Home assignment 1 – lior.kotlar

Answers

Task 5:

Upon visual inspection of the animation, it is evident that the tree appears to move faster than the flowers. This aligns with the expected result, since the tree is closer to the camera, and hence its image displacement due to camera motion is greater (a standard depth-induced parallax effect).

To quantify the motion, I ran my implementation of the Lucas-Kanade algorithm on two selected subimages:

- One containing only flowers.
- One containing only the tree.

I used a regularization parameter of λ = 0 (i.e., no Bayesian prior), and applied the Full_LK_alg function on both subimages with 10 iterations:

The tree calculated motion vector is: [-8.725861600185766, -0.12097059332026297]

The flowers calculated motion vector is: [-5.054193562299164, -0.24378273666737688]

Task 6:

For $v = (1,0), \lambda = 0.1$ the calculated motion vector is: [5.983414743370867, -3.627780565463102e-16]

For $v = (1,1), \lambda = 0.1$ the calculated motion vector is: [5.983184694257898, -3.602728763986361e-16]

For $v = (1,-1), \lambda = 0.1$ the calculated motion vector is: [6.12731120561642, -3.6027287639863575e-16]

The regularized Lucas-Kanade algorithm fails to recover the vertical component of motion in this case, because the image contains only a single vertical edge, which cannot constrain vertical motion. The estimated velocities do not agree with the correct motion vectors though they do match in the horizontal direction.

This result highlights a well-known limitation of gradient-based optical flow: motion perpendicular to an edge cannot be recovered without multiple orientations — a phenomenon known as the aperture problem.

The a matrix calculated in the first iteration of the algorithm is:

```
[[ 3.23396123e+00 -1.21174894e-16]
 [-1.21174894e-16  1.00000000e-01]]
```

Which is invertible.

We visualized the frame pairs using the provided $mymovie.m$ function. In all three cases, our percept matches the output of the algorithm. The edge appears to shift purely to the right, regardless of its vertical motion component. This makes sense: since the edge is vertical, the visual system cannot distinguish between pure horizontal and diagonal motion.

Task 7:

We ran our Lucas-Kanade algorithm (with λ=0.01\lambda = 0.01λ=0.01) on both rhombi with contrast levels ranging from 1.0 to 0.1, and tested multiple iteration counts.

Observations

- At high contrast (1.0), the algorithm produced motion vectors close to the IOC prediction.
- As contrast was lowered, the output drifted toward the VA prediction, especially for the thin rhombus.
- The number of iterations affected convergence speed, but not the general trend.

Interpretation

At low contrast, image gradients become weaker, making the aperture problem more ambiguous. This causes the algorithm (like human perception) to rely more on the VA heuristic due to insufficient local motion constraints.

This goes hand in hand with what we learned in class and with the way our perception works. The lower the contrast, the farther from the ioc the precepted velocity.

Code:

```
import numpy as np
import cv2
import sys
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from scipy import signal
from scipy.interpolate import RegularGridInterpolator
from scipy.ndimage import convolve
import math

PATCHED = True
X_DERIVATIVE_IDX = 0
Y_DERIVATIVE_IDX = 1
PATCH_SIZE = 25
FRAME1_FILE_NAME = 'data/frame1.jpg'
FRAME2_FILE_NAME = 'data/frame2.jpg'
FRAME_SKIP = 3
EDGE_FRAMES = True
RHOMBUS = False
LOAD_FLOWER_FRAMES = True
```

```
FLOWER_FRAME1 = 'data/Ex2-data/Ex2-data/flower-i1.tif'
FLOWER_FRAME2 = 'data/Ex2-data/Ex2-data/flower-i2.tif'
FX2 = 0
FY2 = 1
FYFX = 2
FXFT = 3
FYFT = 4


def soft_thresh(x, t, s=5):
    return 1 / (1 + np.exp(-s * (x - t)))

def rhombus_image(isize, x_shift, fat_flag, contrast):
    # Set angles in degrees based on fat_flag
    theta1 = 0 if fat_flag == 1 else 30
    theta2 = 45

    theta1 = np.deg2rad(theta1)
    theta2 = np.deg2rad(theta2)

    r = isize / 6

    x, y = np.meshgrid(np.arange(1, isize + 1), np.arange(1, isize + 1))
    x = x - isize / 2 - x_shift
    y = y - isize / 2

    # Warp coordinates
    xW = np.sin(theta1) * x + np.cos(theta1) * y
    yW = np.sin(theta2) * x + np.cos(theta2) * y

    M = np.array([
        [np.cos(theta1), np.sin(theta1)],
        [np.cos(theta2), np.sin(theta2)]
    ])
    detM = np.abs(np.linalg.det(M))

    val = np.maximum(np.abs(xW), np.abs(yW))
    im = 1 - soft_thresh(val, r * detM, s=1)
    return im * contrast

def rhombus_movie(fat_flag, contrast, vx):
    im1 = rhombus_image(128, 0, fat_flag, contrast)
    im2 = rhombus_image(128, vx, fat_flag, contrast)
    return im1, im2


def edge_frames(vx, vy):
```

```python
    xx, yy = np.meshgrid(np.arange(1, 129), np.arange(1, 129))
    im1 = (xx < 64).astype(float)
    xx_shifted = xx - vx
    yy_shifted = yy - vy
    im2 = (xx_shifted < 64).astype(float)

    return im1, im2


def get_patch(image, patch_center):
    half_window = PATCH_SIZE // 2
    template = image[patch_center[1] - half_window:patch_center[1] + half_window + 1,
             patch_center[0] - half_window:patch_center[0] + half_window + 1]
    return template


def compute_error_image(i_target, i_shifted):
    error = i_target - i_shifted
    return error


def warp_image(img, tx, ty, mask):
    m = np.array([[1, 0, tx],
                  [0, 1, ty]], dtype=np.float32)

    height, width = img.shape[:2]
    shifted = cv2.warpAffine(img, m, (width, height))
    warp_mask = cv2.warpAffine(mask, m, (width, height))

    return shifted, warp_mask


def warp_matlab(Im, v):
    row_num, col_num = Im.shape

    # Define grid in MATLAB-style coordinates (1-based indexing and flipped rows)
    x = np.arange(1, col_num + 1)
    y = np.arange(row_num, 0, -1)  # Flip vertically to match MATLAB's flipud

    # Create the interpolator
    interpolator = RegularGridInterpolator((y, x), Im, bounds_error=False, fill_value=np.nan)

    # Create meshgrid of coordinates
    xx, yy = np.meshgrid(x, y)

    # Shift coordinates
    coords = np.stack([(yy + v[1]).ravel(), (xx + v[0]).ravel()], axis=-1)
```

```
    # Interpolate
    Iw = interpolator(coords).reshape(Im.shape).astype(np.uint8)

    # Create warp mask (1 where not NaN, 0 where NaN)
    warpMask = ~np.isnan(Iw)
    Iw[~warpMask] = 0
    warpMask = warpMask.astype(np.uint8)

    return Iw, warpMask


def show_image(img, title):
    cv2.imshow(title, img)
    cv2.waitKey(0)


def show_image_plt(img):
    plt.imshow(img, cmap='gray')


def get_time_derivative(img1, img2):
    time_kernel = np.array([[0.25, 0.25],
                            [0.25, 0.25]], dtype=np.float32)

    img1_smoothed = cv2.filter2D(img1, -1, time_kernel)
    img2_smoothed = cv2.filter2D(img2, -1, time_kernel)

    temporal_derivative = img2_smoothed - img1_smoothed

    return temporal_derivative


def get_time_derivative_plt(img1, img2):
    filter_t = np.array([[0.25, 0.25],
                         [0.25, 0.25]])

    it1 = signal.convolve2d(img1, filter_t, mode='same')
    it2 = signal.convolve2d(img2, filter_t, mode='same')
    it = it2 - it1
    return it


def single_image_derivative(img):
    derivative_finder_x = np.array([[0.25, -0.25],
                                    [0.25, -0.25]], dtype=np.float32)
    derivative_finder_y = derivative_finder_x.T
    ix = cv2.filter2D(src=img, ddepth=-1, kernel=derivative_finder_x)
    iy = cv2.filter2D(src=img, ddepth=-1, kernel=derivative_finder_y)
```

```python
    return ix, iy


def single_image_derivative_plt(img):
    derivative_finder_x = np.array([[0.25, -0.25],
                                    [0.25, -0.25]], dtype=np.float32)
    derivative_finder_y = derivative_finder_x.T
    ix = signal.convolve2d(img, derivative_finder_x, mode='same')
    iy = signal.convolve2d(img, derivative_finder_y, mode='same')
    return ix, iy


def image_derivatives(img1, img2):
    img1_derivatives = single_image_derivative(img1)
    img2_derivatives = single_image_derivative(img2)
    ix = img1_derivatives[0] + img2_derivatives[0]
    iy = img1_derivatives[1] + img2_derivatives[1]
    time_derivative = get_time_derivative(img1, img2)
    return ix, iy, time_derivative


def load_images(path1, path2, grayscale=True):
    image1 = cv2.imread(path1)
    image2 = cv2.imread(path2)
    if grayscale:
        image1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
        image2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
    return image1, image2


def read_2_frames(cap, skip):
    ret1, frame1 = cap.read()
    ret_ = None
    if ret1:
        for _ in range(skip):
            ret_, frame_ = cap.read()
            if ret_ is None:
                break
    if ret_ or skip == 0:
        ret2, frame2 = cap.read()
        if ret2:
            frame1 = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)
            frame2 = cv2.cvtColor(frame2, cv2.COLOR_BGR2GRAY)
            return frame1, frame2
    print('no more frames')
    return None, None
```

```
def extract_two_frames(video_file_path, skip=FRAME_SKIP, save=True):
  cap = cv2.VideoCapture(video_file_path)
  frame1, frame2 = read_2_frames(cap, skip)
  if frame1 is None or frame2 is None:
    print('cant read two frames')
    exit(1)
  cv2.imwrite(FRAME1_FILE_NAME, frame1)
  cv2.imwrite(FRAME2_FILE_NAME, frame2)
  return frame1, frame2


def get_patch_center(image):
  coords = []

  def mouse_callback(event, x, y, flags, param):
    if event == cv2.EVENT_LBUTTONDOWN:
      coords.append((x, y))
      # cv2.circle(image, (x, y), radius=5, color=(0, 0, 255), thickness=-1)
      print(f"Clicked at: ({x}, {y})")
      cv2.destroyAllWindows()

  # Show image
  window_name = 'Click to select patch center'
  cv2.namedWindow(window_name)
  cv2.setMouseCallback(window_name, mouse_callback)

  while True:
    cv2.imshow(window_name, image)
    key = cv2.waitKey(1) & 0xFF
    if len(coords) > 0 or key == 27:  # ESC key to exit without clicking
      break

  cv2.destroyAllWindows()
  return coords[0] if coords else None


def blur_downsample(image, kernel_size=5, sigma=0.8):

  g1d = np.array([
        [0.0112972493575865, 0.0149145471310277, 0.0176194555564150,
0.0186260153162562, 0.0176194555564150, 0.0149145471310277, 0.0112972493575865],
        [0.0149145471310277, 0.0196900775651435, 0.0232610781617145,
0.0245899310977849, 0.0232610781617145, 0.0196900775651435, 0.0149145471310277],
        [0.0176194555564150, 0.0232610781617145, 0.0274797169008232,
0.0290495711540169, 0.0274797169008232, 0.0232610781617145, 0.0176194555564150],
        [0.0186260153162562, 0.0245899310977849, 0.0290495711540169,
0.0307091076402979, 0.0290495711540169, 0.0245899310977849, 0.0186260153162562],
        [0.0176194555564150, 0.0232610781617145, 0.0274797169008232,
```

```
0.0290495711540169, 0.0274797169008232, 0.0232610781617145, 0.0176194555564150],
            [0.0149145471310277, 0.0196900775651435, 0.0232610781617145,
0.0245899310977849, 0.0232610781617145, 0.0196900775651435, 0.0149145471310277],
            [0.0112972493575865, 0.0149145471310277, 0.0176194555564150,
0.0186260153162562, 0.0176194555564150, 0.0149145471310277, 0.0112972493575865]
            ])

  blurred = cv2.filter2D(image, -1, g1d)
  downsampled = blurred[::2, ::2] if image.ndim == 2 else blurred[::2, ::2, :]

  return downsampled


def get_2nd_derivative(fx, fy, ft):
  fx_2 = fx**2
  fy_2 = fy**2
  fx_fy = fx*fy
  fx_ft = fx*ft
  fy_ft = fy*ft
  return fx_2, fy_2, fx_fy, fx_ft, fy_ft


def analyze_patch(second_derivatives, lmda):
  a11 = second_derivatives[FX2].sum()
  a12 = second_derivatives[FYFX].sum()
  a22 = second_derivatives[FY2].sum()
  b1 = -1 * second_derivatives[FXFT].sum()
  b2 = -1 * second_derivatives[FYFT].sum()
  a_matrix = np.array([[a11, a12],
             [a12, a22]])
  b_matrix = np.array([b1, b2])
  b_matrix_t = b_matrix.T

  a_matrix_regularized = a_matrix + lmda * np.eye(2)
  print(f'a_matrix_regularized\n'
     f'{a_matrix_regularized}\n')
  a_m_r_inv = np.linalg.pinv(a_matrix_regularized)
  uv = a_m_r_inv @ b_matrix_t

  print(f'uv: {uv}')
  return uv


def lk_alg(i1, i2, mask, lmda, v_initial, num_iterations):
  accumulated_vector = v_initial
  previous_iteration_guess = [0, 0]
  i1x, i1y = single_image_derivative_plt(i1)
  height, width = i1.shape
```

```
    mask_prev = mask
    for iter in range(num_iterations):
        print(f'iteration: {iter}')
        x_shift, y_shift = accumulated_vector
        print(f'x_shift: {x_shift}, y_shift: {y_shift}')
        warped_image, warp_mask = warp_image(i2, x_shift, y_shift, mask_prev)
        # warped_image, warp_mask = warp_matlab(i2, (x_shift, y_shift))
        new_mask = mask_prev * warp_mask
        it = get_time_derivative_plt(i1, warped_image)
        i2x, i2y = single_image_derivative_plt(warped_image)
        ix = i1x + i2x
        iy = i1y + i2y
        second_derivatives = get_2nd_derivative(ix, iy, it)
        second_derivatives_masked = [der*new_mask for der in second_derivatives]
        ux, uy = analyze_patch(second_derivatives_masked, lmda)

        print(f'u = ({ux}, {uy})')

        accumulated_vector[0] += ux
        accumulated_vector[1] += uy

        print(f'accumulated vector: {accumulated_vector}')

        mask_prev = new_mask

    return accumulated_vector


def get_mask(patch_center, image_height, image_width):
    x_center, y_center = patch_center
    mask = np.zeros((image_height, image_width))
    half_window = PATCH_SIZE//2
    mask[y_center - half_window:y_center + half_window + 1,
            x_center - half_window:x_center + half_window + 1] = 1
    return mask

def full_lk_alg(i1, i2, lmda, num_iterations=10):
    d = [0, 0]
    image_height, image_width = i1.shape
    if not RHOMBUS:
        patch_center = get_patch_center(i1)
        mask = get_mask(patch_center, image_height, image_width)
    else:
        mask = np.ones((image_height, image_width))
    i1_downsampled = blur_downsample(i1, kernel_size=5)
    i2_downsampled = blur_downsample(i2, kernel_size=5)
    mask_downsampled = blur_downsample(mask, kernel_size=5)
```

```
    initial_guess = lk_alg(i1_downsampled,
                           i2_downsampled,
                           mask=mask_downsampled,
                           lmda=lmda,
                           v_initial=[0, 0],
                           num_iterations=1)
    initial_guess = [2*p for p in initial_guess]

    if RHOMBUS:
        return initial_guess
    print(f'initial guess: {initial_guess}')

    final_vector = lk_alg(i1, i2,
                          mask=mask,
                          lmda=lmda,
                          v_initial=initial_guess,
                          num_iterations=num_iterations)

    print(f'final vector: {final_vector}')

    return final_vector


def load_flower_frames():
    frame1, frame2 = load_images(FLOWER_FRAME1, FLOWER_FRAME2)
    return frame1, frame2


def find_va(theta1, theta2, movement=(1,0)):
    magnitude1 = movement[0]*np.cos(np.deg2rad(90-theta1))
    theta1_normal = 270 + theta1

    magnitude2 = movement[1]*np.sin(np.deg2rad(90-theta2))
    theta2_normal = 270 + theta2

    theta1_normal_rad, theta2_normal_rad = np.deg2rad(theta1_normal),
np.deg2rad(theta2_normal)
    x1_normal = magnitude1*np.cos(theta1_normal_rad)
    y1_normal = magnitude2*np.sin(theta1_normal_rad)
    x2_normal = magnitude2*np.cos(theta2_normal_rad)
    y2_normal = magnitude2*np.sin(theta2_normal_rad)

    va = [np.mean([x1_normal, x2_normal]), np.mean([y1_normal, y2_normal])]
    return va


def draw_ioc(frame1, frame2):
    ioc_vec = full_lk_alg(frame1, frame2, 0.01, 1)
```

```
    return ioc_vec


def draw_vector(image, vector):
    height, width = image.shape
    origin = (width // 2, height // 2)
    end_point = (
        int(origin[0] + vector[0]),
        int(origin[1] - vector[1])
    )
    cv2.arrowedLine(image, origin, end_point, (255,0,0), thickness=2, tipLength=0.1)
    return image


def main():

    mode = sys.argv[1]
    lmda = 0.0
    if RHOMBUS:
        frame1, frame2 = rhombus_movie(1, 0.5, 1)
        show_image(frame1, 'frame1')
        show_image(frame2, 'frame2')
        ioc_vec = draw_ioc(frame1, frame2)
        print(ioc_vec)
        exit()
    elif EDGE_FRAMES:
        print('edge frames')
        frame1, frame2 = edge_frames(1, -1)
        lmda = 0.1
    elif LOAD_FLOWER_FRAMES:
        frame1, frame2 = load_flower_frames()
    elif mode == 'v':
        print("mode v")
        frame1, frame2 = extract_two_frames(sys.argv[2])
    else:
        frame1, frame2 = load_images(FRAME1_FILE_NAME, FRAME2_FILE_NAME)

    motion_vector = full_lk_alg(frame1, frame2, lmda=lmda)
    print(list([float(p) for p in motion_vector]))


if __name__ == '__main__':
    main()
```