CV HW2

Lifan Yu lifany

September 2023

1.1

1.

$\frac{\partial W(\mathbf{x};\mathbf{p})}{\partial p^T}$ is the Jacobian of the image coordinates with respect to warp parameters p, and because we are given a pure translation, the warp function is

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

The Jacobian is

$$\frac{\partial W(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T} = \begin{bmatrix} \frac{\partial W_x(\mathbf{x};\mathbf{p})}{\partial p_1} & \frac{\partial W_x(\mathbf{x};\mathbf{p})}{\partial p_2} \\ \frac{\partial W_y(\mathbf{x};\mathbf{p})}{\partial p_1} & \frac{\partial W_y(\mathbf{x};\mathbf{p})}{\partial p_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

2.

We minimize for

$$\sum_{x \in N} ||I_{t+1}(\mathbf{x} + \mathbf{p}) - I_t(\mathbf{x})||_2^2$$

Substitute for

$$I_{t+1}(\mathbf{x}' + \Delta\mathbf{p}) \approx I_{t+1}(\mathbf{x}') + \frac{\partial I_{t+1}(\mathbf{x}')}{\partial {\mathbf{x}'}^T} \frac{\partial W(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T} \Delta\mathbf{p}$$

Where $\mathbf{x}' = W(\mathbf{x};\mathbf{p}) = \mathbf{x} + \mathbf{p}$
We arrange the expression into $||\mathbf{A}\Delta\mathbf{p} - \mathbf{b}||_2^2$ as follows

$$\sum_{x \in N} ||\frac{\partial I_{t+1}(\mathbf{x}')}{\partial {\mathbf{x}'}^T} \frac{\partial W(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T} \Delta\mathbf{p} - (I_t(\mathbf{x}) - I_{t+1}(\mathbf{x}'))||_2^2$$

Where

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

$$b = \sum_{x \in N} [I_t(\mathbf{x}) - I_{t+1}(\mathbf{x}')]$$

where $\mathbf{x}' = W(\mathbf{x};\mathbf{p})$

Writing into matrix forms, and substituting $\frac{\partial W(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

$$A = \begin{bmatrix} \frac{\partial I_{t+1}(\mathbf{x}'_\mathbf{1})}{\partial {\mathbf{x}'_\mathbf{1}}^T} * \frac{\partial W(\mathbf{x_1};\mathbf{p})}{\partial \mathbf{p}^T} \\ \frac{\partial I_{t+1}(\mathbf{x}'_\mathbf{2})}{\partial {\mathbf{x}'_\mathbf{2}}^T} * \frac{\partial W(\mathbf{x_2};\mathbf{p})}{\partial \mathbf{p}^T} \\ \cdots\cdots \\ \frac{\partial I_{t+1}(\mathbf{x}'_\mathbf{D})}{\partial {\mathbf{x}'_\mathbf{D}}^T} * \frac{\partial W(\mathbf{x_D};\mathbf{p})}{\partial \mathbf{p}^T} \end{bmatrix} = \begin{bmatrix} \frac{\partial I_{t+1}(\mathbf{x}'_\mathbf{1})}{\partial x} & \frac{\partial I_{t+1}(\mathbf{x}'_\mathbf{1})}{\partial y} \\ \frac{\partial I_{t+1}(\mathbf{x}'_\mathbf{2})}{\partial x} & \frac{\partial I_{t+1}(\mathbf{x}'_\mathbf{2})}{\partial y} \\ \cdots\cdots & \\ \frac{\partial I_{t+1}(\mathbf{x}'_\mathbf{D})}{\partial x} & \frac{\partial I_{t+1}(\mathbf{x}'_\mathbf{D})}{\partial y} \end{bmatrix}$$

$$b = \begin{bmatrix} I_t(\mathbf{x_1}) - I_{t+1}(\mathbf{x'_1}) \\ I_t(\mathbf{x_2}) - I_{t+1}(\mathbf{x'_2}) \\ ..... \\ I_t(\mathbf{x_D}) - I_{t+1}(\mathbf{x'_D}) \end{bmatrix}$$

where $\mathbf{x'_i} = W(\mathbf{x_i}; \mathbf{p}), i \in 1, 2, ..., D$

3.

The $A^T A$ matrix must satisfy:

(1) It should be invertible

(2) It should be symmetric

(3) It should be positive semi definite

1.2 Lucas Kanade Code

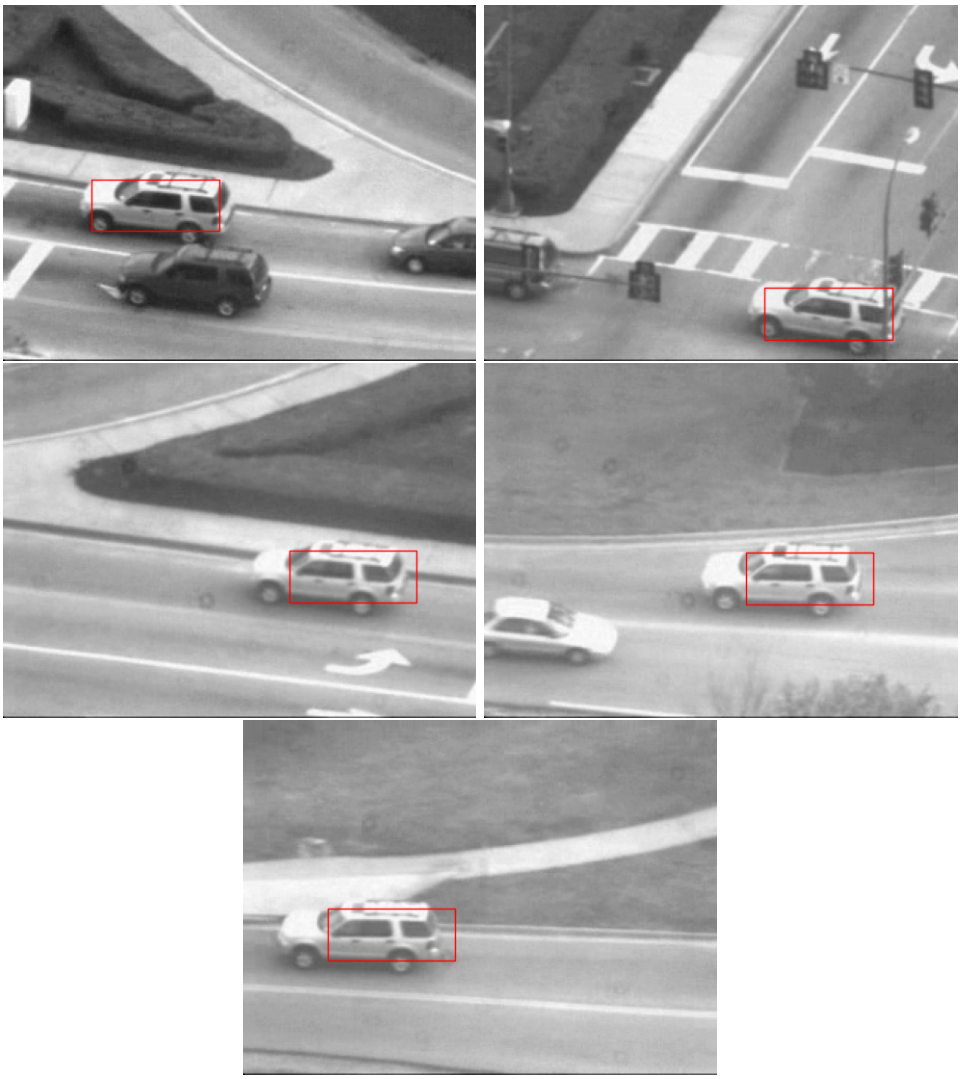
lucasKanade.py

```
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
    Ith, Itw = It.shape[0], It.shape[1]
    x1, y1, x2, y2 = rect[0], rect[1], rect[2], rect[3]

    # interpolate template & current image
    # allign its matrix indices with plotting (x, y) coordinates by transposing It
    yarr = np.arange(0, Ith, 1)
    xarr = np.arange(0, Itw, 1)
    It_interp = RectBivariateSpline(xarr, yarr, It.T)
    It1_interp = RectBivariateSpline(xarr, yarr, It1.T)

    # get points inside the rectangle
    It_xrect =  np.arange(x1, x2, 1)
    It_yrect =  np.arange(y1, y2, 1)
    D = len(It_xrect)*len(It_yrect)
    It_xgrid, It_ygrid =  np.meshgrid(It_xrect, It_yrect, indexing="ij")

    # interpolated values inside the template's rectangle
    It_interp_rect = It_interp.ev(It_xgrid, It_ygrid).reshape((D, 1))

    ### -- update p to minimize error
    # A * delta_p = b
    delta_p = np.Inf
    it = 0
    while it < num_iters and np.linalg.norm(delta_p) > threshold:

      # current image: points inside the rectangle into a meshgrid
      It1_xrect =  np.arange(x1, x2, 1) + p[0] # x coordinates
      It1_yrect =  np.arange(y1, y2, 1) + p[1] # y coordinates
      It1_xgrid, It1_ygrid =  np.meshgrid(It1_xrect, It1_yrect, indexing="ij")

      # compute matrix A from gradient of I over (x, y) in current image's rectangle
      It1_xgrad = It1_interp.ev(It1_xgrid, It1_ygrid, dx = 1, dy = 0).reshape((D, 1))
      It1_ygrad = It1_interp.ev(It1_xgrid, It1_ygrid, dx = 0, dy = 1).reshape((D, 1))
      A = np.hstack((It1_xgrad, It1_ygrad))

      # compute b
      It1_interp_rect = It1_interp.ev(It1_xgrid, It1_ygrid).reshape((D, 1))
      b = It_interp_rect - It1_interp_rect

      # compute delta_p
      ATA_i = np.linalg.inv(np.matmul(A.T, A))
      ATb = np.matmul(A.T, b)
      delta_p = np.matmul(ATA_i, ATb)

      # update p0
      delta_p = delta_p.reshape((2, ))
      p += delta_p
      it += 1

    return p
```

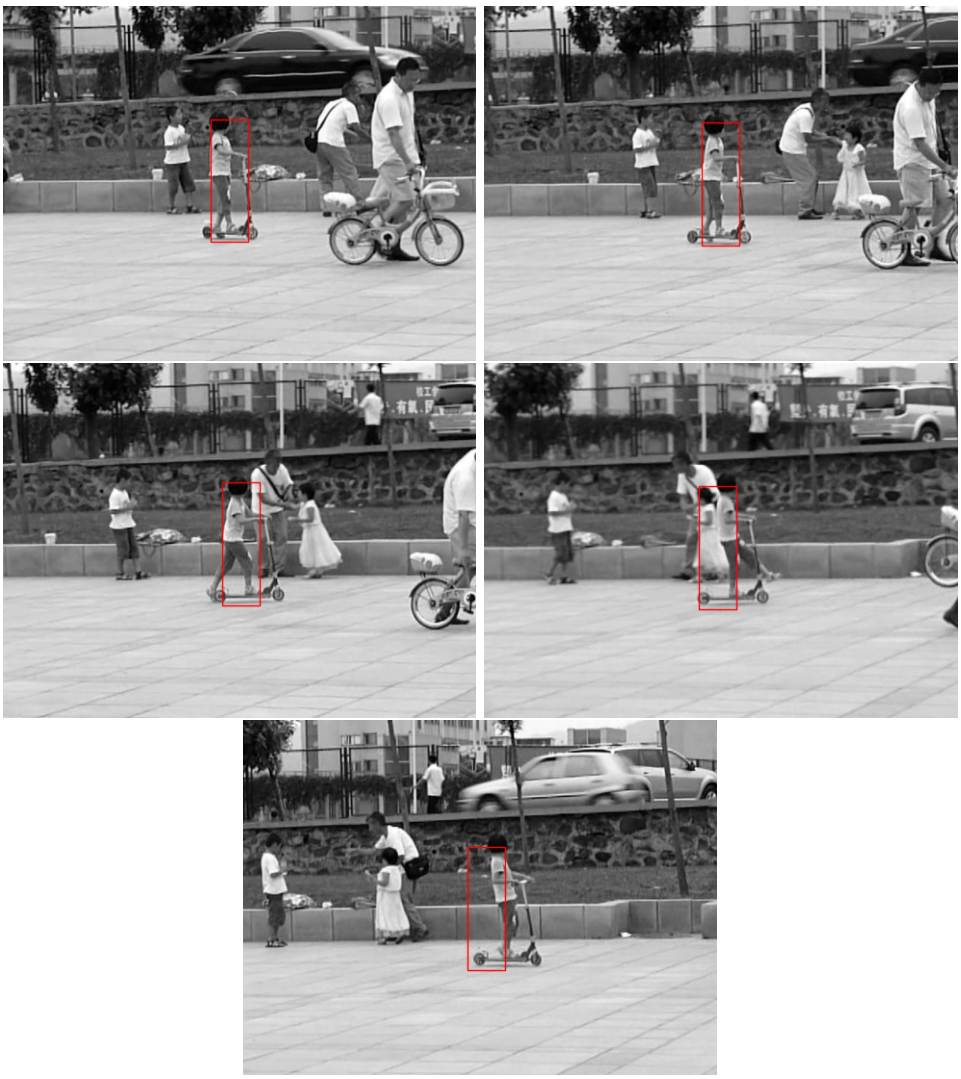
1.3 Lucas Kanade Results

Result Images

Results from the 1, 20, 40, 60, and 80th frames fpr car sequences



Results from the 1, 20, 40, 60, and 80th frames fpr girl sequences



testCarSequence.py

testCarSequence.py and testGirlSequence.py are written in a similar way

```
python testCarSequence.py --num_iters 1000 --threshold 0.0002
```

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

import scipy
import matplotlib
from LucasKanade import *

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

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("../data/carseq.npy")
rect = [59, 116, 145, 151]

# -- my test code
# plot image with rectangle
def save_plot(im, r, fname):
  fig, ax = plt.subplots(figsize=plt.figaspect(im))
  ax.set_axis_off()
  plt.gray()
  ax.imshow(im)
  x1, y1, x2, y2 = r[0], r[1], r[2], r[3]
  w = x2 - x1
  h = y2 - y1
  rect_patch = patches.Rectangle((x1, y1), w, h, linewidth=1, edgecolor='r', facecolor='none')
  ax.add_patch(rect_patch)
  plt.savefig(fname, bbox_inches='tight', pad_inches=0)

# list of all the rectangles
rects = [rect]

# iterate through each image, get delta p and plot rectangles
img_count = seq.shape[2]
for i in range(1, img_count):
  It = seq[:, :, i-1]
  It1 = seq[:, :, i]
  p = LucasKanade(It, It1, rect, threshold, num_iters, p0=np.zeros(2))
  x1, y1, x2, y2  = rect[0]+p[0], rect[1]+p[1], rect[2]+p[0], rect[3]+p[1]
  rect = [x1, y1, x2, y2]
  rects.append(rect)
  if i == 1 or i % 100 == 0:
    save_plot(It1, rect, '1-3-car-iter' + str(i) + '.png')

# save the rectangles as .npy file
np.save('carseqrects.npy', rects)
```

1.4 Template Correction

A way to solve the template drifting problem in previous results is shown in Iain Matthews et al. (2003, https://www.ri.cmu.edu/publication_view.html?pub_id=4433)

The paper suggests that "One possibility is to keep the first template T1(x) around and use it to correct the drift in Tn+1(x). For example, we could take the estimate of Tn+1(x) computed in Strategy 2 and then align Tn+1(x) to T1(x) to eliminate the drift."

Formulas from the paper are as follows

To correct the drift in Strategy 2, we therefore propose to compute updated parameters:

$$\mathbf{p}_n^* = \text{gd} \min_{\mathbf{p}=\mathbf{p}_n} \sum_{\mathbf{x}\in T_1} [I_n(\mathbf{W}(\mathbf{x};\mathbf{p})) - T_1(\mathbf{x})]^2. \quad (5)$$

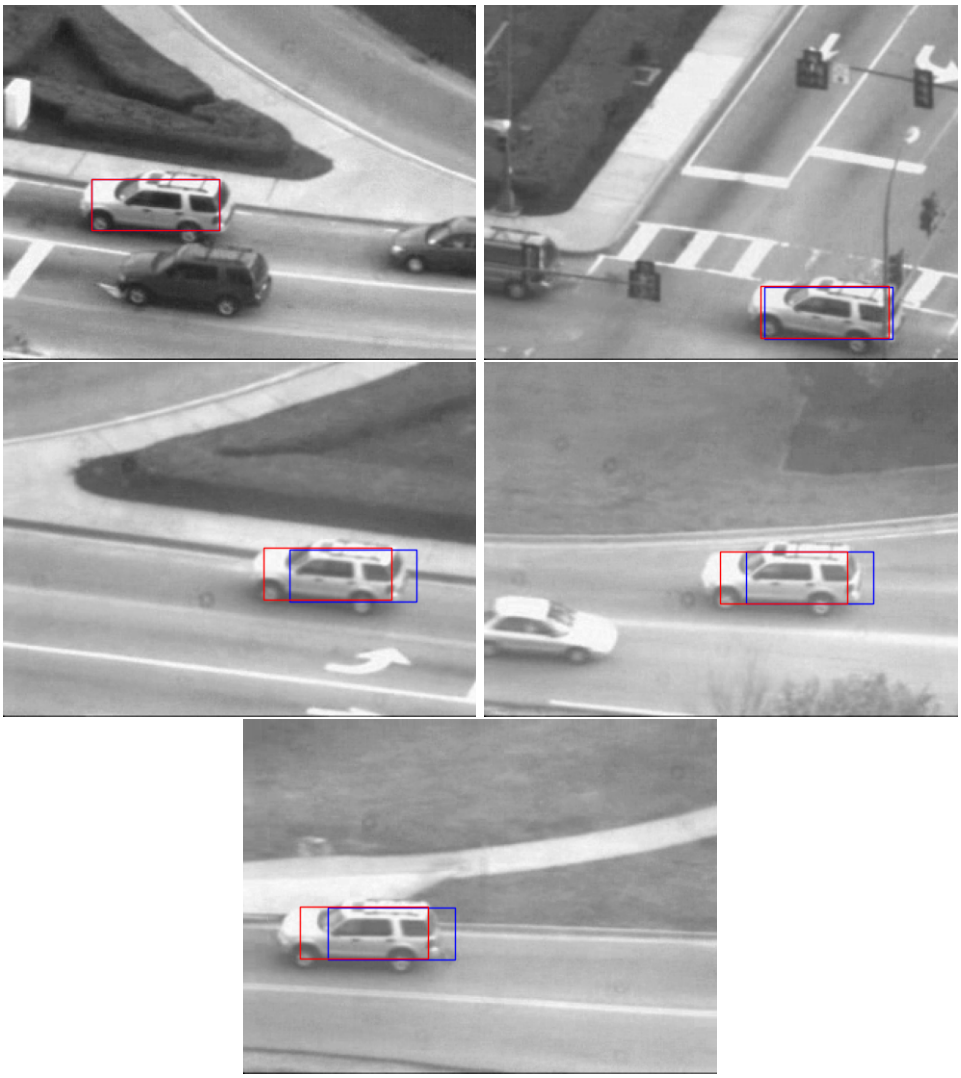
Strategy 3: Template Update with Drift Correction

If $\|\mathbf{p}_n^* - \mathbf{p}_n\| \leq \varepsilon$ then $T_{n+1}(\mathbf{x}) = I_n(\mathbf{W}(\mathbf{x};\mathbf{p}_n^*))$

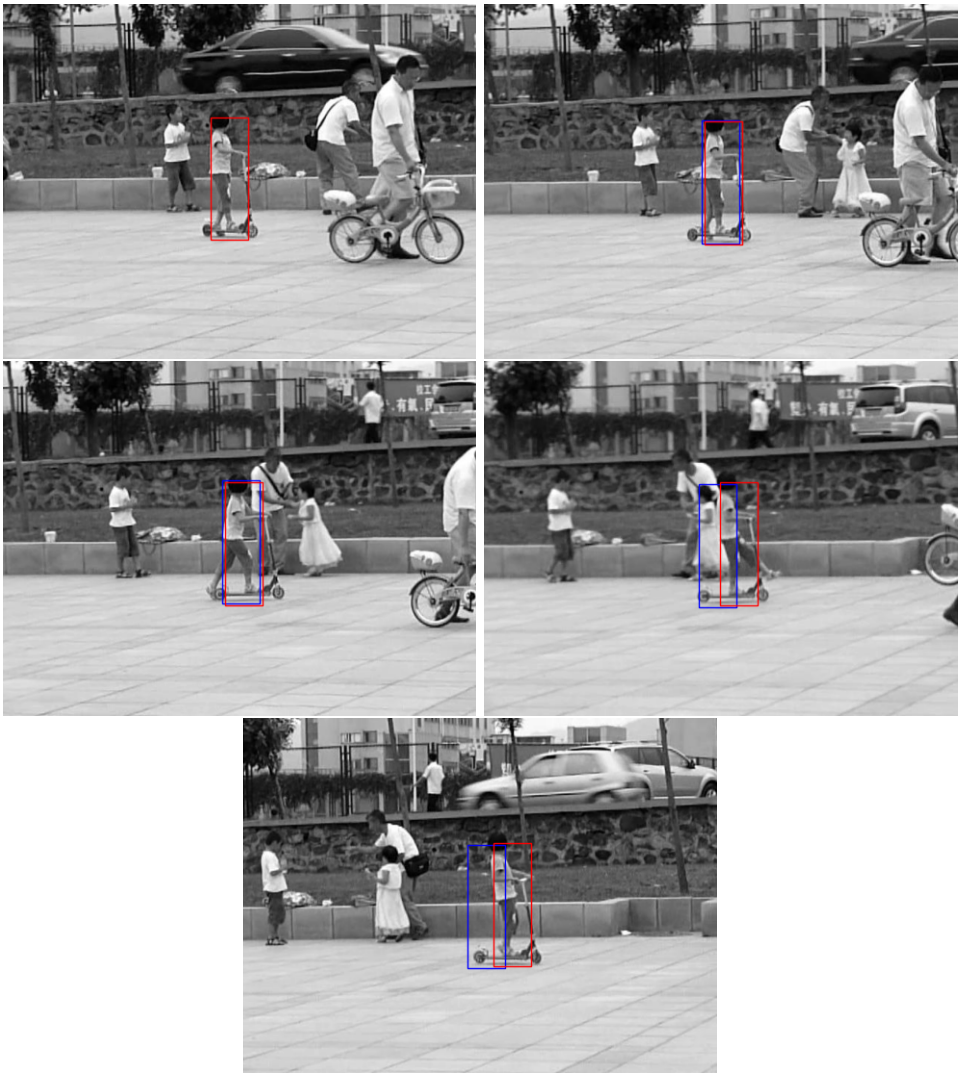
else $T_{n+1}(\mathbf{x}) = T_n(\mathbf{x})$

Result Images

The red rectangles are tracking results with template correction. The blue rectangles are tracking results without template correction.



Template correction results from the 1, 100, 200, 300, and 400th frames (car sequence)



Template correction results from the 1, 20, 40, 60, and 80th frames (girl sequence)

testCarSequenceWithTemplateCorrection.py

I run LucasKanade function twice, the first computes p_n, the second computes p_n^*. Only the test script needs to be changed, not necessarily the function. testGirlSequenceWithTemplateCorrection.py is implemented in a similar way.

```
python testCarSequenceWithTemplateCorrection.py --num_iters 100 --threshold 0.0002
```

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

import scipy
import matplotlib
from LucasKanade import *

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

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]
# -- my test code
# plot image with rectangle
def save_plot(im, r, rp, fname):
  fig, ax = plt.subplots(figsize=plt.figaspect(im))
  ax.set_axis_off()
  plt.gray()
  ax.imshow(im)
  x1, y1, x2, y2, x1p, y1p, x2p, y2p = r[0], r[1], r[2], r[3], rp[0], rp[1], rp[2], rp[3]
  w, h = x2 - x1, y2 - y1
  rect_patch_p = patches.Rectangle((x1p, y1p), w, h, linewidth=1, edgecolor='b', facecolor='none')
  rect_patch = patches.Rectangle((x1, y1), w, h, linewidth=1, edgecolor='r', facecolor='none')
  ax.add_patch(rect_patch_p)
  ax.add_patch(rect_patch)
  plt.savefig(fname, bbox_inches='tight', pad_inches=0)

# list of all the rectangles
rects = [rect]

# -- additional variables for template correction
# the initial template image
I0 = seq[:, :, 0]
# the initial rectangle
rect0 = [item for item in rect]
# the template image at each step
It = seq[:, :, 0]
# for plotting
rects_prev = np.load("carseqrects.npy")
# --

# iterate through each image, get delta p and plot rectangles
img_count = seq.shape[2]
for i in range(1, img_count):
  It1 = seq[:, :, i]
  # -- template correction steps
  # the total shift so far from image0 to imagen
  p_sofar = np.array([rect[0] - rect0[0], rect[1] - rect0[1]])
  # pn at current step based on current template
  pn = LucasKanade(It, It1, rect, threshold, num_iters, p0=np.zeros(2))
  # the total shift so far + the newly calculate pn: p_{p=p_n} W(x;p)
  p0_n = p_sofar + pn
  # pn_star is a shift at current step, so we subtract the total shift so far
  pn_star = LucasKanade(I0, It1, rect0, threshold, num_iters, p0=p0_n) - p_sofar
  # update template
  if np.linalg.norm(pn_star - pn, ord=1) <= template_threshold:
    p = pn_star
    It = It1
  # --

  x1, y1, x2, y2  = rect[0]+p[0], rect[1]+p[1], rect[2]+p[0], rect[3]+p[1]
  rect = [x1, y1, x2, y2]
  rects.append(rect)

  if i == 1 or i % 100 == 0:
    rect_prev = rects_prev[i]
    save_plot(It1, rect, rect_prev, '1-3-car-wcrt-iter' + str(i) + '.png')

# save the rectangles as .npy file
np.save('carseqrects-wcrt.npy', rects)
```

2.1 Affine Motion: Dominant Motion Estimation

With affine transformation, we haev

$$\mathbf{x}' = W(\mathbf{x};\mathbf{p}) = \begin{bmatrix} 1+p_1 & p_2 \\ p_4 & 1+p_5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} p_3 \\ p_6 \end{bmatrix}$$

With homogeneous coordinates $\tilde{\mathbf{x}}' = \mathbf{M}\tilde{\mathbf{x}}$

$$\tilde{\mathbf{x}}' = \begin{bmatrix} 1+p_1 & p_2 & p_3 \\ p_4 & 1+p_5 & p_6 \\ 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{x}}$$

Jacobian

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

Similar to Question 1.1, we compute $A\Delta p = b$ to solve for Δp. Matrix A is constructed as follows

$$A = \frac{\partial I_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T}\frac{\partial W(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T} = \begin{bmatrix} \frac{\partial I_{t+1}(\mathbf{x}'_1)}{\partial x}x_1 & \frac{\partial I_{t+1}(\mathbf{x}'_1)}{\partial x}y_1 & \frac{\partial I_{t+1}(\mathbf{x}'_1)}{\partial x} & \frac{\partial I_{t+1}(\mathbf{x}'_1)}{\partial y}x_1 & \frac{\partial I_{t+1}(\mathbf{x}'_1)}{\partial y}y_1 & \frac{\partial I_{t+1}(\mathbf{x}'_1)}{\partial y} \\ \frac{\partial I_{t+1}(\mathbf{x}'_2)}{\partial x}x_2 & \frac{\partial I_{t+1}(\mathbf{x}'_2)}{\partial x}y_2 & \frac{\partial I_{t+1}(\mathbf{x}'_2)}{\partial x} & \frac{\partial I_{t+1}(\mathbf{x}'_2)}{\partial y}x_2 & \frac{\partial I_{t+1}(\mathbf{x}'_2)}{\partial y}y_2 & \frac{\partial I_{t+1}(\mathbf{x}'_2)}{\partial y} \\ \cdots\cdots & & & & & \\ \frac{\partial I_{t+1}(\mathbf{x}'_D)}{\partial x}x_D & \frac{\partial I_{t+1}(\mathbf{x}'_D)}{\partial x}y_D & \frac{\partial I_{t+1}(\mathbf{x}'_D)}{\partial x} & \frac{\partial I_{t+1}(\mathbf{x}'_D)}{\partial y}x_D & \frac{\partial I_{t+1}(\mathbf{x}'_D)}{\partial y}y_D & \frac{\partial I_{t+1}(\mathbf{x}'_D)}{\partial y} \end{bmatrix}$$

LucasKanadeAffine.py

Discussed approach in a study group with Bhuvan Jhamb (bjhamb) and Roshan Roy (roshanr)

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

import matplotlib.pyplot as plt
import matplotlib.patches as patches

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 ###################

    # initialize p0
    p = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

    # interpolate template & current image
    # allign its matrix indices with plotting (x, y) coordinates by transposing It
    Ith, Itw = It.shape[0], It.shape[1]
    yarr = np.arange(0, Ith)
    xarr = np.arange(0, Itw)
    yarr_reshape = yarr.reshape((Ith,1))
    xarr_reshape = xarr.reshape((Itw,1))
    D = Ith*Itw # total number of points
    It1_interp = RectBivariateSpline(xarr, yarr, It1.T)

    # get gradients of It1
    xall = np.array([[i for j in range(Ith)]for i in range(Itw)])
    yall = np.array([[j for j in range(Ith)]for i in range(Itw)])
    It1_xgrad = It1_interp.ev(xall, yall, dx = 1, dy = 0)
    It1_ygrad = It1_interp.ev(xall, yall, dx = 0, dy = 1)
```

```
    ### -- update p to minimize error, calculate A * delta_p = b
    delta_p = np.Inf
    it = 0
    while it < num_iters and np.linalg.norm(delta_p) > threshold:
      # find the points within the two images' common area
      It1_warped = affine_transform(It1.T, M)
      # the common area mask
      mask = np.where(It1_warped > 0, 1, 0)

      # construct matrix A
      # each element of A: multiplication of gradient and coordinate values
      Idxx = (np.where(mask, It1_xgrad, 0) * xall).reshape(D, 1)
      Idxy = (np.where(mask, It1_xgrad, 0) * yall).reshape(D, 1)
      Idyx = (np.where(mask, It1_ygrad, 0) * xall).reshape(D, 1)
      Idyy = (np.where(mask, It1_ygrad, 0) * yall).reshape(D, 1)
      It1_xgrad = np.where(mask, It1_xgrad, 0)
      It1_ygrad = np.where(mask, It1_ygrad, 0)
      It1_xgrad_flat = It1_xgrad.reshape(D, 1)
      It1_ygrad_flat = It1_ygrad.reshape(D, 1)
      A = np.hstack((Idxx, Idxy, It1_xgrad_flat, Idyx, Idyy, It1_ygrad_flat))

      # construct b
      b = It.T - It1_warped
      b = np.where(mask, b, 0)
      b = b.reshape(D, 1)

      # compute delta_p
      ATA_i = np.linalg.inv(np.matmul(A.T, A))
      ATb = np.matmul(A.T, b)
      delta_p = np.matmul(ATA_i, ATb)

      # update p0
      delta_p = delta_p.reshape((6, ))
      p += delta_p

      # update M with p
      M = np.array([[1 + p[0], p[1], p[2]], [p[3], 1 + p[4], p[5]], [0, 0, 1]])
      it += 1
    return np.linalg.inv(M)
```

2.2 Subtract Dominant Motion

subtractDominantMotion.py

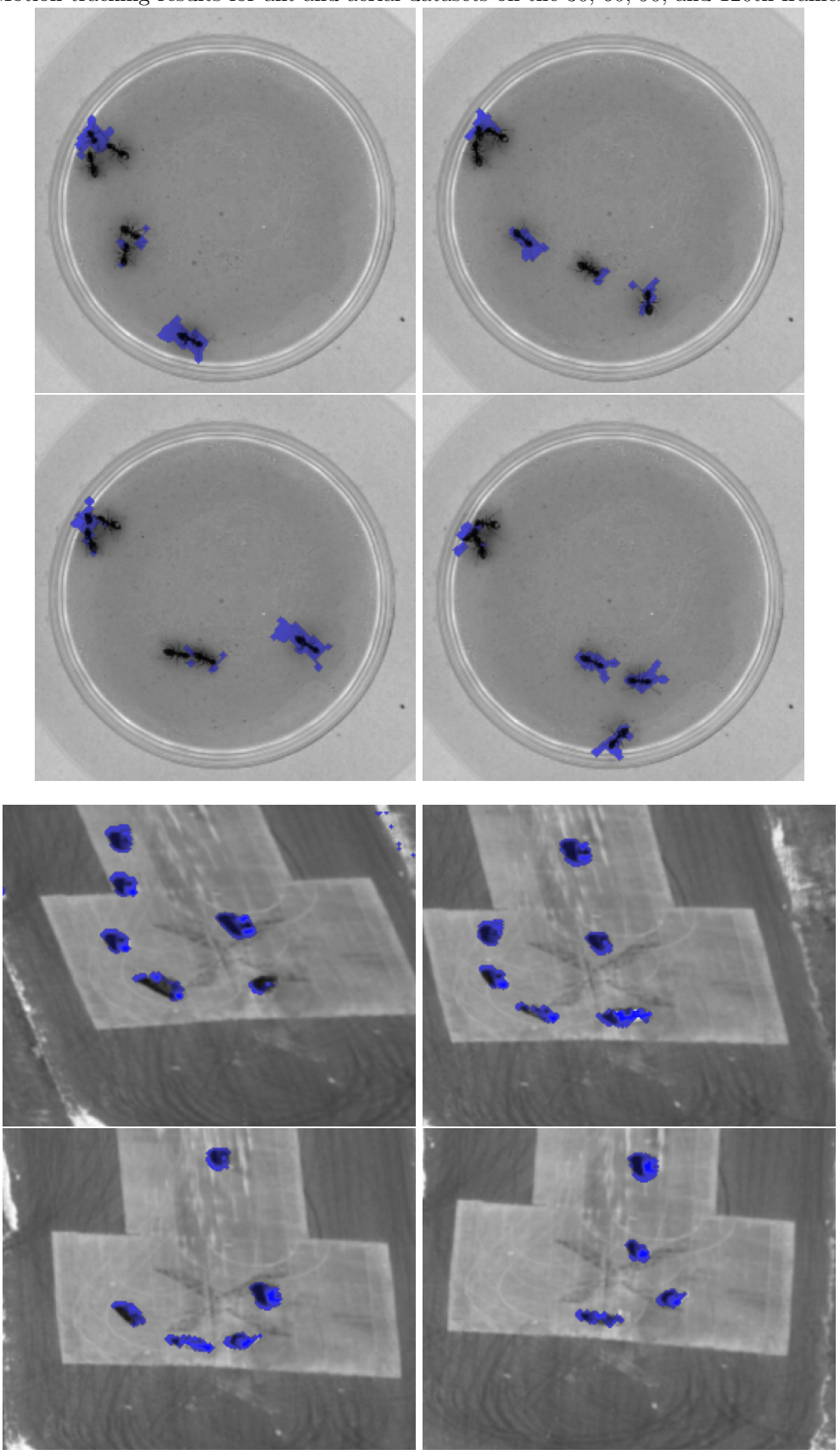
Here dilation and erosion are used to get rid of edges or insignificant points on the mask. Depending on the specific dataset we are working with, the numbers of dilations and erosions are subject to change.

```
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

import matplotlib.pyplot as plt

def SubtractDominantMotion(image1, image2, threshold, num_iters, tolerance, dataname, inverse_affine
                                                    ):
    """
    :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.ones(image1.shape, dtype=bool)

    ################### TODO Implement Substract Dominent Motion ###################
    if inverse_affine:
      M = LucasKanadeAffine(image1, image2, threshold, num_iters)
      M = np.linalg.inv(np.vstack((M, np.array([0, 0, 1]))))
    else:
      M = InverseCompositionAffine(image1, image2, threshold, num_iters)
      M = np.linalg.inv(np.vstack((M, np.array([0, 0, 1]))))

    image1_warped = affine_transform(image1.T, M).T
    sub = np.absolute(image2 - image1_warped)
    mask = np.where(sub > tolerance, 1, 0)

    # -- for aerial
    if "aerial" in dataname:
      mask = binary_erosion(mask).astype(mask.dtype)
      for i in range(3):
        mask = binary_dilation(mask).astype(mask.dtype)
      for i in range(2):
        mask = binary_erosion(mask).astype(mask.dtype)

    # -- for ant
    elif "ant" in dataname:
      for i in range(6):
        mask = binary_dilation(mask).astype(mask.dtype)
      for i in range(4):
        mask = binary_erosion(mask).astype(mask.dtype)

    return mask.astype(bool)
```

2.3 testAntSequence.py and testAerialSequence.py

Result Images

Motion tracking results for ant and aerial datasets on the 30, 60, 90, and 120th frames



Code for testAntSequence.py and testAerialSequence.py

To run testAntSequence.py and testAerialSequence.py

```
python testAntSequence.py --num_iters 1000 --tolerance 0.1 --threshold 0.0001
python testAerialSequence.py --num_iters 1000 --tolerance 0.07 --threshold 0.0002
```

Code for testAntSequence.py

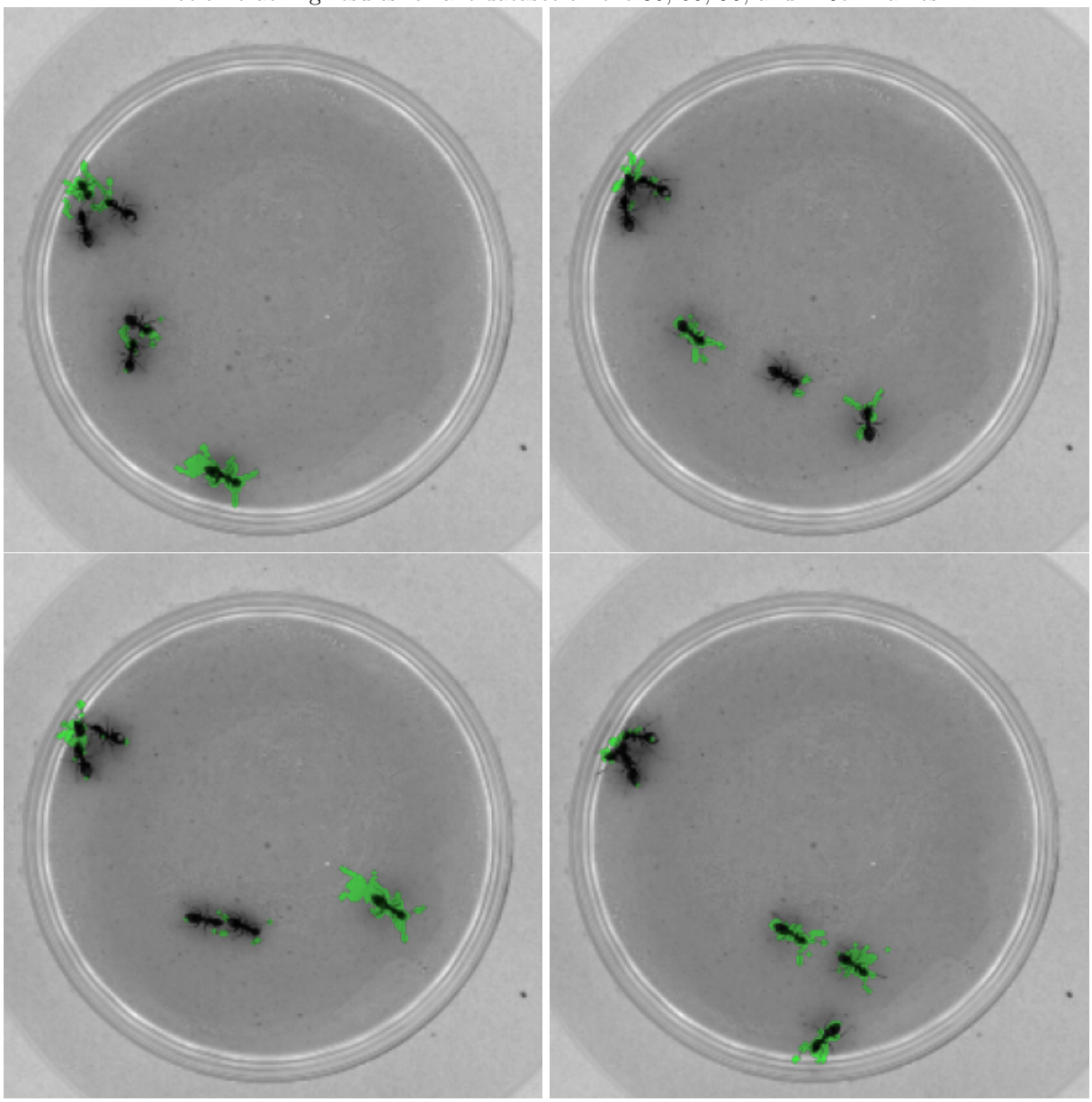
```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from scipy.ndimage import affine_transform
from LucasKanadeAffine import *
from SubtractDominantMotion import *

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=1e-2,
    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_file',
    default='../data/antseq.npy',
)
parser.add_argument(
  '--inverse',
  default = False
)
args = parser.parse_args()
num_iters = args.num_iters
threshold = args.threshold
tolerance = args.tolerance
seq_file = args.seq_file
inverse_affine = args.inverse
seq = np.load(seq_file)

def save_plot(im, mask, fname, inverse_affine):
  fig, ax = plt.subplots(figsize=plt.figaspect(im))
  ax.set_axis_off()
  ax.imshow(im, cmap='gray')
  zeros = np.zeros(im.shape)
  I = np.dstack([zeros, zeros, zeros, zeros])
  x, y = np.where(mask == True)
  if inverse_affine:
    for i in range(len(x)):
      I[x[i], y[i], :] = [0, 128, 0, im[x[i]][y[i]]]
    fname = "i-" + fname
  else:
    for i in range(len(x)):
      I[x[i], y[i], :] = [0, 0, 255, im[x[i]][y[i]]]
  ax.imshow(I)
  plt.savefig(fname, bbox_inches='tight', pad_inches=0)

img_count = seq.shape[2]
for i in range(1, img_count):
  It = seq[:, :, i-1]
  It1 = seq[:, :, i]
  mask = SubtractDominantMotion(It, It1, threshold, num_iters, tolerance, seq_file, inverse_affine)
  if i == 1 or i % 30 == 0:
    save_plot(It1, mask, "2-3-ant-iter-" + str(i) + ".png", inverse_affine)
```

3.1 Efficient Tracking using Inverse Composition

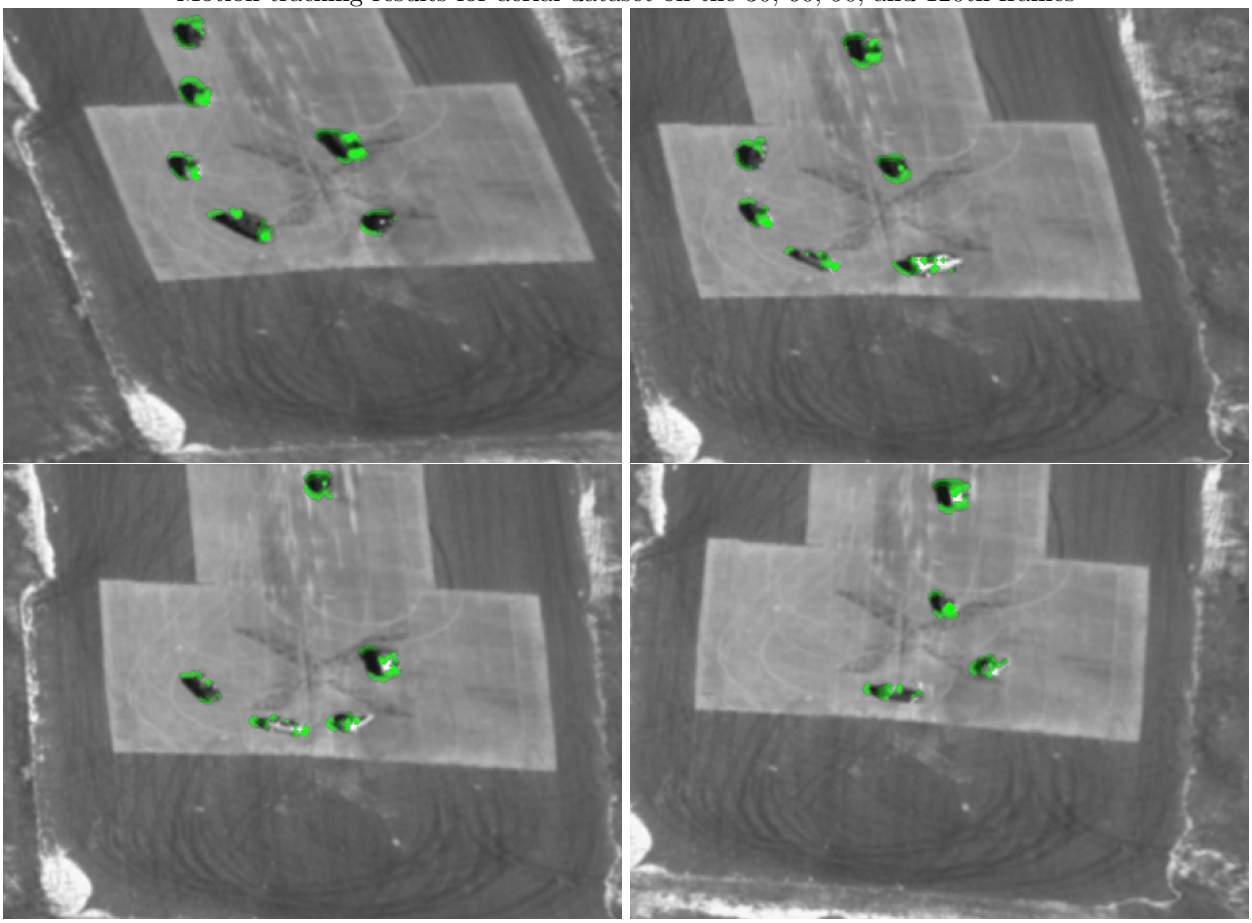
Resource https://www.ri.cmu.edu/pub_files/pub3/baker_simon_2004_1/baker_simon_2004_1.pdf

Result Images

Motion tracking results for ant dataset on the 30, 60, 90, and 120th frames



Motion tracking results for aerial dataset on the 30, 60, 90, and 120th frames



Code for inverseCompositionAffine.py

To run inverse composition affine tracking

```
python testAntSequence.py --num_iters 1000 --tolerance 0.1 --threshold 0.0001 --inverse True
python testAerialSequence.py --num_iters 1000 --tolerance 0.06 --threshold 0.0001 --inverse True
```

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

def InverseCompositionAffine(It, It1, threshold, num_iters):
    # 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 ###################
    # initialize p0
    p = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
    delta_M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])

    # interpolate template & current image
    # allign its matrix indices with plotting (x, y) coordinates by transposing It
    Ith, Itw = It.shape[0], It.shape[1]
    yarr = np.arange(0, Ith)
    xarr = np.arange(0, Itw)
    D = Ith*Itw # total number of points
    It_interp = RectBivariateSpline(xarr, yarr, It.T)

    # get gradients of It
    xgrid, ygrid = np.meshgrid(xarr, yarr)
    It_xgrad = It_interp.ev(xgrid, ygrid, dx = 1, dy = 0)
    It_ygrad = It_interp.ev(xgrid, ygrid, dx = 0, dy = 1)

    # construct matrix A from It (Inversed the roles of It and It1)
    # each element: multiply gradient w/ corresponding coordinate vals
    Idxx = (It_xgrad * xgrid).reshape(D, 1)
    Idxy = (It_xgrad * ygrid).reshape(D, 1)
    Idyx = (It_ygrad * xgrid).reshape(D, 1)
    Idyy = (It_ygrad * ygrid).reshape(D, 1)
    It_xgrad_flat = It_xgrad.reshape(D, 1)
    It_ygrad_flat = It_ygrad.reshape(D, 1)
    A = np.hstack((Idxx, Idxy, It_xgrad_flat, Idyx, Idyy, It_ygrad_flat))
    AT = A.T
    ATA_i = np.linalg.inv(np.matmul(AT, A))

    ### -- update p to minimize error, calculate A * delta_p = b
    delta_p = np.Inf
    it = 0
    while it < num_iters and np.linalg.norm(delta_p) > threshold:
      # find the points within the two images' common area
      It1_warped = affine_transform(It1.T, M).T
      # the common area mask
      mask = np.where(It1_warped > 0, 1, 0)

      # compute b
      b = np.where(mask, It1_warped - It, 0).reshape(D, 1)

      # comptue delta_p
      ATb = np.matmul(AT, b)
      delta_p = np.matmul(ATA_i, ATb)

      # update delta_M using delta_p
      delta_p = delta_p.reshape((6, ))
      delta_M = np.array([[1 + delta_p[0], delta_p[1], delta_p[2]], [delta_p[3], 1 + delta_p[4],
                                                                    delta_p[5]], [0, 0, 1]])

      # update M with M and delta_M using composite method
      M = np.matmul(M, np.linalg.inv(delta_M))
      it += 1

    M = M[:2]
    return M
```

3.2 Improved efficiency

Why Inverse Composition is more efficient than classical Lucas Kanade

Time inefficiency of classical Lacas Kanade method

Lucas Kanade approach solves for a p by updating it with a delta p using gradient descent. In classical Lucas Kanade approach, to solve for delta p, the Hessian matrix is calculated in each iteration from steepest gradient on the current image. This involves finding derivatives on both x and y coordiniates and multiplying them with each pixel's coordinate values. Because an image usually has a large number of pixels. this Hessian matrix can be huge. Calculating it in every iteration is inefficient. However, we cannot just compute Hessian matrix once and reuse it, because it is a function of p in this case, so it is different for every pair of consecutive frames.

Improved time efficiency of inverse composition method

The Inverse Composition approach switches the roles of current and template images. Instead of calculating the gradient of the warped image, it calculates the gradient of the template before the while loop starts. This calculation is only done once. This Hessian matrix does not depend on p, as it is calculated on the template, and this is why we can safely take it outside the while loop. The only other modification is that due to the inversed roles of template and current image, we update M matrix using the inverse of delta M.

Improved accuracy of mask area in inverse composition method

In Inverse Composition, update to the M matrix is more accurate than in regular Lucas Kanade. In the affine case, the pixel transformations between two images are not linear, and the delta p calculated represents an affine transformation. Therefore, update to M is more accurate when we solve for an incremental warp on M rather than a simple linear update. The compositional M matrix is a bilinear combination of W(x, p) and W(x, delta p). This is why the compositional method is able to capture more detailed movements, such as tiny movements of an ant's leg, and provide a more accurate mask.