Amirkabir University of Technology
(Tehran Polytechnic)
Department of Computer Engineering

Computer Vision

Assignment V

Soroush Mahdi
99131050

supervised by
Dr. Reza Safabakhsh

December 24, 2021

Contents

1 Shi-Tomasi Corner Detectors? 1

2 Object Tracking with Lukas-Kanade algorithm 1

3 Gunner Farneback VS Lukas-Kanade 3

4 Object Tracking with Gunner Farneback 4

5 SIFT VS Optical Flow(Object Tracking) 5

1 Shi-Tomasi Corner Detectors?

the basic intuition is that corners can be detected by looking for significant change in all direction. We consider a small window on the image then scan the whole image, looking for corners. Shifting this small window in any direction would result in a large change in appearance, if that particular window happens to be located on a corner.

in opencv we do this using cv2.goodFeaturesToTrack() function that finds N strongest corners in the image by Shi-Tomasi method. we can use this function like this:

```
corners = cv2.goodFeaturesToTrack(gray_img, mask=None, maxCorners=100, qualityLevel
    =0.3, minDistance=7, blockSize=7, useHarrisDetector=False, k=0.04)
```

Parameters:

- first parameter is the input image that we want to find corners in it and must be grayscale.
- mask: Optional region of interest. If the image is not empty, it specifies the region in which the corners are detected.
- maxCorners: Maximum number of corners to return.
- qualityLevel: Parameter characterizing the minimal accepted quality of image corners. The parameter value is multiplied by the best corner quality measure, which is the minimal eigenvalue. The corners with the quality measure less than the product are rejected. For example, if the best corner has the quality measure = 1500, and the qualityLevel=0.01, then all the corners with the quality measure less than 15 are rejected. this parameter should be a value between 0-1. higher values of this parameter will result in less number of detected corners.
- minDistance: Minimum possible Euclidean distance between the returned corners. higher values of this parameter will result in less number of detected corners.
- blockSize: Size of an average block for computing a derivative covariation matrix over each pixel neighborhood.
- useHarrisDetector: Parameter indicating whether to use a Harris detector instead of Shi-Tomasi method.
- k: Free parameter of the Harris detector.

2 Object Tracking with Lukas-Kanade algorithm

here is the first part of the code for this section:

```
# params for ShiTomasi corner detection function
feature_params = dict( maxCorners = 100,
                       qualityLevel = 0.3,
                       minDistance = 7,
                       blockSize = 7 )
# a list for keeping last 3 frames
images = []
# Parameters for lucas kanade optical flow function
lk_params = dict( winSize  = (15, 15),
                  maxLevel = 2,
                  criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10,
    0.03))
```

Listing 1: initializing parameters

here I defined some parameters that i will use in the code.

in the next part i will find corners in the first frame using ShiTomasi corner detection function:

```
# Taking first frame and find corners in it
ret, first_frame = cam.read()
old_gray = cv2.cvtColor(first_frame, cv2.COLOR_BGR2GRAY)
images.append(old_gray)
p0 = cv2.goodFeaturesToTrack(images[0], mask = None, **feature_params)
```

Listing 2: corner detection in the first frame

then i performed Lukas-Kanade algorithm in a loop on the last 3 frames:

```
while True:

    #reading frames
    ret, QueryImgBGR=cam.read()
    QueryImg=cv2.cvtColor(QueryImgBGR,cv2.COLOR_BGR2GRAY) #turning frame to gray

    if ret:
        images.append(QueryImg)

    if len(images) > 2: #skiping till we have seen more than 2 frames

        # calculating optical flows
        p1, st1, er1 = cv2.calcOpticalFlowPyrLK(images[0], images[1], p0, None, **
    lk_params)
        p2, st2, er2 = cv2.calcOpticalFlowPyrLK(images[1], images[2], p1, None, **
    lk_params)

        # Selecting good points
        if p1 is not None:
            good_new1 = p1[st1==1]
            good_old1 = p0[st1==1]

        if p2 is not None:
            good_new2 = p2[st2==1]
            good_old2 = p1[st2==1]

        # draw the tracks
        for (new, old) in zip(good_new1, good_old1):
            a, b = new.ravel()
            c, d = old.ravel()
            QueryImgBGR = cv2.line(QueryImgBGR, (int(a), int(b)), (int(c), int(d)),
    (0,0,255), 2)
            QueryImgBGR = cv2.circle(QueryImgBGR, (int(a), int(b)), 5,(0,0,255) , -1)

        for (new, old) in zip(good_new2, good_old2):
            a, b = new.ravel()
            c, d = old.ravel()
            QueryImgBGR = cv2.line(QueryImgBGR, (int(a), int(b)), (int(c), int(d)),
    (255,0,0), 2)
            QueryImgBGR = cv2.circle(QueryImgBGR, (int(a), int(b)), 5, (255,0,0), -1)

        images.pop(0)

        p0 = good_new1.reshape(-1, 1, 2)
        p1 = good_new2.reshape(-1, 1, 2)
```

Listing 3: Lukas-Kanade algoritm

this section is the main part of the code, first i performed Lukas-Kanade algorithm between -2 and -1 frames and then between -1 and last frames. then i selected good points in each of the returned results. after that, i drew results of first performed algorithm between frames -2 and -1 in red and between -1 and last frames in blue.

at the end of the loop i updated values of images, p0 and p1 variables.

3 Gunner Farneback VS Lukas-Kanade

we can say that there are two types of optical flow algorithms, sparse techniques and dense techniques. Gunnar Farneback Optical Flow is a dense technique and Lukas-Kanade is a sparse technique.

Sparse optical flow gives the flow vectors of some "interesting features" (say few pixels depicting the edges or corners of an object) within the frame while Dense optical flow, which gives the flow vectors of the entire frame (all pixels) - up to one flow vector per pixel.

Dense techniques are slower but can be more accurate.so we can say Gunnar Farneback is slower but more accurate.

in the Lukas-Kanade algorithm, we have to specify some points of interest for the algorithm to follow. so for example, if we have a known object that we want to follow, maybe it's better to use this algorithm. but the gunner Farneback algorithm will show all movements in the frame, so if we don't have a specific object, and we want to track all objects in the frame this algorithm is better.

4 Object Tracking with Gunner Farneback

first I will explain the function that i will use for drawing flow vectors in the image.

```
def draw_flow(img,flow,step=32 , color = (0,0,255)):
    #a function for drawing flow vectors on the image
    h,w = img.shape[:2]
    y, x = np.mgrid[step/2:h:step, step/2:w:step].reshape(2,-1).astype(int)
    fx,fy = flow[y,x].T

    # create line endpoints
    lines = np.vstack([x,y,x+fx,y+fy]).T.reshape(-1,2,2)
    lines = np.int32(lines)

    for (x1,y1),(x2,y2) in lines:
        #flow vectors are bigger than a treshold
        #then we will draw them
        if distance.euclidean((x1,y1),(x2,y2)) > 5:
            img = cv2.line(img,(x1,y1),(x2,y2),color,2)
            img = cv2.circle(img,(x1,y1),1,color, 2)
    return img
```

Listing 4: draw flow function

in this function first I created a grid on the image. I will only draw flow vectors on the points in the grid. then I just selected flow vectors that are bigger than a threshold to draw.

the main part of code in this section is this loop:

```
while True:
    #reading frames
    ret, QueryImgBGR=cam.read()
    QueryImg=cv2.cvtColor(QueryImgBGR,cv2.COLOR_BGR2GRAY) #turning frame to gray

    if ret:
        images.append(QueryImg)

    if len(images) > 2: #skiping till we have seen more than 2 frames

        # calculate optical flow
        flow1 = cv2.calcOpticalFlowFarneback(images[0], images[1], None, 0.5, 3, 15,
    3, 5, 1.2, 0)
        flow2 = cv2.calcOpticalFlowFarneback(images[1], images[2], None, 0.5, 3, 15,
    3, 5, 1.2, 0)
        #drawing flows
        draw_flow(QueryImgBGR, flow1 , color = (0,0,255))
        draw_flow(QueryImgBGR, flow2 , color = (255,0,0))

        cv2.imshow('frame', QueryImgBGR)
        out.write(QueryImgBGR)
        images.pop(0)
```

Listing 5: applying Gunner Farneback

in this loop, I performed gunner farneback two times, first time on -2 and -1 frames and then on -1 and last frame. then I drew result of first time in red and result of second time in blue.

5 SIFT VS Optical Flow(Object Tracking)

Feature matching uses the feature descriptors to match features with one another (usually) using a nearest neighbor search in the feature descriptor space. The basic idea is you have descriptor vectors, and the same feature in two images should be near each other in the descriptor space, so you just match that way.

Most feature matching methods are scale and rotation invariant and are robust for changes in illuminations (e.g caused by shadow or different contrast). The disadvantage of Feature Matching methods is the difficulty of defining where the feature matches are spawn and that the feature pair (which in a image sequence are motion vectors) are in general very sparse. In addition the subpixel accuracy of matching approaches are very limited as most detector are fine-graded to integer positions.

the main advantage of feature matching approaches is that they can compute very large motions/ displacements.

Optical flow methods in contrast rely on the minimization of the brightness constancy and additional constrain e.g. smoothness etc. Thus they derive motion vector based on spatial and temporal image gradients of a sequence of consecutive frames. The main challenges in the estimation of motion with optical flow vectors are large motions, occlusion, strong illumination changes and changes of the appearance of the objects and mostly the low runtime. However optical flow methods can be highly accurate and compute dense motion fields which respect to shared motion boundaries of the objects in a scene.

Optical flow algorithms do not look at a descriptor space, and instead, looks at pixel patches around features and tries to match those patches instead. sparse optical flow just does dense optical flow but on small patches of the image around feature points. Thus optical flow assumes brightness constancy, that is, that pixel brightness doesn't change between frames. Also, since you're looking around neighboring pixels, you need to make the assumption that neighboring points to your features move similarly to your feature. Finally, since it's using a dense flow algorithm on small patches, the points where they move cannot be very far in the image from the original feature location. If they are, then the pyramid-resolution approach is recommended, where you scale down the image before you do this so that what once was a 16 pixel translation is now a 2 pixel translation, and then you can scale up with the found transformation as your prior.

So feature matching algorithms are all-in-all far better when it comes to using templates where the scale is not exactly the same, or if there's a perspective difference in the image and template, or if the transformations are large. However, your matches are only as good as your feature detector is exact. On optical flow algorithms, as long as it's looking in the right spot, the transformations can be really, really precise. They're both computationally expensive a bit; optical flow algorithms being an iterative approach makes them expensive (and although you'd think the pyramid approach can eat up more costs by running on more images, it can actually make it faster in some cases to reach the desired accuracy), and nearest neighbor searches are also expensive.

Which one to use definitely depends on the project.