Q1.1

1. $W(x;p) = \begin{bmatrix} 1 & 0 & p_1 \\ 0 & 1 & p_2 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} W_x \\ W_y \end{bmatrix} = \begin{bmatrix} x + p_1 \\ y + p_2 \end{bmatrix}$

   $\frac{\partial W(x;p)}{\partial p^T} = \begin{bmatrix} \frac{\partial W_x(x;p)}{\partial p_1^T} & \frac{\partial W_x(x;p)}{\partial p_2^T} \\ \frac{\partial W_y(x;p)}{\partial p_1^T} & \frac{\partial W_y(x;p)}{\partial p_2^T} \end{bmatrix}$

   $\frac{\partial W_x(x;p)}{\partial p_1^T} = 1;\ \frac{\partial W_x(x;p)}{\partial p_2^T} = 0;\ \frac{\partial W_y(x;p)}{\partial p_1^T} = 0;\ \frac{\partial W_y(x;p)}{\partial p_2^T} = 1$

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

2. Aim: Minimize Δp

   $\arg min_p = \sum_{x \in N} \|\mathcal{L}_{t+1}(x' + \Delta p) - \mathcal{L}_t(x)\|_2^2$ (1)

   $where\ \mathcal{L}_{t+1}(x' + \Delta p) \approx \mathcal{L}_{t+1}(x') + \frac{\partial \mathcal{L}_{t+1}(x')}{\partial x'^T} \frac{\partial W(x;p)}{\partial p^T} \Delta p$ (2)

   Subs 2 in 1,

   $$arg \min_{\Delta p} \sum_{x \in N} \left\| \mathcal{L}_{t+1}(x') + \frac{\partial \mathcal{L}_{t+1}(x')}{\partial x'^T} \frac{\partial W(x;p)}{\partial p^T} \Delta p\ - \mathcal{L}_t(x) \right\|_2^2 \quad (3)$$

   $$arg \min_{\Delta p} \sum_{x \in N} \left\| \frac{\partial \mathcal{L}_{t+1}(x')}{\partial x'^T} \frac{\partial W(x;p)}{\partial p^T} \Delta p\ - (\mathcal{L}_t(x)\ -\ \mathcal{L}_{t+1}(x')) \right\|_2^2 \quad (4)$$

   $comparing\ with\ arg \min_{\Delta p} \|A\Delta p - b\|_2^2$

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

   $b = \mathcal{L}_t(x)\ -\ \mathcal{L}_{t+1}(x')$

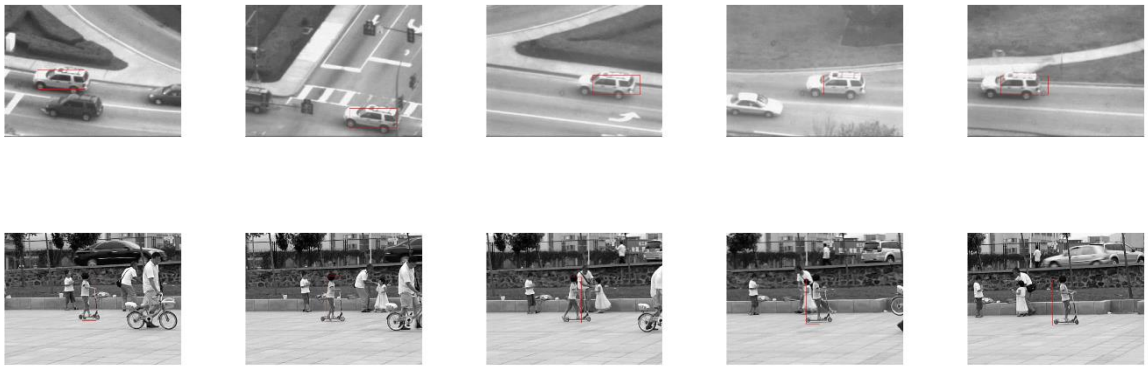
3. A^TA must be full rank. $\det(A^TA)$ should not be zero

Q1.2

```
import numpy as np
from scipy.interpolate import RectBivariateSpline
import cv2

def LucasKanade(It, It1, rect, threshold, num_iters, p0=np.zeros(2)):
    """
    :param It: template image
    :param It1: Current image
    :param rect: Current position of the car (top left, bot right coordinates)
    :param threshold: if the length of dp is smaller than the threshold,
terminate the optimization
    :param num_iters: number of iterations of the optimization
    :param p0: Initial movement vector [dp_x0, dp_y0]
    :return: p: movement vector [dp_x, dp_y]
    """

    # Put your implementation here
    # set up the threshold
    ################### TODO Implement Lucas Kanade ###################
    p = p0
    x1, y1, x2, y2 = rect

    spline = RectBivariateSpline(np.arange(It.shape[0]),
np.arange(It.shape[1]), It)
    spline1 = RectBivariateSpline(np.arange(It1.shape[0]),
np.arange(It1.shape[1]), It1)

    patchY, patchX = np.meshgrid(np.linspace(y1, y2, int(y2-y1+1),
endpoint=True), np.linspace(x1, x2, int(x2-x1+1), endpoint=True))

    for i in range(int(num_iters)):
        # Warp I with W(x;p) to compute I(W(x;p))
        WIt1 = spline1.ev(patchY + p[1], patchX + p[0])

        # Compute error image T(x) - I(W(x;p))
        error = spline.ev(patchY, patchX) - WIt1

        # Warp gradient
        dItx = spline1.ev(patchY + p[1], patchX + p[0], dx=1).reshape(1,-1)
        dIty = spline1.ev(patchY + p[1], patchX + p[0], dy=1).reshape(1,-1)
        dIt = np.hstack((dIty.T, dItx.T))

        # Evaluate Jacobian
        jacobian = np.array([[1, 0], [0, 1]])

        # Compute approximate Hessian
        preH = dIt @ jacobian
```

```
        H = preH.T @ preH

        # Compute dp
        dp = np.linalg.inv(H) @ preH.T @ error.reshape(-1, 1)

        # Update the warp parameters
        p = p + dp.flatten()

        # Check for convergence
        if np.linalg.norm(dp)**2 < threshold:
            break

    return p
```

Q1.3



https://drive.google.com/drive/folders/1f0VQblCVwo7hgFtBZEO6TCq307AGHLsz?usp=sharing

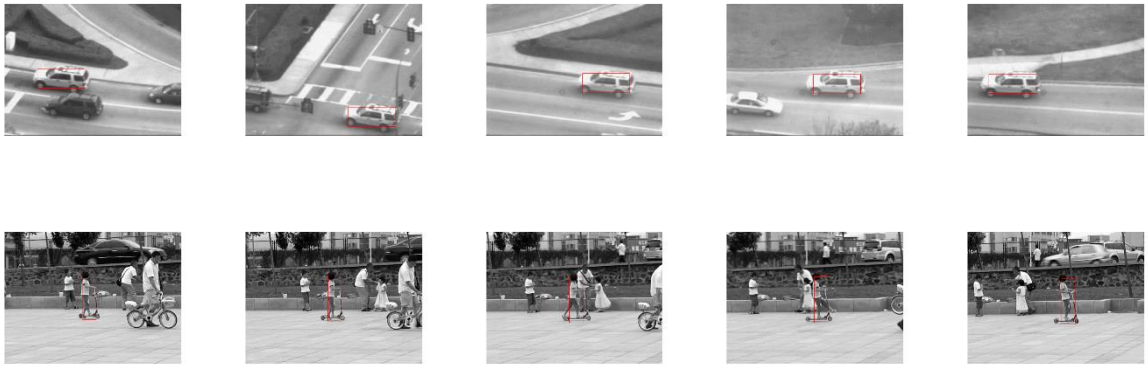
Car

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from LucasKanade import LucasKanade

parser = argparse.ArgumentParser()
parser.add_argument(
    '--num_iters', type=int, default=1e4, help='number of iterations of Lucas-
Kanade'
)
parser.add_argument(
    '--threshold',
    type=float,
    default=1e-2,
    help='dp threshold of Lucas-Kanade for terminating optimization',
)
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold

seq = np.load('C:/D/CMU/Courses/F23/16-820/hw2/data/carseq.npy')
rect = [59, 116, 145, 151]
rects = [rect]

for i in range(seq.shape[2] - 1):
    It = seq[:,:,i]
    It1 = seq[:,:,i+1]

    p = LucasKanade(It, It1, rect, threshold, num_iters)
    rect = rect + np.array([p[0], p[1], p[0], p[1]])

    rects = np.vstack((rects, rect))

    if(i == 1 or i == 100 or i == 200 or i == 300 or i == 400):
        plt.figure(figsize=(10, 10))
        plt.imshow(It1, cmap='gray')
        plt.axis('off')
        patch = patches.Rectangle((rect[0], rect[1]), rect[2]-rect[0],
rect[3]-rect[1], linewidth=1, edgecolor='r', facecolor='none')
        plt.gca().add_patch(patch)
        plt.savefig('C:/D/CMU/Courses/F23/16-
820/hw2/submission/results/carseq_{}'.format(i))

    np.save('C:/D/CMU/Courses/F23/16-820/hw2/submission/data/carseqrects.npy',
rects)
```

Girl

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from LucasKanade import LucasKanade

parser = argparse.ArgumentParser()
parser.add_argument(
    '--num_iters', type=int, default=1e4, help='number of iterations of Lucas-
Kanade'
)
parser.add_argument(
    '--threshold',
    type=float,
    default=1e-2,
    help='dp threshold of Lucas-Kanade for terminating optimization',
)
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold

seq = np.load('C:/D/CMU/Courses/F23/16-820/hw2/data/girlseq.npy')
rect = [280, 152, 330, 318]

rects = [rect]

for i in range(seq.shape[2] - 1):
    It = seq[:,:,i]
    It1 = seq[:,:,i+1]

    p = LucasKanade(It, It1, rect, threshold, num_iters)
    rect = rect + np.array([p[0], p[1], p[0], p[1]])

    rects = np.vstack((rects, rect))

    if(i == 1 or i == 20 or i == 40 or i == 60 or i == 80):
        plt.figure(figsize=(10, 10))
        plt.imshow(It1, cmap='gray')
        plt.axis('off')
        patch = patches.Rectangle((rect[0], rect[1]), rect[2]-rect[0],
rect[3]-rect[1], linewidth=1, edgecolor='r', facecolor='none')
        plt.gca().add_patch(patch)
        plt.savefig('C:/D/CMU/Courses/F23/16-
820/hw2/submission/results/girlseq_{}'.format(i))

    np.save('C:/D/CMU/Courses/F23/16-
820/hw2/submission/data/girlseqrects.npy', rects)
```

Q1.4



https://drive.google.com/drive/folders/1f0VQblCVwo7hgFtBZEO6TCq307AGHLsz?usp=sharing

Car

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from LucasKanade import LucasKanade

parser = argparse.ArgumentParser()
parser.add_argument(
    '--num_iters', type=int, default=1e4, help='number of iterations of Lucas-
Kanade'
)
parser.add_argument(
    '--threshold',
    type=float,
    default=1e-2,
    help='dp threshold of Lucas-Kanade for terminating optimization',
)
parser.add_argument(
    '--template_threshold',
    type=float,
    default=5,
    help='threshold for determining whether to update template',
)
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold
template_threshold = args.template_threshold

seq = np.load('data/carseq.npy')
rect = [59, 116, 145, 151]
rects = [rect]


It0 = seq[:,:,0]
It = seq[:,:,0]

for i in range(seq.shape[2] - 1):
    It1 = seq[:,:,i+1]

    p = LucasKanade(It, It1, rect, threshold, num_iters)
    pn = np.array(rects[-1][:2]) - np.array(rects[0][:2]) + p
    p_star = LucasKanade(It0, It1, rects[0], threshold, num_iters, pn)

    if(np.linalg.norm(p_star - pn) <= template_threshold):
        It = seq[:,:,i+1]
```

```
        p_star = np.array(rects[0][:2]) - np.array(rects[-1][:2]) + p_star
        p = p_star
    else:
        p = p

    rect = rect + np.array([p[0], p[1], p[0], p[1]])
    rects = np.vstack((rects, rect))

    if(i == 1 or i == 100 or i == 200 or i == 300 or i == 400):
        plt.figure(figsize=(10, 10))
        plt.imshow(It1, cmap='gray')
        plt.axis('off')
        patch = patches.Rectangle((rect[0], rect[1]), rect[2]-rect[0],
rect[3]-rect[1], linewidth=1, edgecolor='r', facecolor='none')
        plt.gca().add_patch(patch)
        plt.savefig('C:/D/CMU/Courses/F23/16-
820/hw2/submission/results/carseq_wcrt_{}'.format(i))

    np.save('C:/D/CMU/Courses/F23/16-820/hw2/submission/data/carseqrects-
wcrt.npy', rects)
```

Girl

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from LucasKanade import LucasKanade

parser = argparse.ArgumentParser()
parser.add_argument(
    '--num_iters', type=int, default=1e4, help='number of iterations of Lucas-
Kanade'
)
parser.add_argument(
    '--threshold',
    type=float,
    default=1e-2,
    help='dp threshold of Lucas-Kanade for terminating optimization',
)
parser.add_argument(
    '--template_threshold',
    type=float,
    default=5,
    help='threshold for determining whether to update template',
)
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold
template_threshold = args.template_threshold

seq = np.load('data/girlseq.npy')
rect = [280, 152, 330, 318]

rects = [rect]

It0 = seq[:,:,0]
It = seq[:,:,0]

for i in range(seq.shape[2] - 1):
    It1 = seq[:,:,i+1]

    p = LucasKanade(It, It1, rect, threshold, num_iters)
    pn = np.array(rects[-1][:2]) - np.array(rects[0][:2]) + p
    p_star = LucasKanade(It0, It1, rects[0], threshold, num_iters, pn)

    if(np.linalg.norm(p_star - pn) <= template_threshold):
        It = seq[:,:,i+1]
```

```
        p_star = np.array(rects[0][:2]) - np.array(rects[-1][:2]) + p_star
        p = p_star
    else:
        p = p

    rect = rect + np.array([p[0], p[1], p[0], p[1]])

    rects = np.vstack((rects, rect))

    if(i == 1 or i == 20 or i == 40 or i == 60 or i == 80):
        plt.figure(figsize=(10, 10))
        plt.imshow(It1, cmap='gray')
        plt.axis('off')
        patch = patches.Rectangle((rect[0], rect[1]), rect[2]-rect[0],
rect[3]-rect[1], linewidth=1, edgecolor='r', facecolor='none')
        plt.gca().add_patch(patch)
        plt.savefig('C:/D/CMU/Courses/F23/16-
820/hw2/submission/results/girlseq_wcrt_{}'.format(i))

    np.save('C:/D/CMU/Courses/F23/16-
820/hw2/submission/data/girlseqrects_wrct.npy', rects)
```

Q2.1

```
import numpy as np
from scipy.interpolate import RectBivariateSpline
from scipy.ndimage import affine_transform
import cv2

def LucasKanadeAffine(It, It1, threshold, num_iters):
    """
    :param It: template image
    :param It1: Current image
    :param threshold: if the length of dp is smaller than the threshold,
terminate the optimization
    :param num_iters: number of iterations of the optimization
    :return: M: the Affine warp matrix [2x3 numpy array] put your
implementation here
    """

    # put your implementation here
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])
    ################### TODO Implement Lucas Kanade Affine ###################
    p = np.zeros(6)

    spline = RectBivariateSpline(np.arange(It.shape[0]),
np.arange(It.shape[1]), It)
    spline1 = RectBivariateSpline(np.arange(It1.shape[0]),
np.arange(It1.shape[1]), It1)

    Y,X = np.meshgrid(np.linspace(0, It1.shape[1]-1, It1.shape[1],
endpoint=True), np.linspace(0, It1.shape[0]-1, It1.shape[0], endpoint=True))
    X, Y = X.flatten(), Y.flatten()
    coords = np.vstack((X, Y, np.ones(X.size)))

    for i in range(int(num_iters)):
        # Warp I with W(x;p) to compute I(W(x;p))
        # WIt1 = affine_transform(coords, M) Slow
        WIt1 = M @ coords

        # Pixels common to It and WIt1
        outside = np.nonzero(((WIt1[0] < 0) | (WIt1[0] >= It.shape[1])) |
(WIt1[1] < 0) | (WIt1[1] >= It.shape[0]))

        # Compute error image T(x) - I(W(x;p))
        error = spline.ev(Y, X) - spline1.ev(WIt1[1], WIt1[0])
        error[outside] = 0

        # Warp gradient
        dItx = spline1.ev(WIt1[1], WIt1[0], dx=1).flatten()
        dIty = spline1.ev(WIt1[1], WIt1[0], dy=1).flatten()
```

```
        # Evaluate Jacobian
        # jacobian = np.array([[X,0,Y,0,1,0], [0,X,0,Y,0,1]])
        # Crashed

        # Compute approximate Hessian
        # preH = dIt @ jacobian
        preH = np.array([dIty*X, dIty*Y, dIty, dItx*X, dItx*Y, dItx]).T
        H = preH.T @ preH

        # Compute dp
        dp = np.linalg.inv(H) @ preH.T @ error.reshape(-1, 1)

        # Update the warp parameters
        p = p + dp.flatten()

        # Update M
        M = np.array([[1.0+p[0], p[1], p[2]], [p[3], 1.0+p[4], p[5]], [0.0,
0.0, 1.0]])

        # Check for convergence
        if np.linalg.norm(dp)**2 < threshold:
            break

    return M
```

Q2.2

```python
import numpy as np
from scipy.ndimage.morphology import binary_erosion
from scipy.ndimage.morphology import binary_dilation
from scipy.ndimage import affine_transform
from LucasKanadeAffine import LucasKanadeAffine
from InverseCompositionAffine import InverseCompositionAffine

def SubtractDominantMotion(image1, image2, threshold, num_iters, tolerance):
    """
    :param image1: Images at time t
    :param image2: Images at time t+1
    :param threshold: used for LucasKanadeAffine
    :param num_iters: used for LucasKanadeAffine
    :param tolerance: binary threshold of intensity difference when computing
the mask
    :return: mask: [nxm]
    """

    # put your implementation here
    mask = np.zeros(image1.shape, dtype=bool)

    ################### TODO Implement Substract Dominent Motion
###################
    # 1. Warp the image It using M
    M = LucasKanadeAffine(image1, image2, threshold, num_iters)
    # M = InverseCompositionAffine(image1, image2, threshold, num_iters)
    WIt = affine_transform(image1, M)

    # 2. Subtract It from It+1
    difference = abs(image2 - WIt)

    # 3. Motion where the absolute difference exceeds a threshold
    mask[difference < tolerance] = 0
    mask[difference > tolerance] = 1

    # 4. Erode and dilate the mask
    struct = np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]])
    mask = binary_dilation(mask, struct, iterations=5)
    mask = binary_erosion(mask, struct, iterations=5)

    return mask.astype(bool)
```

Q2.3

https://drive.google.com/drive/folders/1f0VQblCVwo7hgFtBZEO6TCq307AGHLsz?usp=sharing

Ant

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from SubtractDominantMotion import SubtractDominantMotion

# write your script here, we recommend the above libraries for making your
animation

parser = argparse.ArgumentParser()
parser.add_argument(
    '--num_iters', type=int, default=1e3, help='number of iterations of Lucas-
Kanade'
)
parser.add_argument(
    '--threshold',
    type=float,
    default=0.01,
    help='dp threshold of Lucas-Kanade for terminating optimization',
)
parser.add_argument(
    '--tolerance',
    type=float,
    default=0.03,
    help='binary threshold of intensity difference when computing the mask',
)
parser.add_argument(
    '--seq_file',
    default='../data/antseq.npy',
)

args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold
tolerance = args.tolerance
seq_file = args.seq_file

seq = np.load(seq_file)

'''
HINT:
1. Create an empty array 'masks' to store the motion masks for each frame.
2. Set the initial mask for the first frame to False.
3. Use the SubtractDominantMotion function to compute the motion mask between
consecutive frames.
4. Use the motion 'masks; array for visualization.
'''
```

```
for i in range(seq.shape[2]-1):
    mask = SubtractDominantMotion(seq[:,:,i], seq[:,:,i+1], threshold,
num_iters, tolerance)

    if i in [30, 60, 90, 120]:
        plt.figure(figsize=(10, 10))
        plt.imshow(seq[:,:,i], cmap='gray')
        plt.imshow(mask, alpha=0.6, cmap='Blues')
        plt.axis('off')
        plt.savefig('C:/D/CMU/Courses/F23/16-
820/hw2/submission/results/ant_{}'.format(i))
```

Aerial

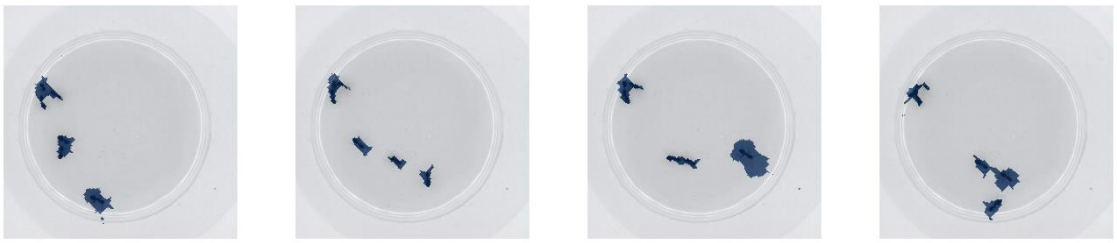
```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from SubtractDominantMotion import SubtractDominantMotion

parser = argparse.ArgumentParser()
parser.add_argument(
    '--num_iters', type=int, default=1e3, help='number of iterations of Lucas-
Kanade'
)
parser.add_argument(
    '--threshold',
    type=float,
    default=5,
    help='dp threshold of Lucas-Kanade for terminating optimization',
)
parser.add_argument(
    '--tolerance',
    type=float,
    default=0.2,
    help='binary threshold of intensity difference when computing the mask',
)
parser.add_argument(
    '--seq',
    default='C:/D/CMU/Courses/F23/16-820/hw2/data/aerialseq.npy',
)

args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold
tolerance = args.tolerance
seq_file_path = args.seq

seq = np.load(seq_file_path)

'''
HINT:
1. Create an empty array 'masks' to store the motion masks for each frame.
2. Set the initial mask for the first frame to False.
3. Use the SubtractDominantMotion function to compute the motion mask between
consecutive frames.
4. Use the motion 'masks; array for visualization.
'''
for i in range(seq.shape[2]-1):
    mask = SubtractDominantMotion(seq[:,:,i], seq[:,:,i+1], threshold,
num_iters, tolerance)
```

```
    if i in [30, 60, 90, 120]:
        plt.figure(figsize=(10, 10))
        plt.imshow(seq[:,:,i], cmap='gray')
        plt.imshow(mask, alpha=0.45, cmap='Blues')
        plt.axis('off')
        plt.savefig('C:/D/CMU/Courses/F23/16-
820/hw2/submission/results/aerial_{}'.format(i))
```

Q3.1

```
import numpy as np
from scipy.interpolate import RectBivariateSpline
from scipy.ndimage import affine_transform
import cv2

def InverseCompositionAffine(It, It1, threshold, num_iters):
    """
    :param It: template image
    :param It1: Current image
    :param threshold: if the length of dp is smaller than the threshold,
terminate the optimization
    :param num_iters: number of iterations of the optimization
    :return: M: the Affine warp matrix [2x3 numpy array]
    """

    # put your implementation here
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])

    ##################### TODO Implement Inverse Composition Affine
####################
    p = np.zeros(6)

    spline = RectBivariateSpline(np.arange(It.shape[0]),
np.arange(It.shape[1]), It)
    spline1 = RectBivariateSpline(np.arange(It1.shape[0]),
np.arange(It1.shape[1]), It1)

    X,Y = np.meshgrid(np.linspace(0, It1.shape[1]-1, It1.shape[1],
endpoint=True), np.linspace(0, It1.shape[0]-1, It1.shape[0], endpoint=True))
    Y, X = Y.flatten(), X.flatten()
    coords = np.vstack((X, Y, np.ones(X.size)))

    # Warp gradient
    dItx = spline.ev(Y, X, dx=1).flatten()
    dIty = spline.ev(Y, X, dy=1).flatten()

    # Evaluate Jacobian
    # jacobian = np.array([[X,0,Y,0,1,0], [0,X,0,Y,0,1]])

    # Compute approximate Hessian
    # preH = dIt @ jacobian Crashes
    preH = np.array([dIty*X, dIty*Y, dIty, dItx*X, dItx*Y, dItx]).T
    H = preH.T @ preH
    postH = np.linalg.inv(H) @ preH.T

    for i in range(int(num_iters)):
        # Warp I with W(x;p) to compute I(W(x;p))
```

```
        # WIt1 = affine_transform(coords, M) Slow
        WIt1 = M @ coords

        # Pixels common to It and WIt1
        outside = np.nonzero(((WIt1[0] < 0) | (WIt1[0] >= It.shape[1])) |
(WIt1[1] < 0) | (WIt1[1] >= It.shape[0]))

        # Compute error image T(x) - I(W(x;p))
        error = spline.ev(Y, X) - spline1.ev(WIt1[1], WIt1[0])
        error[outside] = 0

        # Debug
        # import pdb; pdb.set_trace()

        # Compute dp
        dp = postH @ error.reshape(-1, 1)

        # Update the warp parameters
        p = p + dp.flatten()

        # Update M
        delM = np.array([[1.0+p[0], p[1], p[2]], [p[3], 1.0+p[4], p[5]], [0.0,
0.0, 1.0]], dtype=np.float)
        M = M @ np.linalg.inv(delM)

        # Check for convergence
        if np.linalg.norm(dp)**2 < threshold:
            break

    return M
```



https://drive.google.com/drive/folders/1pl2cv8yBz8C2CX5NsZ0180yNYJyzIRV5?usp=sharing

Q3.2

In inverse compositional approach, the gradient, the jacobian and the hessian are precomputed. The repeated computation in classical approach is avoided. Therefore, making it more efficient.