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 the calculated motion vector is: [5.983414743370867, -3.627780565463102e-16]

For the calculated motion vector is: [5.983184694257898, -3.602728763986361e-16]

For 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:

A number on a black background

AI-generated content may be incorrect.

Which is invertible.

We visualized the frame pairs using the provided 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):  
  
 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)  
 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()