        # TODO: add your code track the object of interest in the sequence

        It1 = seq[:, :, i]
        p = LucasKanade(It, It1, rect, threshold, num_iters, p0).flatten()

        p_star = LucasKanade(initial_frame, It1, initial_rect, threshold, num_iters, p0).flatten()

        if np.linalg.norm(p_star - p) <= epsilon:
            rect = rect + np.array([p_star[0], p_star[1], p_star[0], p_star[1]])
        else:
            rect = rect + np.array([p[0], p[1], p[0], p[1]])
            p0 = p

        # rect = rect + np.array([p[0], p[1], p[0], p[1]])
        rects.append(rect.copy())
        It = It1



        # ===== 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.

```
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');
```

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

# NOTE: feel free to play with these parameters
# TUNED PARAMETERS
num_iters = 10000
threshold = 0.02

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.

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

# NOTE: feel free to play with these parameters
# TUNED PARAMETERS
num_iters = 10000
threshold = 0.02

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

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

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

```
--2025-09-26 22:00:54--  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: ‘aerialseq.npy’

aerialseq.npy       100%[===================>]  87.89M   554KB/s    in 2m 49s

2025-09-26 22:03:45 (532 KB/s) - ‘aerialseq.npy’ saved [92160128/92160128]

--2025-09-26 22:03:45--  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: ‘antseq.npy’

antseq.npy          100%[===================>]  62.50M   574KB/s    in 1m 55s

2025-09-26 22:05:41 (556 KB/s) - ‘antseq.npy’ saved [65536128/65536128]
```

Q3: Affine Motion Subtraction

Q3.1: Dominant Motion Estimation (15 points)

```
from scipy.interpolate import RectBivariateSpline
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! =====
    height, width = It.shape
    p = np.zeros(6)
```

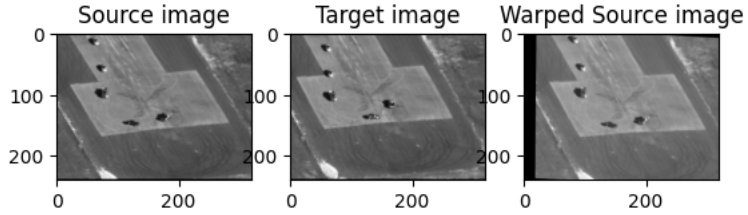
```
    rows = np.arange(height)
    columns = np.arange(width)

    x, y = np.meshgrid(rows, columns, indexing='ij')
    x = x.flatten()
    y = y.flatten()
    It_flat = It.flatten()

    for i in range(num_iters):
        It1_warp = affine_transform(It1, M, mode='nearest')
        error = It_flat - It1_warp.flatten()

        Ix, Iy = np.gradient(It1_warp)
        Ix = Ix.flatten()
        Iy = Iy.flatten()

        A = np.vstack((Ix * x, Iy * x, Ix * y, Iy * y, Ix, Iy)).T

        hessian = A.T @ A
        delta_p = np.linalg.inv(hessian) @ A.T @ error

        p += delta_p
        M = np.array([[1.0 + p[0], p[2], p[4]], [p[1], 1.0 + p[3], p[5]]])

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

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

```
import cv2

num_iters = 100
threshold = 0.02
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')
```

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



```
Start coding or generate with AI.
```

Q3.2: Moving Object Detection (10 points)

```
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! =====
    H, W = It.shape

    M = LucasKanadeAffine(It, It1, threshold, num_iters)
    # warped_img = affine_transform(It1, M)
    warped_It1 = cv2.warpAffine(It1, M, (W, H), flags=cv2.INTER_LINEAR, borderMode=cv2.BORDER_CONSTANT, borderValue=0)
    valid_mask = cv2.warpAffine(np.ones_like(It1), M, (W, H), flags=cv2.INTER_NEAREST, borderMode=cv2.BORDER_CONSTANT, borderValue
    difference = np.abs(It - warped_It1) * valid_mask

    mask = (difference > tolerance)
    mask = binary_dilation(mask, iterations=2, brute_force=True).astype(mask.dtype)
    mask = binary_erosion(mask, iterations=2, brute_force=True).astype(mask.dtype)

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

    return mask.astype(bool)
```

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

```
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])):
        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
```

Ant Sequence

Q3.3 (a) - Track Ant Sequence

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

# NOTE: feel free to play with these parameters
#TUNED PARAMETERS - ITERATION FOR EROSION AND DILATION = 1
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 [00:13<00:00,  9.48it/s]

Ant Sequence takes 13.120779 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')
```

```
Start coding or generate with AI.
```

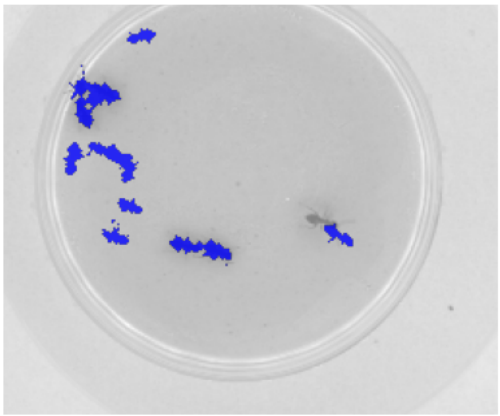
Aerial Sequence

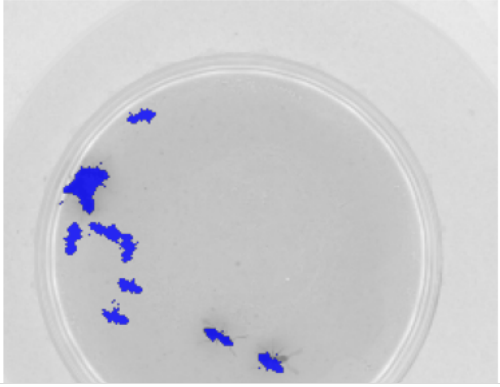
Q3.3 (b) - Track Aerial Sequence

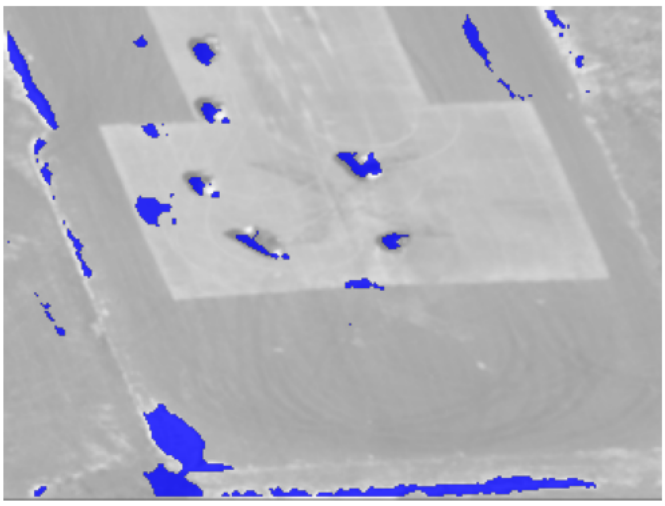
```
seq = np.load("aerialseq.npy")

# NOTE: feel free to play with these parameters
#TUNED PARAMETERS - ITERATION FOR EROSION AND DILATION = 2
num_iters = 1000
threshold = 0.01
tolerance = 0.3

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

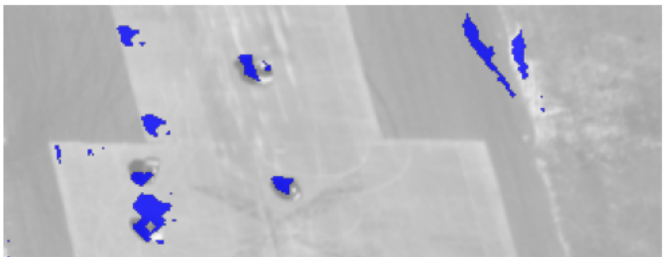
```
100%|██████████| 149/149 [30:13<00:00, 12.17s/it]
Ariel Sequence takes 1813.514410 seconds
```

```
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
```

```
--2025-09-26 16:17:00--  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  9.51MB/s    in 8.2s

2025-09-26 16:17:08 (10.7 MB/s) - ‘/content/aerialseq.npy’ saved [92160128/92160128]

--2025-09-26 16:17:08--  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  12.0MB/s    in 5.1s

2025-09-26 16:17:14 (12.3 MB/s) - ‘/content/antseq.npy’ saved [65536128/65536128]
```

Q4: Efficient Tracking

Q4.1: Inverse Composition (15 points)

```
from scipy.interpolate import RectBivariateSpline
import cv2

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, y = np.meshgrid(np.arange(W), np.arange(H))
```

```
    x, y = x.flatten(), y.flatten()
    ones = np.ones_like(x)
    coords = np.vstack([x, y, ones])

    spline_It = RectBivariateSpline(np.arange(H), np.arange(W), It)
    spline_It1 = RectBivariateSpline(np.arange(H), np.arange(W), It1)
    Ix = spline_It.ev(y, x, dx=0, dy=1).flatten()
    Iy = spline_It.ev(y, x, dx=1, dy=0).flatten()

    A = np.vstack([
        Ix * x, Ix * y, Ix,
        Iy * x, Iy * y, Iy
    ]).T

    H_inv = np.linalg.inv(A.T @ A)

    It_flat = It.flatten()

    for i in range(num_iters):
        It1_warped = cv2.warpAffine(It1, M, (W, H), flags=cv2.INTER_LINEAR, borderMode=cv2.BORDER_REPLICATE)
        It1_warped_flat = It1_warped.flatten()

        warp_coords = M @ coords
        x_warp, y_warp = warp_coords[0, :], warp_coords[1, :]

        out_of_bounds = (x_warp < 0) | (x_warp >= W) | (y_warp < 0) | (y_warp >= H)
        It_warped = spline_It.ev(y_warp, x_warp)
        It1_warped = spline_It1.ev(y_warp, x_warp)
        error = (It_flat - It1_warped_flat).reshape(-1, 1)
        error[out_of_bounds] = 0

        dp = H_inv @ (A.T @ error)
        dp = dp.flatten()

        delta_M = np.array([
            [1 + dp[0], dp[1], dp[2]],
            [dp[3], 1 + dp[4], dp[5]],
            [0, 0, 1]
        ])

        M_full = np.vstack([M, [0, 0, 1]])
        M = (M_full @ np.linalg.inv(delta_M))[:2, :]

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


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

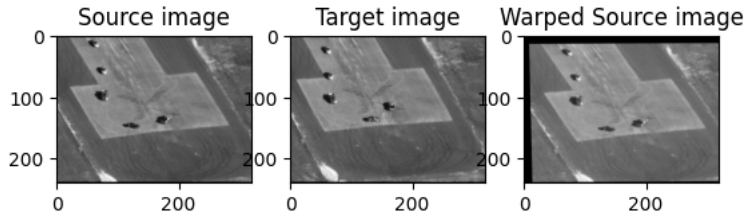
Debug Q4.1

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.

```
import cv2

num_iters = 100
threshold = 0.02
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')
```

```
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.

```
import numpy as np
from scipy.ndimage import binary_erosion, binary_dilation
from scipy.ndimage import affine_transform
import cv2
def SubtractDominantMotion(It, It1, num_iters, threshold, tolerance):
    """
    Returns a mask of moving objects by subtracting dominant global motion.
    """
    H, W = It.shape

    M = InverseCompositionAffine(It, It1, threshold, num_iters)

    # warped_It1 = affine_transform(It1, M, mode='nearest')
    # valid_mask = affine_transform(np.ones_like(It1), M, mode='nearest')
    warped_It1 = cv2.warpAffine(It1, M, (W, H), flags=cv2.INTER_LINEAR, borderMode=cv2.BORDER_CONSTANT, borderValue=0)
    valid_mask = cv2.warpAffine(np.ones_like(It1), M, (W, H), flags=cv2.INTER_NEAREST, borderMode=cv2.BORDER_CONSTANT, borderValue

    diff = np.abs(It - warped_It1) * valid_mask
    mask = diff > tolerance

    mask = binary_dilation(mask, iterations=2)
    mask = binary_erosion(mask, iterations=2)

    return mask
```

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

```
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, seq.shape[2])):
        It1 = seq[:, :, i]
        mask = SubtractDominantMotion(It, It1, num_iters, threshold, tolerance)
        masks.append(mask)


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

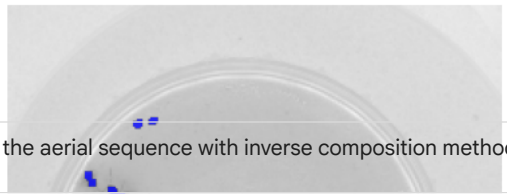
```
    masks = np.stack(masks, axis=2)
    return masks
```

Track the ant sequence with inverse composition method.

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

# NOTE: feel free to play with these parameters
#dilation and erosion iteration = 1
num_iters = 1000
threshold = 0.02
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 [00:12<00:00, 10.29it/s]

Ant Sequence takes 12.105574 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')
```

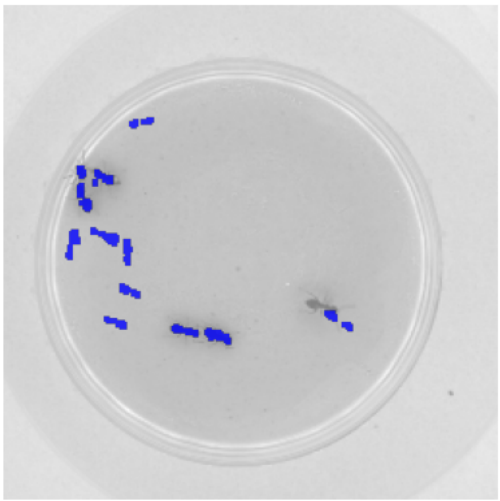


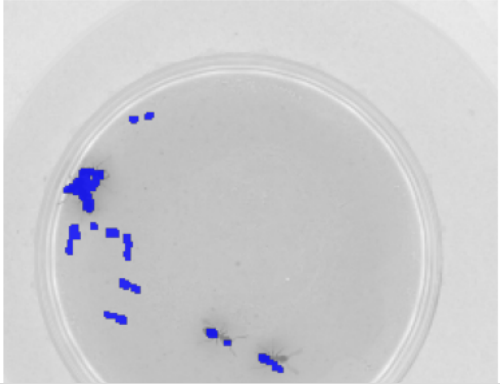
Track the aerial sequence with inverse composition method.

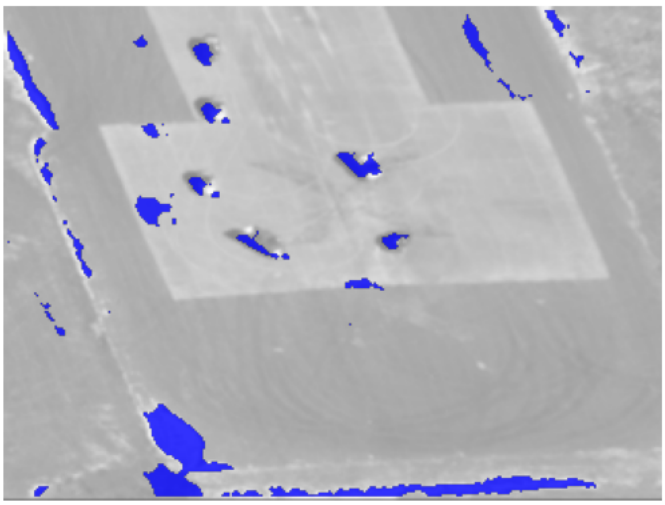
```
seq = np.load("/content/aerialseq.npy")

# NOTE: feel free to play with these parameters
#dilation and erosion iteration = 2
num_iters = 1000
threshold = 1.0
tolerance = 0.3

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

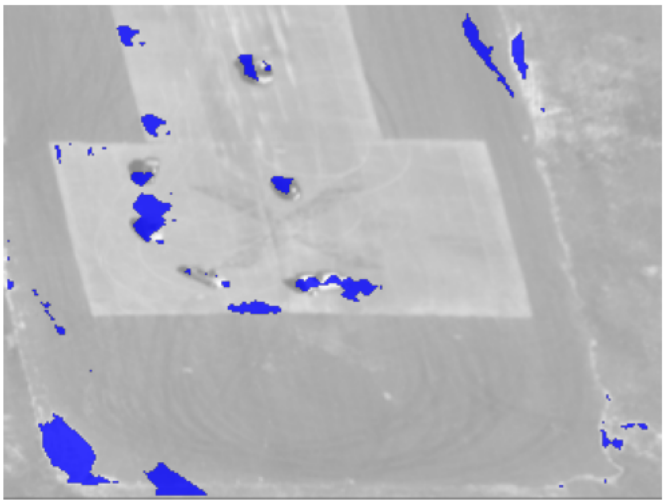
```
100%|██████████| 149/149 [00:43<00:00,  3.42it/s]
Aerial Sequence takes 43.708784 seconds
```

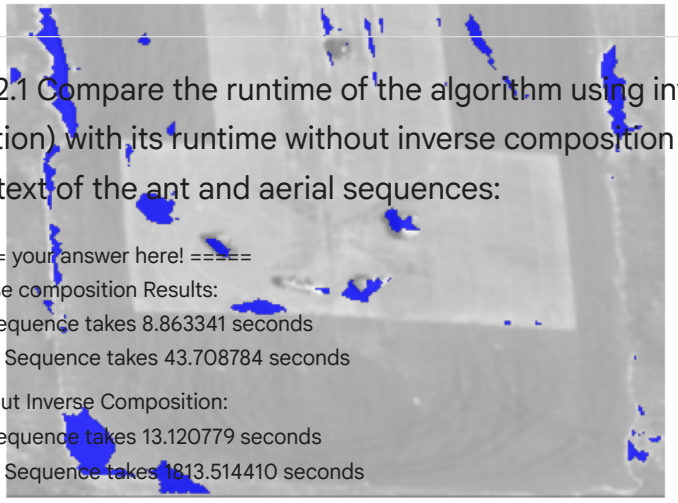
```
frames_to_save = [29, 59, 89, 119]

# 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')
```











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! =====
Inverse composition Results:
Ant Sequence takes 8.863341 seconds
Aerial Sequence takes 43.708784 seconds
Without Inverse Composition:
Ant Sequence takes 13.120779 seconds
Aerial Sequence takes 1813.514410 seconds
===== end of your answer ====
```

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
```

Download data

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

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

if not os.path.exists('/content/hammer_handle_data.npz'):
    !wget https://www.cs.cmu.edu/~deva/data/hammer_handle_data.npz -O /content/hammer_handle_data.npz
```

```
--2025-09-26 19:17:18--  https://www.cs.cmu.edu/~deva/data/bead_data.npz
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: 138394388 (132M)
Saving to: ‘/content/bead_data.npz’

/content/bead_data. 100%[===================>] 131.98M  1.58MB/s    in 77s

2025-09-26 19:18:36 (1.72 MB/s) - ‘/content/bead_data.npz’ saved [138394388/138394388]

--2025-09-26 19:18:37--  https://www.cs.cmu.edu/~deva/data/hammer_handle_data.npz
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: 92698388 (88M)
Saving to: ‘/content/hammer_handle_data.npz’

/content/hammer_han 100%[===================>]  88.40M  1.95MB/s    in 46s

2025-09-26 19:19:23 (1.92 MB/s) - ‘/content/hammer_handle_data.npz’ saved [92698388/92698388]
```

We provide the following helper function, which computes the rotation matrix from the parameterization θ and its derivative with respect to rotation component of θ.

```
"""
Helper Functions
"""

def get_dR_dtheta_xyz(theta_x, theta_y, theta_z):
    """
    Compute parital derivatives of the rotation matrix R = Rz(theta_z) @ Ry(theta_y) @ Rx(theta_x) wrt theta_x, theta_y, theta_z.

    :param[float] theta_x              : x element of x-y-z Euler Rotation.
    :param[float] theta_y              : y element of x-y-z Euler Rotation.
    :param[float] theta_z              : z element of x-y-z Euler Rotation.
    :return
        [np.array (3, 3)] dR_dtheta_x   : The partial derivative of the rotation matrix with regards to theta_x.
        [np.array (3, 3)] dR_dtheta_y   : The partial derivative of the rotation matrix with regards to theta_y.
        [np.array (3, 3)] dR_dtheta_z   : The partial derivative of the rotation matrix with regards to theta_z.
    """
    cx, cy, cz = np.cos(theta_x), np.cos(theta_y), np.cos(theta_z)
    sx, sy, sz = np.sin(theta_x), np.sin(theta_y), np.sin(theta_z)

    # Rotation matrices
    Rx = np.array([
        [1, 0, 0],
        [0, cx, -sx],
        [0, sx, cx]
```

```
        ])
        Ry = np.array([
            [cy, 0, sy],
            [0, 1, 0],
            [-sy, 0, cy]
        ])
        Rz = np.array([
            [cz, -sz, 0],
            [sz,  cz, 0],
            [0,   0,  1]
        ])

        # Derivatives of Rx wrt x
        dRx_dx = np.array([
            [0, 0, 0],
            [0, -sx, -cx],
            [0, cx, -sx]
        ])
        # Derivatives of Ry wrt y
        dRy_dy = np.array([
            [-sy, 0, cy],
            [0, 0, 0],
            [-cy, 0, -sy]
        ])
        # Derivatives of Rz wrt z
        dRz_dz = np.array([
            [-sz, -cz, 0],
            [cz, -sz, 0],
            [0, 0, 0]
        ])

        # Chain rule for full R = Rz * Ry * Rx
        dR_dx = Rz @ Ry @ dRx_dx
        dR_dy = Rz @ dRy_dy @ Rx
        dR_dz = dRz_dz @ Ry @ Rx

        return dR_dx, dR_dy, dR_dz

    def get_R(theta_x, theta_y, theta_z):
        """
        Compute the rotation matrix R = Rz(theta_z) @ Ry(theta_y) @ Rx(theta_x).

        :param[float] theta_x        : x element of x-y-z Euler Rotation.
        :param[float] theta_y        : y element of x-y-z Euler Rotation.
        :param[float] theta_z        : z element of x-y-z Euler Rotation.
        :return[np.array (3, 3)] R   : The rotation matrix.
        """
        cx, cy, cz = np.cos(theta_x), np.cos(theta_y), np.cos(theta_z)
        sx, sy, sz = np.sin(theta_x), np.sin(theta_y), np.sin(theta_z)

        # Rotation matrices
        Rx = np.array([
            [1, 0, 0],
            [0, cx, -sx],
            [0, sx, cx]
        ])
        Ry = np.array([
            [cy, 0, sy],
            [0, 1, 0],
            [-sy, 0, cy]
        ])
        Rz = np.array([
            [cz, -sz, 0],
            [sz,  cz, 0],
            [0,   0,  1]
        ])
        R = Rz @ Ry @ Rx
        return R
```

Q5.2: Implement NormalFlow (15 points)

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

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

def rbs(img):
    H, W = img.shape
    return RectBivariateSpline(np.arange(H), np.arange(W), img, kx=1, ky=1)

def eval3(rbs_list, y, x):
    return np.stack([r.ev(y, x) for r in rbs_list], axis=1)

def gradients(img3):
    H, W, C = img3.shape
    Ix = np.zeros_like(img3, dtype=np.float64)
    Iy = np.zeros_like(img3, dtype=np.float64)
    for c in range(C):
        Iy[..., c], Ix[..., c] = np.gradient(img3[..., c])
    return Ix, Iy

def inside_bounds(x, y, W, H):
    return (x >= 0) & (x <= W - 1) & (y >= 0) & (y <= H - 1)

def NormalFlow(It, It1, Ht, Ht1, Ct, Ct1, max_iters=20, rot_threshold=0.0001, trans_threshold=0.1, init_theta=np.zeros(5)):
    """
    :param[np.array(H, W, 3)] It   : Normal Map at time t [float]
    :param[np.array(H, W, 3)] It1  : Normal Map at time t+1 [float]
    :param[np.array(H, W)] Ht      : Height Map at time t, unit: pixel [float]
    :param[np.array(H, W)] Ht1     : Height Map at time t+1, unit: pixel [float]
    :param[np.array(H, W)] Ct      : Contact Mask at time t [bool]
    :param[np.array(H, W)] Ct1     : Contact Mask at time t+1 [bool]
    :param[int] max_iters          : Max number of optimization iterations
    :param[float] rot_threshold    : If change in rotation parameters is less than thresh, terminate the optimization
    :param[float] trans_threshold  : If change in translation parameters is less than thresh, terminate the optimization
    :param[np.array(5)] init_theta : The initial parameter
    :return[np.array(4, 4)] t1_T_t : The Homogeneous transformation matrix of the object from frame t to frame t+1
    """
    # Initialize theta to zeros
    theta = init_theta.copy()

    # ===== your code here! =====
    # Hint: Iterate over max_iters and for each iteration, construct a linear system (Ax=b) that solves for a x=delta_theta update
    # Construct [A] by computing image gradients at (possibly fractional) pixel locations and for each channel
    # It might be easier to consider compute the gradient over each element of theta independently
    # For each iteration, we suggest to first compute the warped pixel, then compute the shared contact mask, finally construct the
    # 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_theta], add it to [theta] (and move on to next iteration)
    # Use the following termination condition:
    #
    # if (np.linalg.norm(delta_theta[:3]) < rot_threshold and np.linalg.norm(delta_theta[3:]) < trans_threshold):
    #     break
    #
    # 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.RectBiva
    # https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html

    It   = It.astype(np.float64)
    It1  = It1.astype(np.float64)
    Ht   = Ht.astype(np.float64)
    Ct   = Ct.astype(bool)
    Ct1  = Ct1.astype(bool)

    H, W, _ = It.shape
    yy, xx = np.meshgrid(np.arange(H), np.arange(W), indexing='ij')

    rbs_I1  = [rbs(It1[..., c]) for c in range(3)]
    Ix1, Iy1 = gradients(It1)
    rbs_Ix1 = [rbs(Ix1[..., c]) for c in range(3)]
    rbs_Iy1 = [rbs(Iy1[..., c]) for c in range(3)]
    rbs_Ct1 = rbs(Ct1.astype(np.float64))

    mask_ref = Ct
    y_ref = yy[mask_ref].ravel()
    x_ref = xx[mask_ref].ravel()
    N = x_ref.size

    if N == 0:
        return np.asarray(init_theta, dtype=np.float64).reshape(5,)
```

```
n_ref = It[mask_ref].reshape(N, 3)
h_ref = Ht[mask_ref].reshape(N)
x3_ref = np.stack([x_ref, y_ref, h_ref], axis=1)

def build_rotation_and_partials(th):
    thx, thy, thz = float(th[0]), float(th[1]), float(th[2])
    R = get_R(thx, thy, thz)
    dRx, dRy, dRz = get_dR_dtheta_xyz(thx, thy, thz)
    return R, dRx, dRy, dRz

for i in range(max_iters):
    R, dRx, dRy, dRz = build_rotation_and_partials(theta)

    RX = (x3_ref @ R.T)
    xw = RX[:, 0] + theta[3]
    yw = RX[:, 1] + theta[4]

    inb = inside_bounds(xw, yw, W, H)
    if not np.any(inb):
        break

    xw_in = xw[inb]
    yw_in = yw[inb]
    X3_in = x3_ref[inb, :]
    n_ref_in = n_ref[inb, :]

    Ct1_vals = rbs_Ct1.ev(yw_in, xw_in)
    shared = Ct1_vals > 0.5
    if not np.any(shared):
        break

    xw_c = xw_in[shared]
    yw_c = yw_in[shared]
    X3_c = X3_in[shared, :]
    n_ref_c = n_ref_in[shared, :]
    M = xw_c.size

    n_warp = eval3(rbs_I1,  yw_c, xw_c)
    Ix_s   = eval3(rbs_Ix1, yw_c, xw_c)
    Iy_s   = eval3(rbs_Iy1, yw_c, xw_c)

    Rn_ref = (R @ n_ref_c.T).T
    r = (n_warp - Rn_ref).reshape(-1)

    A = np.zeros((3*M, 5), dtype=np.float64)

    dRX_x = (X3_c @ dRx.T)
    dRX_y = (X3_c @ dRy.T)
    dRX_z = (X3_c @ dRz.T)

    dW_dthx = np.stack([dRX_x[:, 0], dRX_x[:, 1]], axis=1)
    dW_dthy = np.stack([dRX_y[:, 0], dRX_y[:, 1]], axis=1)
    dW_dthz = np.stack([dRX_z[:, 0], dRX_z[:, 1]], axis=1)
    dW_dtx  = np.tile(np.array([1.0, 0.0]), (M,1))
    dW_dty  = np.tile(np.array([0.0, 1.0]), (M,1))

    dR_n_thx = (dRx @ n_ref_c.T).T  # (M,3)
    dR_n_thy = (dRy @ n_ref_c.T).T
    dR_n_thz = (dRz @ n_ref_c.T).T

    row = 0
    for c in range(3):
        grad_c = np.stack([Ix_s[:, c], Iy_s[:, c]], axis=1)  # (M,2)
        A[row:row+M, 0] = np.sum(grad_c * dW_dthx, axis=1) - dR_n_thx[:, c]
        A[row:row+M, 1] = np.sum(grad_c * dW_dthy, axis=1) - dR_n_thy[:, c]
        A[row:row+M, 2] = np.sum(grad_c * dW_dthz, axis=1) - dR_n_thz[:, c]
        A[row:row+M, 3] = np.sum(grad_c * dW_dtx,  axis=1)
        A[row:row+M, 4] = np.sum(grad_c * dW_dty,  axis=1)
        row += M

    delta_theta, *_ = lstsq(A, -r, rcond=None)
    theta += delta_theta.reshape(5,)

    if (np.linalg.norm(delta_theta[:3]) < rot_threshold and
        np.linalg.norm(delta_theta[3:]) < trans_threshold):
        break
```

```
    theta = np.asarray(theta, dtype=np.float64).reshape(5,)

    # ===== End of code =====
    return theta
```

Debug Q5.2

Debugging NormalFlow can be challenging. We provide the ground-truth object pose (in θ) between frame 90 and frame 0 in the code below. You can use this to verify your implementation. Below are some tips for debugging:

- **Warping function**: Plot $I_{t+1}(\mathbf{W}(x;\theta))$ and $\mathbf{R}_\theta I_t(x)$, masked with the shared mask $\bar{C}$. If your implementation is correct, the two should match. A helper function below is provided to convert a normal map into an RGB image.
- **Gradient of the warped image**: Once you have a correct implementation of $I_{t+1}(\mathbf{W}(x;\theta))$, compute its numerical derivative $\big(I_{t+1}(\mathbf{W}(x;\theta+d\theta)) - I_{t+1}(\mathbf{W}(x;\theta))\big)/d\theta$ for each dimension of θ. Plot the results and check if they match your implementation of $\nabla I_{t+1}(\mathbf{W}(x;\theta))$.
- **Gradient of the rotated image**: Use the same procedure to debug your implementation of $\nabla \mathbf{R}_\theta I_t(x)$.
- **Convergence with provided initialization**: If you use the provided θ as `init_theta`, NormalFlow should converge very quickly to a value very close to the ground-truth θ.
- **Convergence plot**: Plot the iterations versus the norm of `delta_theta`. Your algorithm should show fast convergence.

```
"""
Helper Functions
"""

def normal2image(I):
    """
    Given a normal map, turn it into an image and return that

    :param[np.array (H, W, 3)] I        : The Normal Map
    :return[np.array (H, W, 3)] image   : The image
    """
    image = (np.clip((I + 1.0) / 2.0, 0.0, 1.0) * 255).astype(np.uint8)
    return image
```

```
# load the data
data_path = "/content/bead_data.npz"
data = np.load(data_path, allow_pickle=True)
Cs = data["contact_masks"]
Is = data["normal_maps"]
Hs = data["height_maps"]

# We offer you the true object pose (in theta) of frame 90 relevant to frame 0
It = Is[0]; Ct = Cs[0]; Ht = Hs[0]
It1 = Is[90]; Ct1 = Cs[90]; Ht1 = Hs[90]
true_theta = [-0.00875, -0.00544, 0.15949, 43.419, -49.342]

# Plot the images
plt.imshow(normal2image(It))
plt.axis('off')
plt.show()
plt.imshow(normal2image(It1))
plt.axis('off')
plt.show()
```

```
Start coding or generate with AI.
```

Q5.3: Tracking with NormalFlow (10 points)

Implement the tracking method that, given a sequence of tactile-derived local geometries, returns the object pose of each frame relative to the first frame in the form of a 4×4 homogeneous transformation matrix. A helper function is provided to convert θ into a homogeneous matrix. For convenience, this helper assumes zero translation along the z-axis, since the object remains in contact with the sensor.

```
"""
Helper Functions
"""

def theta2T(theta):
    """
    Given the theta, assume zero z-translation, compute homogeneous transform matrix.

    :param[np.array (5)] theta        : The parameterization of the object transformation.
    :return[np.array (4, 4)] T        : The homogeneous transformation matrix.
    """
    T = np.eye(4, dtype=np.float32)
    T[:3, :3] = get_R(theta[0], theta[1], theta[2])
    T[0, 3] = theta[3]
    T[1, 3] = theta[4]
    return T
```

```
def TrackSequence(Is, Hs, Cs):
    """
    :param[np.array(N, H, W, 3)] Is   : Normal Map sequence [float]
```

```
    :param[np.array(N, H, W)] Hs      : Height Map sequence, unit: pixel [float]
    :param[np.array(N, H, W)] Cs      : Contact Mask sequence [bool]
    :param[int] max_iters             : Max number of optimization iterations
    :param[float] rot_threshold       : If change in rotation parameters is less than thresh, terminate the optimization
    :param[float] trans_threshold     : If change in translation parameters is less than thresh, terminate the optimization
    :return:[np.array(N, 4, 4)] Ts    : object pose with regards to the first frame in the form of homogeneous transformation
    """
    init_T = np.eye(4, dtype=np.float32)
    Ts = [init_T.copy()]

    max_iters = 20
    rot_threshold = 1e-4
    trans_threshold = 0.1

    # Iterate over the car sequence and track the car
    for i in range(Is.shape[0] - 1):

        # ===== your code here! =====
        # TODO: add your code track the 6DoF pose of the object
        It, It1 = Is[i], Is[i + 1]
        Ht, Ht1 = Hs[i], Hs[i + 1]
        Ct, Ct1 = Cs[i], Cs[i + 1]

        theta = NormalFlow(It, It1, Ht, Ht1, Ct, Ct1, max_iters=max_iters, rot_threshold=rot_threshold, trans_threshold=trans_thre
        t1_T_t = theta2T(theta)

        global_T = t1_T_t @ Ts[-1]
        Ts.append(global_T)

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

    Ts = np.stack(Ts, 0)
    assert Ts.shape == (Is.shape[0], 4, 4), f"Your output sequence {Ts.shape} is not ({Is.shape[0]}x{4}x{4})"
    return Ts
```

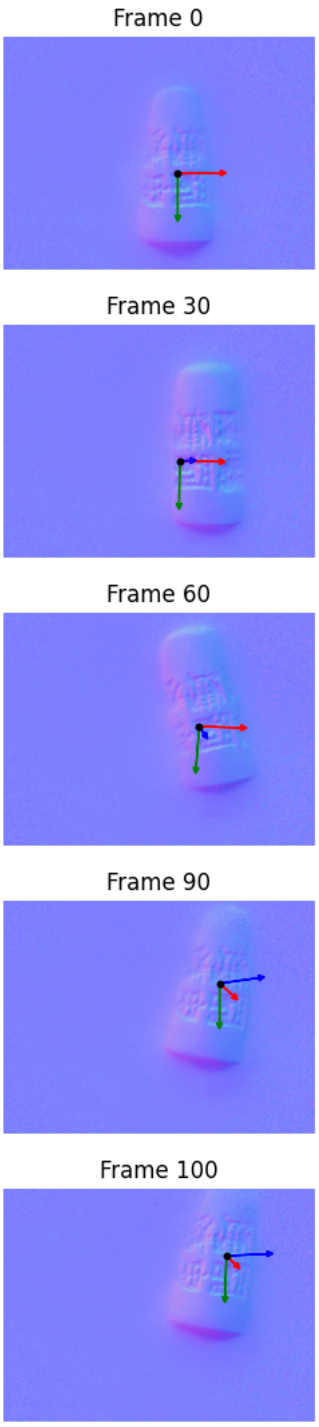
Q5.3 (a) - Track Bead Sequence

Run the snippets below. If your implementations of NormalFlow and TrackSequence are correct, each normal image will show the correct estimated 6DoF object pose with an annotated coordinate frame. Feel free to experiment with the parameters in the code.

```
"""
Helper Functions
"""

def display_normal_with_coordinate_system(I, T, center_ref, title, vector_scale=50):
    """
    Show the normal map annotating with the computed coordinate system
    :param[np.array(H, W, 3)] I     : Normal Map [float]
    :param[np.array(4, 4)] T        : The homogeneous transformation matrix
    :param[np.array(2)] center_ref  : The x-y origin of the reference frame (first frame)
    :param[float] vector_scale      : The size of the coordinate axis arrow
    """
    fig, ax = plt.subplots(figsize=(3,3))
    ax.imshow(normal2image(I))

    # Get the center and the unit_vector
    unit_vectors_ref = np.eye(3)[:, :2]
    center_3d_ref = np.array([(center_ref[0] - I.shape[1] / 2 + 0.5), (center_ref[1] - I.shape[0] / 2 + 0.5), 0])
    unit_vectors_3d_ref = np.eye(3)
    remapped_center_3d_ref = (
        np.dot(T[:3, :3], center_3d_ref) + T[:3, 3]
    )
    remapped_cx_ref = remapped_center_3d_ref[0] + I.shape[1] / 2 - 0.5
    remapped_cy_ref = remapped_center_3d_ref[1] + I.shape[0] / 2 - 0.5
    remapped_center_ref = np.array([remapped_cx_ref, remapped_cy_ref]).astype(
        np.int32
    )
    remapped_unit_vectors_ref = (
        np.dot(T[:3, :3], unit_vectors_3d_ref.T).T
    )[:, :2]

    # Annotate the origin
    ax.scatter(remapped_center_ref[0], remapped_center_ref[1], c="k", alpha=1, s=10, zorder=3)
```

```
    # Annotate the three axes
    colors = ["r", "g", "b"]
    for i in range(3):
        dx, dy = remapped_unit_vectors_ref[i] * vector_scale
        ax.arrow(
            remapped_center_ref[0], remapped_center_ref[1], dx, dy,
            head_width=6, head_length=6,
            fc=colors[i], ec=colors[i],
            linewidth=1, length_includes_head=True
        )

    ax.set_axis_off()
    ax.set_title(title)
    plt.show()

def visualize_track(Is, Ts, vis_seq=[0]):
    # Visualize tracks on a seuqence of a selected sequence
    ys, xs = np.nonzero(Cs[0])
    cx_ref = xs.mean()
    cy_ref = ys.mean()
    center_ref = np.array([cx_ref, cy_ref]).astype(np.int32)
    for idx in vis_seq:
        display_normal_with_coordinate_system(Is[idx], Ts[idx], center_ref, "Frame %d"%idx)
```

```
# load the data
data_path = "/content/bead_data.npz"
data = np.load(data_path, allow_pickle=True)
Cs = data["contact_masks"]
Is = data["normal_maps"]
Hs = data["height_maps"]

# Track the Bead
Ts = TrackSequence(Is, Hs, Cs)
# Visualize the tracking result
visualize_track(Is, Ts, [0, 30, 60, 90, 100])
```



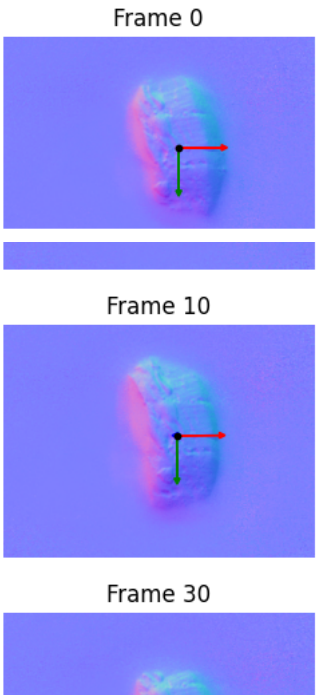

```
Start coding or generate with AI.
```

Q5.3 (b) - Track Hammer Handle Sequence

Track the handle of the hammer, which is made by wood.

```
# load the data
data_path = "/content/hammer_handle_data.npz"
data = np.load(data_path, allow_pickle=True)
Cs = data["contact_masks"]
Is = data["normal_maps"]
Hs = data["height_maps"]

# Track the Hammer Handle
Ts = TrackSequence(Is, Hs, Cs)
```

```
# Visualize the tracking result
```



16-820: Advanced Computer Vision
HW-2

Akanksha Singal (asingal2)

September 26, 2025

Section 3 Affine Motion Subtraction Ant

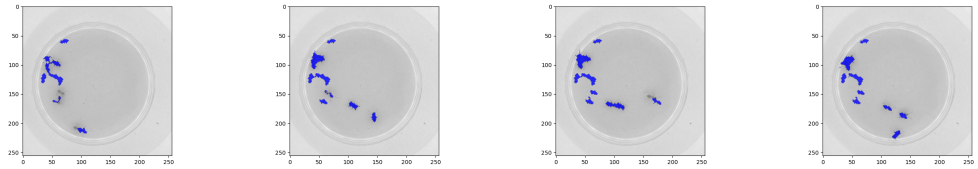


Figure 1: ITERATION FOR EROSION AND DILATION = 1, num iters = 1000, threshold = 0.01, tolerance = 0.2

Section 3 Affine Motion Subtraction Aerial

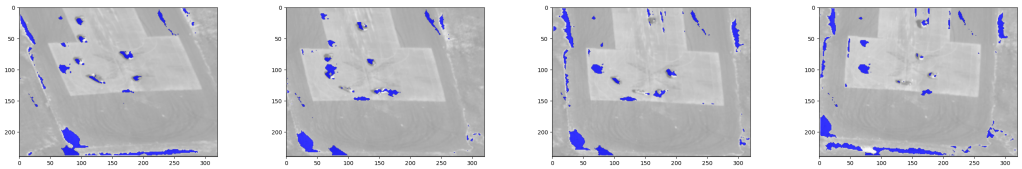


Figure 2: ITERATION FOR EROSION AND DILATION = 2, num iters = 1000, threshold = 0.01, tolerance = 0.3

Section 4 Inverse Affine Ant

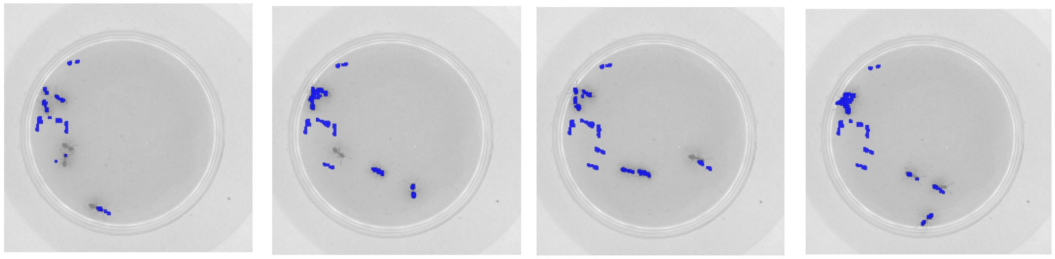


Figure 3: ITERATION FOR EROSION AND DILATION = 1, num iters = 1000, threshold = 0.02, tolerance = 0.2

Section 4 Inverse Affine Aerial

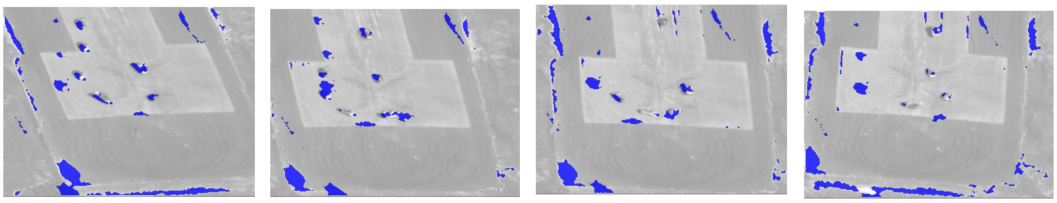


Figure 4: ITERATION FOR EROSION AND DILATION = 2, num iters = 1000, threshold = 1.0, tolerance = 0.3

Section 4 Theory

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! =====
Inverse composition Results:
Ant Sequence takes 8.863341 seconds
Aerial Sequence takes 43.708784 seconds

Without Inverse Composition:
Ant Sequence takes 13.120779 seconds
Aerial Sequence takes 1813.514410 seconds

===== 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! =====

We see that the inverse compositional approach is more computationally efficient than the classical approach. This is because in classical approach, we were computing the gradients, A matrix and hessian in each iteration. In inverse compositional approach, the gradients, A matrix and the hessian are computed once on the fixed template image and reused for all iterations making it faster. Only the inverse of hessian needs to be computed in each iteration. The difference is not very huge in my implementation as I am using opencv functions in classical approach to run on colab. Classical approach from scratch was taking up approximately 2hrs for the aerial sequence hence had to migrate to opencv and numpy functions that use cpp wrapper and are relatively faster. Also in case of ant sequence the camera movement is not as much as in aerial sequence. Hence the time difference is less in ant sequence as compared to aerial sequence.

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

Figure 5: 4.2.1 and 4.2.2

Section 5.3 b Hammer Handle sequence

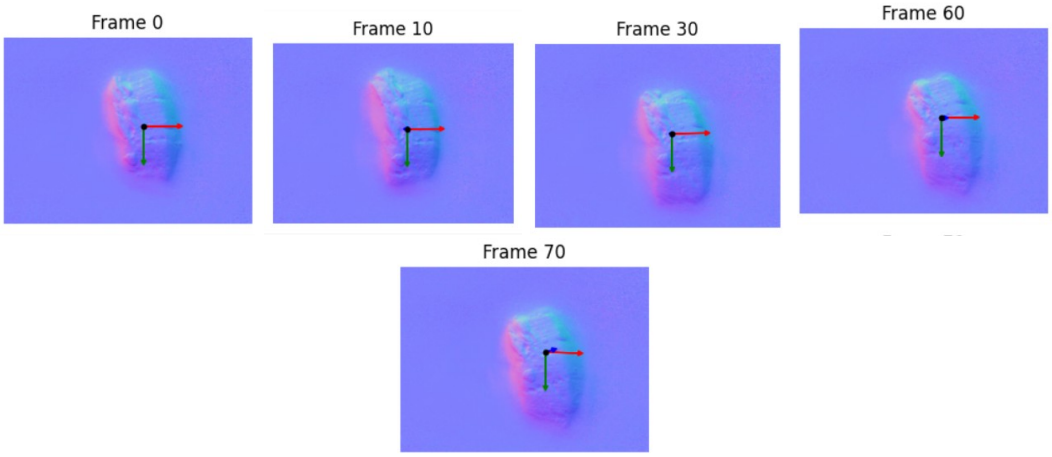


Figure 6: Normal flow part b