HOMEWORK 2: LUCAS-KANADE TRACKING

16-720A Computer Vision (Spring 2023)
https://canvas.cmu.edu/courses/32966

OUT: Feb 13th, 2023
DUE: March 3rd, 2023 11:59 PM
Instructor: Deva Ramanan
TAs: Kangle Deng, Vidhi Jain, Xiaofeng Guo, Chung Hee Kim, Ingrid Navarro

Instructions

- Please refer to the course logistics page for information on the **Collaboration Policy** and **Late Submission Policy**.
- **Submitting your work:** There will be two submission slots for this homework on **Gradescope**: Written and Programming.
  - **Write-up.** For written problems such as short answer, multiple choice, derivations, proofs, or plots, we will be using the written submission slot. Please use this provided template. We don't accept handwritten submissions. Each answer should be completed in the boxes provided below the question. You are allowed to adjust the size of these boxes, but make sure to link your answer to each question when submitting to Gradescope. Otherwise, your submission will not be graded.
  - **Code.** You are also required to upload the code that you wrote to solve this homework, to the Programming submission slot. Your code may be run by TAs so please make sure it is in a working state. Also, make sure to use appropriate names for your variables and to add comments and explanations to your code for the TAs to understand it better. The assignment must be completed using Python 3 in Jupyter. We recommend setting up a conda environment, but you are free to set up your environment however you like.
  - Regrade requests can be made after the homework grades are released, however this gives the TA the opportunity to regrade your entire paper, meaning if additional mistakes are found then points will be deducted.
- **Start early!** This homework may take a long time to complete.
- **Verify your implementation as you proceed.** If you don't verify that your implementation is correct on toy examples, you will risk having a huge mess when you put everything together.
- **Q&A.** If you have any questions or need clarifications, please post in Slack or visit the TAs during office hours. Additionally, we provide a **FAQ** (section 6) with questions from previous semesters. Make sure you read it prior to starting your implementations.

Overview

Lucas-Kanade (LK) is a widely known vision-based method for tracking features in image sequences. In a nutshell, it uses the notion of optical flow for estimating the motion of pixels in subsequent images. This homework explores the core aspects for implementing the LK tracking method and efficiency improvements.

What you'll be doing:

This homework is worth 100 points + 20 extra credit points, and consists of four parts:

1. In section 1, you will implement a Lucas-Kanade (LK) tracker for **pure translation with a single template**. You will explore two methods for performing template updates: a naive update, and an update with drift correction. You will test your implementation on two sequences `carseq.npy` and `girlseq.npy` which we provide in the `data` folder. This section is worth 50 points.
2. In section 2, you will modify the tracker to account for **affine motion**, and you will then implement a motion subtraction method to track moving pixels on a scene. You will test your implementation on two sequences `antseq.npy` and `aerialseq.npy` which we provide in the `data` folder. This section is worth 35 points.
3. In section 3, you will implement **efficient** tracking via inverse composition. You will also test your implementation `antseq.npy` and `aerialseq.npy`. This section is worth 15 points.
4. For **extra credit**, in section 4 you're asked to run your implementation on a video of your choice. This section is worth 20 extra points.

Resources:

In addition to the course materials, you may find the following references useful;

- Simon Baker, et al. *Lucas-Kanade 20 Years On: A Unifying Framework: Part 1*. CMU-RI-TR-02-16, Robotics Institute, Carnegie Mellon University, 2002. [link]
- Simon Baker, et al. *Lucas-Kanade 20 Years On: A Unifying Framework: Part 2*. CMU-RI-TR-03-35, Robotics Institute, Carnegie Mellon University, 2003. [link]

1 Lucas-Kanade Tracking (50 total points)

In this section, you will implement a simple Lucas-Kanade (LK) tracker with a single template.

Coding questions for **part 1** must be implemented in `LucasKanade.ipynb`. We provide you with starter code for each question, as well as default parameters. To test your tracker, you will use `carseq.npy` (top row in fig. 1.1) and `girlseq.npy` (bottom row in fig. 1.1). You can find these files in the `data` folder.

Problem Formulation. Following the notation in [1], let us consider a tracking problem for a 2D scenario. We refer to $\mathcal{I}_{1:T}$ as a sequence of T frames, where $\mathcal{I}_t$ is the current frame and $\mathcal{I}_{t+1}$ is the subsequent one. We represent a **pure translation** warp function as,

$$\mathbf{x}^{'} = \mathcal{W}(\mathbf{x}; \mathbf{p}) = \mathbf{x} + \mathbf{p} \tag{1.1}$$

where $\mathbf{x} = [x, y]^T$ is a pixel coordinate and $\mathbf{p} = [p_x, p_y]^T$ is an offset.

Given a template, $\mathcal{T}_t$ in frame, $\mathcal{I}_t$ which contains D pixels, the Lucas-Kanade tracker aims to find an offset $\mathbf{p}$ by which to translate the template on $\mathcal{I}_{t+1}$, such that the squared difference between the pixels on those two regions is minimized,

$$\mathbf{p}^* = \underset{\mathbf{p}}{\operatorname{argmin}} \sum_{\mathbf{x} \in \mathcal{T}_\mathbf{t}} ||\mathcal{I}_{\mathbf{t+1}}(\mathcal{W}(\mathbf{x}; \mathbf{p})) - \mathcal{T}_\mathbf{t}(\mathbf{x})||_\mathbf{2}^\mathbf{2} = \left|\left| \begin{bmatrix} \mathcal{I}_{t+1}(\mathcal{W}(\mathbf{x}_1; \mathbf{p})) \\ \vdots \\ \mathcal{I}_{t+1}(\mathcal{W}(\mathbf{x}_D; \mathbf{p})) \end{bmatrix} - \begin{bmatrix} \mathcal{T}_t(\mathbf{x}_1) \\ \vdots \\ \mathcal{T}_t(\mathbf{x}_D) \end{bmatrix} \right|\right|_\mathbf{2}^\mathbf{2} \tag{1.2}$$

In other words, we are trying to align two patches on subsequent frames by minimizing the difference between the two.

1.1 Theory Questions (5 points)

Starting with an initial guess for the offset, *e.g.* $\mathbf{p} = [0, 0]^T$, we can compute the optimal $\mathbf{p}^*$, iteratively. In each iteration, the objective function is locally linearized by the first-order Taylor expansion,

$$\mathcal{I}_{t+1}(\mathbf{x}' + \Delta\mathbf{p}) \approx \mathcal{I}_{t+1}(\mathbf{x}') + \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial \mathcal{W}(\mathbf{x}; \mathbf{p})}{\partial \mathbf{p}^T} \Delta\mathbf{p} \tag{1.3}$$

where $\Delta\mathbf{p} = [\delta p_x, \delta p_y]^T$, is the delta change of the offset, and $\frac{\partial \mathcal{I}(\mathbf{x}')}{\partial \mathbf{x}'^T}$ is a vector of the $x-$ and $y-$ image gradients at pixel coordinate $\mathbf{x}'$. We can then take this linearization into equation 1.2 into a vectorized form,

$$\arg\min_{\Delta\mathbf{p}} ||\mathbf{A}\Delta\mathbf{p} - \mathbf{b}||_2^2 \tag{1.4}$$

such that $\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p}$ at each iteration.

The questions that follow will be useful for your implementation. Please answer each of them in their corresponding box. The answers should be short.

Q1.1.1 What is $\frac{\partial \mathcal{W}(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T}$? (**Hint:** It should be a 2x2 matrix)

Answer for Q1.1.1

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

Q1.1.2 What is $\mathbf{A}$ and $\mathbf{b}$?

Answer for Q1.1.2

$\mathbf{A}$:

$$\mathbf{A} = \begin{bmatrix} \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}'_1)}{\partial \mathbf{x}'^T_1} I \\ \vdots \\ \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}'_D)}{\partial \mathbf{x}'^T_D} I \end{bmatrix}$$

$\mathbf{b}$:

$$\mathbf{b} = \begin{bmatrix} \mathcal{T}_t(\mathbf{x}_1) \\ \vdots \\ \mathcal{T}_t(\mathbf{x}_D) \end{bmatrix} - \begin{bmatrix} \mathcal{I}_{t+1}(\mathbf{x}'_1) \\ \vdots \\ \mathcal{I}_{t+1}(\mathbf{x}'_D) \end{bmatrix} = \begin{bmatrix} \mathcal{T}_t(\mathbf{x}_1) - \mathcal{I}_{t+1}(\mathbf{x}'_1) \\ \vdots \\ \mathcal{T}_t(\mathbf{x}_D) - \mathcal{I}_{t+1}(\mathbf{x}'_D) \end{bmatrix}$$

Q1.1.3 What conditions must $\mathbf{A}^T\mathbf{A}$ meet so that a unique solution to $\Delta\mathbf{p}$ can be found?

Answer for Q1.1.2

conditions: $\mathbf{A}^T\mathbf{A}$ is a full rank matrix ($r(\mathbf{A}^T\mathbf{A}) = n$). Also, $\mathbf{A}^T\mathbf{A}$ is invertible.

1.2 Lucas-Kanade (15 points)

Implement the function,

```
p = LucasKanade(It, It1, rect, threshold, num_iters, p0 = np.zeros(2))
```

where, `It` is the image frame, $\mathcal{I}_t$; `It1` is the image frame $\mathcal{I}_{t+1}$; `rect` is a 4-by-1 vector defined as $[x_1, y_1, x_2, y_2]^T$ that represents the corners of the rectangle comprising the template in $\mathcal{I}_t$. Here, $[x_1, y_1]^T$ is the top-left corner and $[x_2, y_2]^T$ is the bottom-right corner. The rectangle is inclusive, i.e., it includes all four corners; `p0` is the initial parameter guess $[\delta p_x, \delta p_y]^T$. Your optimization will be run for `num_iters`, or until $||\Delta\mathbf{p}||_2^2$ is below a `threshold`.

Your code must **iteratively** compute the optimal local motion ($\mathbf{p}^*$) from frame $\mathcal{I}_t$ to frame $\mathcal{I}_{t+1}$ that minimizes eq. (1.2). As you learned during lecture, at a high-level, this is done by following these steps:

1. Warp the template;
2. Build your linear system (**Q1.1.2**);
3. Run least-squares optimization (eq. (1.4));
4. Update the local motion.

Note: For this part, you will have to deal with fractional movement of the template by doing interpolations. To do so, you may find Scipy's function `RectBivariateSpline` useful. Read the documentation for `RectBivariateSpline`, as well as, for evaluating the spline `RectBivariateSpline.ev`.

Though we recommend using the aforesaid function, other similar functions for performing interpolations can be used as well. See the FAQ (section 6) for more details.

Include the code you wrote for this part in the box below:

Q1.2 Code for LucasKanade()

```
# We recommend using this function, but you can explore other methods as well
↪  (e.g., ndimage.shift).
from scipy.interpolate import RectBivariateSpline

# The function below could be useful as well :)
from numpy.linalg import lstsq

def LucasKanade(It, It1, rect, threshold, num_iters, p0=np.zeros(2)):
        # Initialize p to p0. Don't remove these lines.
        p = p0
        delta_p = np.ones((1,2))
        # --------------------
        # TODO: Add your LK implementation here:

        # infor: w and h
        w = int(np.round(rect[2] - rect[0] + 1))
        h = int(np.round(rect[3] - rect[1] + 1))
        # Use whole img to build mode:
        h_all=np.arange(0,It.shape[0])
        w_all=np.arange(0,It.shape[1])

        # build mode for interpolation of It, It1
        mode = RectBivariateSpline(h_all, w_all, It)
        mode1 = RectBivariateSpline(h_all, w_all, It1)

        # index for stop loop
        times = 0
        # values in template, rectangle is inclusive
        temp_x=np.arange(rect[0],rect[2]+1)[:w]
        temp_y=np.arange(rect[1],rect[3]+1)[:h]

        # get interpolation value of template from mode
        template = mode(temp_y,temp_x)
        # compare with threshold
        delta_threshold=delta_p[0][0]**2 + delta_p[0][1]**2

        # incase input get changed, make a copy
        rect1= np.copy(rect).astype(float)
        while ( delta_threshold>= threshold) and (times<num_iters):
                # 1. Warp the template;
                # update rect
                rect1[0] = p[0] + rect[0]
```

(see the rest in next page)

Q1.2 Code for LucasKanade()

```
            rect1[1] = p[1] + rect[1]
            rect1[2] = p[0] + rect[2]
            rect1[3] = p[1] + rect[3]

            # 2. Build your linear system (Q1.1.2);
            # get img of X' using interpolation
            x1=np.arange(rect1[0],rect1[2]+1)[:w]
            y1=np.arange(rect1[1],rect1[3]+1)[:h]

            # get new image values with interpolation
            new_img = mode1(y1,x1)

            # get b: n*1
            b_temp = template.reshape(-1,1)
            b_new_img=new_img.reshape(-1,1)
            b=b_temp-b_new_img
            # get A: n*2
            A=np.zeros((w*h,2))

            # gradient in x and y direction, derivative=1
            gradient_x=mode1(y1, x1, dy=1)
            gradient_x=gradient_x.reshape(1,-1)
            gradient_y=mode1(y1, x1, dx=1)
            gradient_y=gradient_y.reshape(1,-1)
            # append to A
            A[:,0]=gradient_x[0]
            A[:,1]=gradient_y[0]

            # 3. Run least-squares optimization (eq. (1.4));
            delta_p=np.linalg.lstsq(A, b, rcond=None)[0]

            # 4. Update the local motion.
            p[0]+=delta_p[0]
            p[1]+=delta_p[1]
            #print(p,delta_p)

            delta_threshold=delta_p[0]**2 + delta_p[1]**2
            times+=1


            # ---------------------
    return p
```

1.3 Tracking with *naive* template update (10 points)

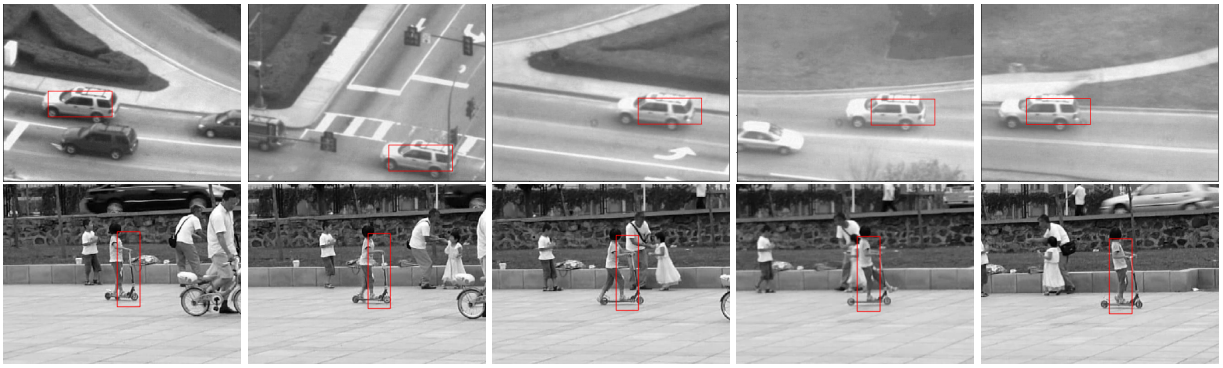


Figure 1.1: Lucas-Kanade Tracking with One Single Template

You will now test your Lucas-Kanade tracker on the `carseq.npy` and `girlseq.npy` sequences.

For this part, you have to implement the following function,

```
rects = TrackSequence(seq, rect, num_iters, threshold)
```

which receives a sequence of frames, the initial coordinates for the template to be tracked, and the number of iterations and threshold for running the LK optimization. The function will use the LK tracker you implemented in **Q1.2** to estimate the motion of the template at each frame. Finally, it must return a matrix containing the rectangle coordinates of the tracked template at each frame.

Once you implement the above function, for the **car sequence** you will have to:

1. Load the `carseq.npy` frame sequence. This sequence has a shape (H, W, N_c).
2. Run `TrackSequence` to track the car (see the top row in fig. 1.1). The `rects` matrix should have a shape $N_c \times 4$. At frame 1, the car is located at coordinates $\texttt{rect} = [59, 116, 145, 151]^\intercal$.
3. Provide tracking visualizations for frames $[1, 80, 160, 280, 410]$. These frames are not the same as those in fig. 1.1, but the visualizations should look be similar. Please use the box below to add your visualizations.
4. Save the resulting `rects` to a file called `carseqrects.npy`. You will submit it to Gradescope.

Q1.3 code TrackSequence

```
def TrackSequence(seq, rect, num_iters, threshold):
        H, W, N = seq.shape

        rects =[rect]
        It = seq[:, :, 0]

        # ------------------
        # TODO: Iterate over the sequence to track the object in of interest. Add
        ↪  your code here:
        for i in tqdm(range(1, N)):
                p = LucasKanade(seq[:,:,i-1], seq[:,:,i], rect, threshold,
                ↪  num_iters, p0=np.zeros(2))
                #print(p)
                rect_new=np.copy(rect)
                rect_new[0] += p[0]
                rect_new[1] += p[1]
                rect_new[2] += p[0]
                rect_new[3] += p[1]
                # print(rect_new,p)
                rects.append(rect_new)
                rect=rect_new
        #pass
        # ------------------
        rects = np.array(rects)
        print(rects.shape)
        # Just a sanity check
        assert rects.shape == (N, 4), f"Your output sequence {rects.shape} is not
        ↪  {N}x{4}"
        return rects
```

Q1.3 code for visualization(car as example)

```
seq = np.load("../data/carseq.npy")

# TODO: update
rect = [59.0, 116.0, 145.0, 151.0]

# NOTE: these are default parameters. Once your code works, you're encouraged to
↪  play with them by
# running ablations, and report your best results.
num_iters = 1e4
threshold = 1e-2

rects = TrackSequence(seq, rect, num_iters, threshold)
np.save(f'../out/carseqrects.npy', rects)

(next block):
seq = np.load("../data/carseq.npy")
frames_to_save =   [1, 80, 160, 280, 410]
rects = np.load("../out/carseqrects.npy")
for idx in frames_to_save:
        # NOTE: here's an example on how to visualize a frame with its template.
        frame = seq[:,:,idx]
        rect = rects[idx]
        w = rect[2] - rect[0]
        h = rect[3] - rect[1]
        plt.figure()
        axis = plt.gca()
        #print(rect[0], rect[1])
        rect_draw = patches.Rectangle((rect[0], rect[1]), w, h, linewidth=1,
        ↪  edgecolor='r', facecolor='none')
        axis.add_patch(rect_draw)
        plt.imshow(frame, cmap='gray')
        plt.axis('off')
        plt.savefig(f"../out/sol_1.3_carseq_{idx+1}.png")
        print(rect)
```

Q1.3 Car sequence visualizations

visualizations:for$[1, 80, 160, 280, 410]$

Q1.3 rect

[59.40128908 117.12893963 145.40128908 152.12893963] [206.02672444 186.5562702 292.02672444 221.5562702] [119.36751864 142.09782308 205.36751864 177.09782308] [190.26332576 123.33563642 276.26332576 158.33563642] [69.56898102 138.72016178 155.56898102 173.72016178]

For the **girl sequence** you will have to:

1. Load the `girlseq.npy` frame sequence. This sequence has a shape (H, W, N_g).
2. Run the `TrackSequence` to track the girl (see bottom row in fig. 1.1). The `rects` matrix should have a shape $N_g \times 4$. At frame 1, the girl is located at coordinates $\texttt{rect} = [280, 152, 330, 318]^\intercal$.
3. Provide tracking visualizations for frames $[1, 15, 35, 65, 85]$. These frames are not the same as those in fig. 1.1, but the visualizations should look similar. Please use the box below to add your visualizations.
4. Save the resulting `rects` to a file called `girlseqrects.npy`. You will submit it to Gradescope.

Q1.3 Girl sequence visualizations

visualizations:for$[1, 15, 35, 65, 85]$

Q1.3 rect

[280.58179593 152.88816916 330.58179593 318.88816916] [290.90378274 156.54468375 340.90378274 322.54468375] [296.16722369 162.74346042 346.16722369 328.74346042] [281.69250735 166.18142536 331.69250735 332.18142536] [312.38775003 172.83098904 362.38775003 338.83098904]

Notes:

- A frame can be visualized as `plt.imshow(frames[:, :, i]); plt.show()`
- This equation might be useful for updating the coordinates of the rectangle $\mathcal{T}_{t+1}(\mathbf{x}) = \mathcal{I}_t(\mathcal{W}(\mathbf{x}; \mathbf{p}_t))$.
- We provided initial values for the threshold and the number of iterations for the tracker, but you are encouraged to play with the parameters defined in the scripts and report the best results.

1.4 Tracking with template correction (20 points)

As you might have noticed, the image content we are tracking in the first frame differs from that in the last frame. This issue is known as *template drifting*, and it is due to error accumulation that stems from doing a *naive* update of the template. There are several template update strategies for mitigating this drifting problem. The one we will explore here is explained in the paper below,

- Ian Matthews, et al. *The Template Update Problem*. Proceedings of British Machine Vision Conference (BMVC '03), 2003. [link]

This method proposes an extension to the *naive* update. Roughly, the idea is to update the template at each step, as before, but it must be re-aligned to the initial template, $\mathcal{T}_0$, to correct for drift [3].

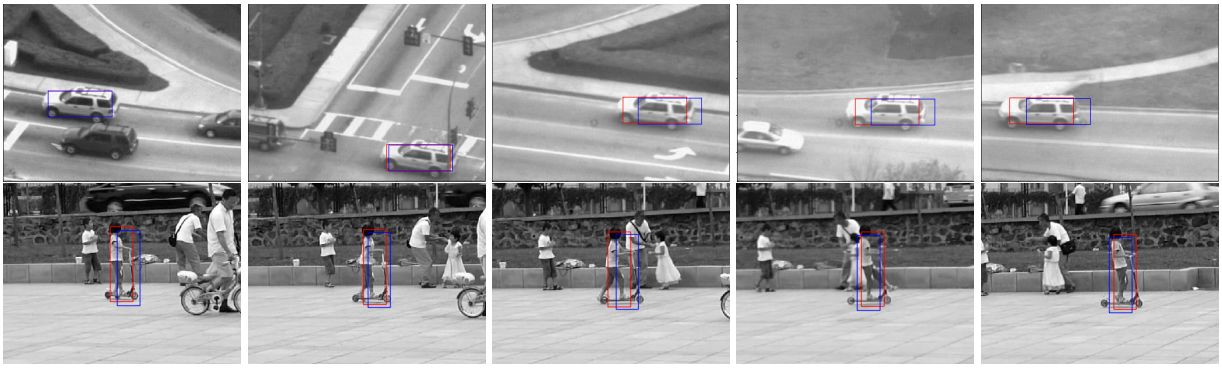


Figure 1.2: Lucas-Kanade Tracking with Template Correction

For this question, you will follow a similar process as in **Q1.3** to test, both, the **car** and **girl** sequences. Except, now, we additionally ask you to implement the template update strategy from the previous paper. Specifically, you will need to implement **Strategy 3: Template Update with Drift Correction** which follows the update below,

$$\texttt{if} \quad ||\mathbf{p}_t^* - \mathbf{p}_t|| \leq \epsilon \quad \texttt{then} \quad \mathcal{T}_{t+1}(\mathbf{x}) = \mathcal{I}_t(\mathcal{W}(\mathbf{x}; \mathbf{p}^*))$$
$$\texttt{else} \quad \mathcal{T}_{t+1}(\mathbf{x}) = \mathcal{T}_t(\mathbf{x})$$

An example of LK with template correction is given in fig. 1.2. The blue rectangles are results obtained with the baseline tracker (**Q1.3**), the red ones are results obtained with the tracker with drift correction (**Q1.4**).

Before implementing the drift correction, please read **section 2.1** and **section 2.3** of the paper. Note that you **do not** need to modify your Lucas-Kanade implementation. Once you're done reading these sections, implement the function,

```
rects = TrackSequenceWithTemplateCorrection(seq, rect, num_iters, lk_threshold,
                                            drift_threshold)
```

which, as before, receives a frame sequence, the initial coordinates of the object of interest, the number of iterations, and the threshold for running the optimization, and now, it also receives **the drift threshold parameter for the template update.**

Then, follow a similar procedure as in **Q1.3** to test and visualize your results, *i.e.* follow this to-do list for the **car sequence**,

1. Load the `carseq.npy` frame sequence.
2. Run `TrackSequenceWithTemplateCorrection` to track the car.
3. Provide tracking visualizations for frames $[1, 80, 160, 280, 410]$. Here you will provide visualizations **with** and **without** template correction in red and blue, respectively as in fig. 1.2.
4. Save the result to a file called `carseqrects-wtcr.npy`. You will submit it to Gradescope.

Q1.4 TrackSequenceWithTemplateCorrection

```
def TrackSequenceWithTemplateCorrection(seq, rect_0, num_iters, lk_threshold,
↪  drift_threshold):
        H, W, N = seq.shape
        rect = np.copy(rect_0)
        rects_wtcr =[]
        It = seq[:,:,0]
        It0 = seq[:,:,0]
        # --------------
        # TODO: Iterate over the sequence to track the object in of interest and
        ↪  do template correction.
        #  Add your code here:
        rects_wtcr.append(rect)
        # initial p
        pn=np.zeros(2)
        # initial rect
        rect_it=np.copy(rect_0)
        rect_it0=np.copy(rect_0)
        for i in tqdm(range(1, N)):
                # first formula: starts with pn-1, use Tn as tmplate
                pn = LucasKanade(It, seq[:,:,i], rect_it, lk_threshold,
                ↪  num_iters, p0=np.copy(pn))
                # append result to results list
                rect_new=np.copy(rect_it)
                rect_new[0] += pn[0]
                rect_new[1] += pn[1]
                rect_new[2] += pn[0]
                rect_new[3] += pn[1]
                rects_wtcr.append(rect_new)
                # update method
                # second formula, start with pn, use T1 as tmplate
                pn_star=LucasKanade(It0, seq[:,:,i], rect_it0, lk_threshold,
                ↪  num_iters, p0=np.copy(pn))
                # update with p_star
                p_diff = pn_star-pn
                # if '<', use pn_star and this frame
                if np.linalg.norm(p_diff) <= drift_threshold:
                        rect_it=np.copy(rect_it0)
                        rect_it[0] += pn_star[0]
                        rect_it[1] += pn_star[1]
                        rect_it[2] += pn_star[0]
                        rect_it[3] += pn_star[1]
                        It=seq[:,:,i]
                else:
                        # tmplate not changed, still use it and rect_it
                        continue
        rects_wtcr = np.array(rects_wtcr)
        # Just a sanity check
        assert rects_wtcr.shape == (N, 4), f"Your output sequence
        ↪  {rects_wtcr.shape} is not {N}x{4}"
        return rects_wtcr
```

Q1.3 code for visualization(car as example)

```
seq = np.load("../data/carseq.npy")
# TODO: update
rect =  [59.0, 116.0, 145.0, 151.0]
# NOTE: these are default parameters. Once your code works, you're encouraged to
↪  play with them by
# running ablations, and report your best results.
num_iters = 1e4
threshold = 1e-2
drift_threshold = 5
rects = TrackSequenceWithTemplateCorrection(seq, rect, num_iters, threshold,
↪  drift_threshold)
np.save("../out/carseqrects-wtcr.npy", rects)

(next block):
# TODO: visualize. An example for how to visualize the outputs
# List of frames to visualize
frames_to_save =  [1, 80, 160, 280, 410]
seq = np.load("../data/carseq.npy")
rects_wtcr = np.load("../out/carseqrects-wtcr.npy")
rects = np.load("../out/carseqrects.npy")
for idx in frames_to_save:
        #pass
        rect_wtcr = rects_wtcr[idx]
        rect = rects[idx]
        frame = seq[:,:,idx]
        w = rect[2] - rect[0]
        h = rect[3] - rect[1]
        plt.figure()
        currentAxis = plt.gca()
        rect_wcrt_draw = patches.Rectangle(
        (rect_wtcr[0], rect_wtcr[1]), w, h, linewidth=1, edgecolor='r',
        ↪  facecolor='none')
        currentAxis.add_patch(rect_wcrt_draw)
        rect_draw = patches.Rectangle(
        (rect[0], rect[1]), w, h, linewidth=1, edgecolor='b', facecolor='none')
        currentAxis.add_patch(rect_draw)
        plt.imshow(frame, cmap='gray')
        plt.axis('off')
        plt.savefig(f"../out/sol_1.4_carseq_{idx+1}.png")
        print("rect:"+str(rect))
        print("rect_wtcr:"+str(rect_wtcr))
```

Q1.4 Car Sequence with and without template correction

visualizations in red and blue :for$[1, 80, 160, 280, 410]$

Q1.4 Car Sequence rect

[59.40128908 117.12893963 145.40128908 152.12893963] [59.40128908 117.12893963 145.40128908 152.12893963]
[206.02672444 186.5562702 292.02672444 221.5562702] [206.25204846 186.71187455 292.25204846 221.71187455]
[119.36751864 142.09782308 205.36751864 177.09782308] [101.29995648 140.64668564 187.29995648 175.64668564]
[190.26332576 123.33563642 276.26332576 158.33563642] [169.80042063 122.51810479 255.80042063 157.51810479]
[69.56898102 138.72016178 155.56898102 173.72016178] [47.49168477 137.38808432 133.49168477 172.38808432]

And follow this to-do list for the **girl sequence**,

1. Load the `girlseq.npy` frame sequence.
2. Run `TrackSequenceWithTemplateCorrection` to track the girl.
3. Provide tracking visualizations for frames $[1, 15, 35, 65, 85]$. Here you will provide visualizations **with** and **without** template correction in red and blue, respectively as in fig. 1.2.
4. Save the result to a file called `girlseqrects-wtcr.npy`. You will submit it to Gradescope.

Q1.4 Girl sequence with and without template correction

visualizations in red and blue:for$[1, 15, 35, 65, 85]$

Q1.4 Car Sequence rect

[280.58179593 152.88816916 330.58179593 318.88816916] [280.59490183 152.89923326 330.59490183 318.89923326]
[290.90378274 156.54468375 340.90378274 322.54468375] [294.36798192 158.06902019 344.36798192 324.06902019]
[296.16722369 162.74346042 346.16722369 328.74346042] [303.56531107 164.16063632 353.56531107 330.16063632]
[281.69250735 166.18142536 331.69250735 332.18142536] [316.40983807 164.63039457 366.40983807 330.63039457]
[312.38775003 172.83098904 362.38775003 338.83098904] [340.38437912 173.84775062 390.38437912 339.84775062]

Notes:

- Again, we provide you with initial parameters, but you are encouraged to play with them to see how each parameter affects the tracking results and report your best ones.

2 Affine Motion Subtraction (35 total points)

In the first section of this homework, we assumed the motion is limited to pure translation and a single template. In this section, you will now implement a tracker for **affine motion**. For implementing such tracker, you will be working on two main parts: 1) estimating the dominant affine motion in subsequent images (section 2.1) ; and 2) identifying pixels corresponding to moving objects in the scene (section 2.2). For testing it, as before, you will be asked to visualize the results of your implementation (section 2.3).

Coding questions for **part 2** should be implemented in `LucasKanadeAffine.ipynb`. We provide starter code for each question, as well as default parameters. You will test your tracker, on an ant sequence, `antseq.npy` (top row in fig. 2.1), and an aerial sequence of moving vehicles, `aerialseq.npy` (bottom row in fig. 2.1), both of which you can find in the `data` folder.

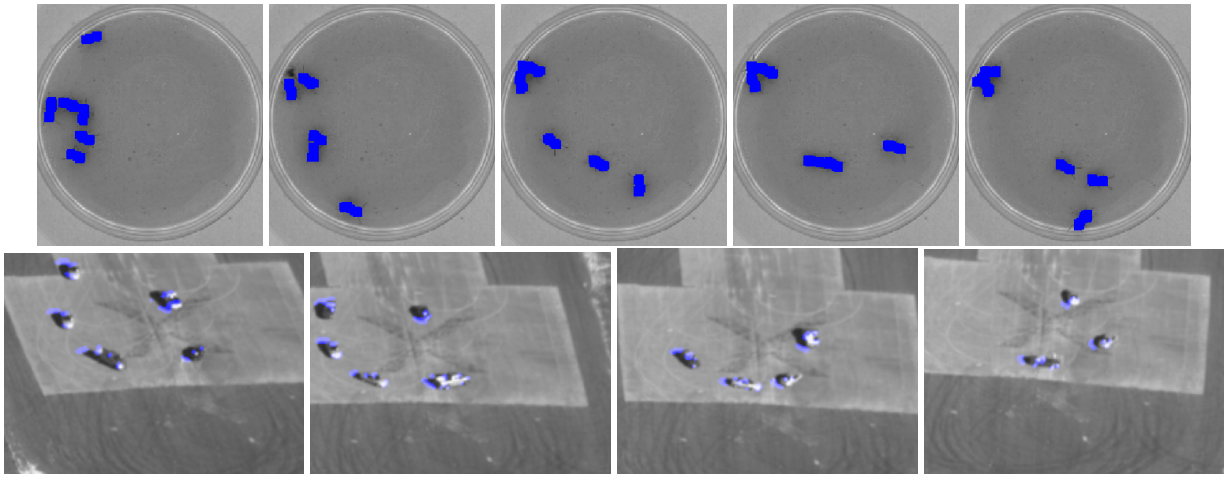


Figure 2.1: Lucas-Kanade Tracking with Motion Detection

2.1 Dominant Motion Estimation (15 points)

In this section, you will implement a tracker for affine motion using a planar affine warp function. To estimate the dominant motion, the entire image $\mathcal{I}_t$ will serve as the template to be tracked in image $\mathcal{I}_{t+1}$, that is, $\mathcal{I}_{t+1}$ is assumed to be approximately an affine warped version of $\mathcal{I}_t$. This approach is reasonable under the assumption that a majority of the pixels correspond to stationary objects in the scene whose depth variation is small relative to their distance from the camera.

Let us now define an **affine** warp function as,

$$\mathbf{x}' = \mathcal{W}(\mathbf{x}; \mathbf{p}) = \begin{bmatrix} 1+p_1 & p_2 \\ p_4 & 1+p_5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} p_3 \\ p_6 \end{bmatrix} \quad . \tag{2.1}$$

As described in [2], one can represent this affine warp in homogeneous coordinates as,

$$\tilde{\mathbf{x}}' = \mathbf{M}\tilde{\mathbf{x}} \tag{2.2}$$

where

$$\mathbf{M} = \begin{bmatrix} 1+p_1 & p_2 & p_3 \\ p_4 & 1+p_5 & p_6 \\ 0 & 0 & 1 \end{bmatrix} \quad . \tag{2.3}$$

where $\mathbf{M}$ will differ between successive image pairs.

Similar as before, the algorithm starts with an initial guess of $\mathbf{p} = [0, 0, 0, 0, 0, 0]^T$, *i.e.* $\mathbf{M} = \mathbb{I}$. To determine $\Delta\mathbf{p}$, you will need to iteratively solve a least-squares such that $\mathbf{p} \rightarrow \mathbf{p} + \Delta\mathbf{p}$ at each iteration.

Write the function,

```
M = LucasKanadeAffine(It, It1, num_iters, threshold)
```

where the input parameters are similar to those of `LucasKanade` in **Q1.2**, but note that we are **not** using `rect` since we will use the entire image as the template. The function should now return a 3×3 affine transformation matrix `M`.

Notes:

- `LucasKanadeAffine` should be relatively similar to `LucasKanade`. In **Q1.2** the template to be tracked is usually small, compared to tracking the whole image. For this part, image $\mathcal{I}_t$ will almost always not be fully contained in the warped version $\mathcal{I}_{t+1}$. Therefore, the matrix of image derivatives, $\mathbf{A}$, and the temporal derivatives, $\partial\mathcal{I}_t$, must be computed only on the pixels lying in the region common to $\mathcal{I}_t$ and the warped version of $\mathcal{I}_{t+1}$.

Include the code you wrote for this part in the box below:

Q2.1 Code for LucasKanadeAffine

```
# We recommend using this function, but you can explore other methods as well
↪  (e.g., ndimage.shift).
from scipy.interpolate import RectBivariateSpline
# The function below could be useful as well :)
from numpy.linalg import lstsq

def LucasKanadeAffine(It, It1, threshold, num_iters):
        # Initial M
        # M = np.eye(3)
        M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
        # ----------
        # TODO: Add your LK implementation here:
        # Use whole img to build mode:
        y=np.arange(0,It1.shape[0])
        x=np.arange(0,It1.shape[1])

        # build mode for interpolation of It, It1
        mode1 = RectBivariateSpline(y, x, It1)
        mode_tmp=RectBivariateSpline(y, x, It)
        # model for gradient
        gIy,gIx=np.gradient(It1)
        mode_gx = RectBivariateSpline(y, x, gIx)
        mode_gy=  RectBivariateSpline(y, x, gIy)
        # for iteration times
        times = 0
        delta_threshold=2
        xv,yv=np.meshgrid(x,y)
        while ( delta_threshold>= threshold) and (times<num_iters):
                # 1.Warp I with W(x;p), I(W(x;p);
                # get new w and h
                M_x = M[0]
                M_y = M[1]
                new_x = M_x[0] * xv + M_x[1] * yv+M_x[2]
                new_y = M_y[0] * xv + M_y[1] * yv+M_y[2]
                #computed only on the pixels lying in the region common to It and
                ↪  the warped version of It+1.
                # build matrix -> range
                x_region = np.logical_and(new_x>0,new_x<It.shape[1])
                y_region = np.logical_and(new_y>0,new_y<It.shape[0])
                in_region = np.logical_and(x_region,y_region)
                # delte pixels not in range
                new_x = new_x[in_region]
                new_y = new_y[in_region]
```

(please see the rest code in next two page)

Q2.1 Code for LucasKanadeAffine

```
                new_xv=xv[in_region]
                new_yv=yv[in_region]
                # 2.Compute error image T(x) - I(W(x;p))
                # get new img
                new_I=model(new_y,new_x,grid=False)
                #print(new_I.shape)
                b_new_I = new_I.reshape(-1,1)
                # to match dimension, also delte tmplate not in range
                b_tmp=It[in_region]
                #print(b_new_I.shape,b_tmp.shape)
                b_tmp=b_tmp.reshape(-1,1)
                b=b_tmp-b_new_I

                # 3.Warp gradient image I with W(x;p) -> according to writeup,
                ↪  need to use 2nd method
                # method1 -> not use
                # gradient_x=model(new_y, new_x, dy=1,grid=False)
                # gradient_y=model(new_y, new_x, dx=1,grid=False)
                # method 2
                gradient_x=mode_gx(new_y,new_x,grid=False)
                gradient_y=mode_gy(new_y,new_x,grid=False)
                gradient_x=gradient_x.reshape(1,-1)[0]
                gradient_y=gradient_y.reshape(1,-1)[0]
                # 4.Evaluate Jacobian
                # [x y 1 0 0 0]
                # [0 0 0 x y 1]
                A=[]
                A1=[]
                A2=[]
                A3=[]
                A4=[]
                new_xv=new_xv.reshape(1,-1)[0]
                new_yv=new_yv.reshape(1,-1)[0]
                #5. Compute the steepest descent img
                for i in range(len(gradient_x)):
                        A1.append(gradient_x[i]*new_xv[i])
                        A2.append(gradient_y[i]*new_xv[i])
                        A3.append(gradient_x[i]*new_yv[i])
                        A4.append(gradient_y[i]*new_yv[i])
                A.append(A1)
                A.append(A3)
                A.append(gradient_x.tolist())
                A.append(A2)
                A.append(A4)
                A.append(gradient_y.tolist())
                A=np.array(A)
                # transpose
(please see the rest code in next page)
```

Q2.1 Code for LucasKanadeAffine

```
            A=np.transpose(A,(1,0))
            # 6.Compute delta p
            delta_p=np.linalg.lstsq(A, b, rcond=None)[0]
            # 7.Update parameters p ← p + delta p
            delta_threshold=np.linalg.norm(delta_p)
            # print(delta_p.shape)
            M[0][0]+=delta_p[0]
            M[0][1]+=delta_p[1]
            M[0][2]+=delta_p[2]
            M[1][0]+=delta_p[3]
            M[1][1]+=delta_p[4]
            M[1][2]+=delta_p[5]
            #print(M)
            # ------------------------------------
    return M
```

2.2 Moving Object Detection (10 points)

Once you're able to compute the affine warp $\mathbf{M}$ between the image pair $\mathcal{I}_t$ and $\mathcal{I}_{t+1}$, you have to determine the pixels corresponding to moving objects. One naive way to do so is as follows:

1. Warp the image $\mathcal{I}_t$ using $\mathbf{M}$ so that it is registered to $\mathcal{I}_{t+1}$. To do this, you may find the functions `scipy.ndimage.affine_transform` or `cv2.warpAffine` useful. Please carefully read their corresponding documentation on whether $\mathbf{M}$ or $\mathbf{M}^{-1}$ needs to be passed to the function.
2. Subtract the warped image from $\mathcal{I}_{t+1}$; the locations where the absolute difference exceeds a tolerance can then be declared as corresponding to the locations of moving objects. To obtain better results, you might find the following functions useful: `scipy.morphology.binary_erosion`, and `scipy.morphology.binary_dilation`.

Write the following function,

```
mask = SubtractDominantMotion(It, It1, num_iters, threshold, tolerance)
```

which receives the image pair `It` and `It1`, the number of iterations and threshold parameters for running the LK optimization, and the **tolerance to determine the moving pixels**. The function must return a `mask` which is a binary image specifying which pixels correspond to moving objects. Note that you should use `LucasKanadeAffine` within this function to derive the transformation matrix $\mathbf{M}$, and produce the according binary mask.

Include the code you wrote for this part in the box below:

Q2.2 Code for SubtractDominantMotion

```
# These functions could be useful for your implementation.
from scipy.ndimage import binary_erosion, binary_dilation, affine_transform
import cv2

def SubtractDominantMotion(It, It1, num_iters, threshold, tolerance):
        mask = np.ones(It1.shape, dtype=bool)
        # ------------------
        # TODO: Add your code here:
        m=LucasKanadeAffine(It,It1,threshold,num_iters)
        #computed only on the pixels lying in the region common to It and the
        ↪  warped version of It+1.
        m_x = m[0]
        m_y = m[1]
        y=np.arange(0,It1.shape[0])
        x=np.arange(0,It1.shape[1])
        xv,yv=np.meshgrid(x,y)
        new_x = m_x[0] * xv + m_x[1] * yv+m_x[2]
        new_y = m_y[0] * xv + m_y[1] * yv+m_y[2]
        # in region matrix
        x_region = np.logical_and(new_x>0,new_x<It.shape[1])
        y_region = np.logical_and(new_y>0,new_y<It.shape[0])
        in_region = np.logical_and(x_region,y_region)
        # Warp the image It using M, register to It+1
        #m_inv = np.linalg.inv(m)
        #It_warp = affine_transform(It,m_inv)
        h=It.shape[0]
        w=It.shape[1]
        It_warp= cv2.warpAffine(It,m,(w,h))
        # Subtract the warped image from It+1
        It_diff = np.abs(It_warp-It1)
        # mask function to get 0 or 1
        # if > tolerance: True
        def mask(x,tol):
                if x < tol:
                        return False
                else:
                        return True
        func = np.vectorize(mask)
        mask = func(It_diff,tolerance)
        # these not in region: false
        mask[~in_region] = False
        # make it looks better
        kernel_e = np.ones((3,3))
        kernel_d = np.ones((4,4))
        #mask=binary_erosion(mask,kernel_e)#,iterations=1)
        mask=binary_dilation(mask,kernel_d,iterations=2)
        mask=binary_erosion(mask,kernel_e,iterations=2)
        # print(mask)
        # --------
        return mask
```

2.3 Tracking with affine motion (10 points)

Similar to **Q1.3** and **Q1.4**, you will now test your implementation of the Lucas-Kanade tracker with affine motion on the `antseq.npy` and `aerialseq.npy`.

Implement the function,

```
mask = TrackSequenceAffineMotion(seq, num_iters, threshold, tolerance)
```

which receives a sequence of frames, the number of iterations and threshold for running the optimization and the tolerance for the motion subtraction. The function must return `masks`, a matrix which contains the binary outputs from the motion subtraction method you implemented in the previous section.

Then, follow a similar procedure as in **Q1.3** to test and visualize your results, *i.e.* follow this to-do list for the **ant sequence**,

1. Loads the `antseq.npy` frame sequence.
2. Runs `TrackSequenceAffineMotion` to track the ants.
3. Provides visualizations for frames $[30, 60, 90, 120]$, overlaying the corresponding binary masks similar to those in fig. 2.1.
4. You will **not** upload the `masks` to Gradescope.
5. Report your run-time performance.

Q2.3 code for TrackSequenceAffineMotion

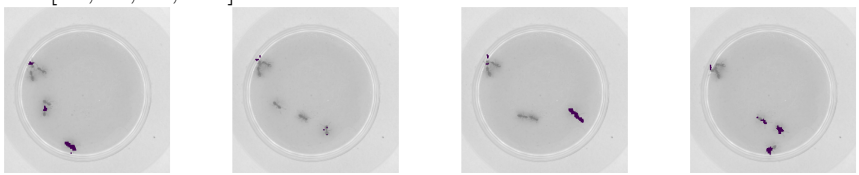
```
def TrackSequenceAffineMotion(seq, num_iters, threshold, tolerance):
        H, W, N = seq.shape

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

        # --------------
        # TODO: Add your code here:
        for i in tqdm(range(1, seq.shape[2])):
                mask =
                ↪  SubtractDominantMotion(seq[:,:,i-1],seq[:,:,i],num_iters,threshold,tolerance)
                masks.append(mask)

        # ------------
        masks = np.stack(masks, axis=2)
        return masks
```

Q2.3 code for visualization(ant as example)

```
seq = np.load("../data/antseq.npy")
# NOTE: feel free to play with these parameters
num_iters = 1e4
threshold = 1e-2
tolerance = 0.2
masks = TrackSequenceAffineMotion(seq, num_iters, threshold, tolerance)
np.save(f'../out/antseqmasks.npy', masks)

(next block)
# TODO: visualize
frames_to_save = [30, 60, 90, 120]
seq= np.load("../data/antseq.npy")
masks=np.load("../out/antseqmasks.npy")

for idx in frames_to_save:
        #pass
        frame = seq[:,:,idx]
        mask = masks[:,:,idx]

        plt.figure()
        plt.imshow(frame, cmap="gray", alpha=0.5)
        plt.imshow(np.ma.masked_where(np.invert(mask), mask))
        plt.axis('off')
        plt.savefig(f"../out/sol_2.3_antseq_{idx+1}.png")
```

Q2.3 Ant sequence visualizations

visualizations:for[30, 60, 90, 120]



Q2.3 Ant sequence run-time performance

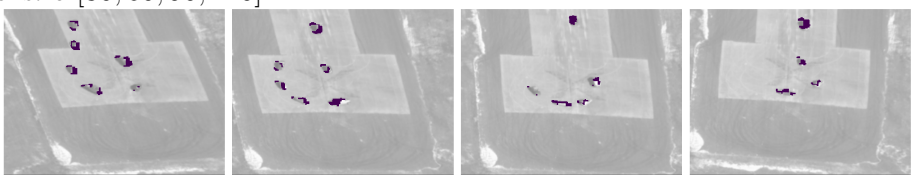
124/124 [01:52¡00:00, 1.10it/s]

And follow this to-do list for the **aerial sequence**,

1. Loads the `aerial.npy` frame sequence.
2. Runs `TrackSequenceAffineMotion` to track the cars.
3. Provides visualizations for frames [30, 60, 90, 120], overlaying the corresponding binary masks similar to those in fig. 2.1.
4. You will **not** upload the `masks` to Gradescope.
5. Report your run-time performance.

Q2.3 Aerial sequence Visualizations

visualizations:for[30, 60, 90, 120]



Q2.3 Aerial sequence run-time performance

149/149 [05:06¡00:00, 2.06s/it]

Notes:

1. The **ant sequence** involves little camera movement and can help you debug your mask generation procedure.

2. Feel free to visualize the motion detection performance in a way that you would prefer, but please make sure it can be visually inspected without undue effort.

3 Efficient Tracking (15 total points)

In this section, you will explore Lucas-Kanade with inverse composition for efficient tracking.

Coding questions for **part 3** should be implemented in `LucasKanadeEfficient.ipynb`. Again, we provide starter code for each question, as well as default parameters. You will test your tracker, on `antseq.npy` and `aerialseq.npy`.

3.1 Inverse Composition (10 points)

The inverse compositional extension of the Lucas-Kanade algorithm [2] has been used in literature to great effect for the task of efficient tracking. When utilized within tracking it attempts to linearize the current frame as:

$$\mathcal{I}_t(\mathcal{W}(\mathbf{x}; \mathbf{0} + \Delta\mathbf{p}) \approx \mathcal{I}_t(\mathbf{x}) + \frac{\partial \mathcal{I}_t(\mathbf{x})}{\partial \mathbf{x}^T} \frac{\partial \mathcal{W}(\mathbf{x}; \mathbf{0})}{\partial \mathbf{p}^T} \Delta\mathbf{p} \quad . \tag{3.1}$$

In a similar manner to the conventional Lucas-Kanade algorithm one can incorporate these linearized approximations into a vectorized form such that,

$$\arg\min_{\Delta\mathbf{p}} ||\mathbf{A}'\Delta\mathbf{p} - \mathbf{b}'||_2^2 \tag{3.2}$$

for the specific case of an affine warp where we can recover $\mathbf{p}$ from $\mathbf{M}$ and $\Delta\mathbf{p}$ from $\Delta\mathbf{M}$. This results in the update $\mathbf{M} = \mathbf{M}(\Delta\mathbf{M})^{-1}$.

Here,

$$\Delta\mathbf{M} = \begin{bmatrix} 1 + \Delta p_1 & \Delta p_2 & \Delta p_3 \\ \Delta p_4 & 1 + \Delta p_5 & \Delta p_6 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

With this in mind, write the function,

```
M = InverseCompositionAffine(It, It1, num_iters, threshold)
```

which re-implements function `LucasKanadeAffine` from **Q2.1**, but now using the aforesaid inverse compositional method.

Include the code you wrote for this part in the box below:

Q3.1 Code for InverseCompositionAffine

```
from scipy.ndimage import affine_transform
import cv2
from scipy.interpolate import RectBivariateSpline
def InverseCompositionAffine(It, It1, threshold, num_iters):
        # Initial M
        # M = np.eye(3)
        M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
        # ---------------
        # TODO: Add your Efficient LK implementation here:
        # Pre-compute:
        # Evaluate the gradient of the template
        gradient_y,gradient_x=np.gradient(It)
        # Evaluate the Jacobian, Compute the steepest descent images
        y=np.arange(0,It.shape[0])
        x=np.arange(0,It.shape[1])
        xv,yv=np.meshgrid(x,y)
        gradient_x=gradient_x.reshape(1,-1)[0]
        gradient_y=gradient_y.reshape(1,-1)[0]
        xv=xv.reshape(1,-1)[0]
        yv=yv.reshape(1,-1)[0]
        A=[]
        A1=[]
        A2=[]
        A3=[]
        A4=[]
        for i in range(len(gradient_x)):
                A1.append(gradient_x[i]*xv[i])
                A2.append(gradient_y[i]*xv[i])
                A3.append(gradient_x[i]*yv[i])
                A4.append(gradient_y[i]*yv[i])
        A.append(A1)
        A.append(A3)
        A.append(gradient_x.tolist())
        A.append(A2)
        A.append(A4)
        A.append(gradient_y.tolist())
        A=np.array(A)
        #print(A.shape)
        # transpose
        A=np.transpose(A,(1,0))
        # Compute the inverse Hessian matrix using Equation
        H = A.T @ A
        H_inv = np.linalg.inv(H)
```

(see rest code in next page)

Q3.1 Code for InverseCompositionAffine

```
delta_threshold=2
times = 0
mode1 = RectBivariateSpline(y, x, It1)
while ( delta_threshold>= threshold) and (times<num_iters):
        #computed only on the pixels lying in the region common to It and
        ↪  the warped version of It+1.
        M_x = M[0]
        M_y = M[1]
        new_x = M_x[0] * xv + M_x[1] * yv+M_x[2]
        new_y = M_y[0] * xv + M_y[1] * yv+M_y[2]
        x_region = np.logical_and(new_x>0,new_x<It.shape[1])
        y_region = np.logical_and(new_y>0,new_y<It.shape[0])
        in_region = np.logical_and(x_region,y_region)
        # warp I with W
        h, w= It1.shape
        # It_warp= cv2.warpAffine(It1,M,(w,h))
        new_x = new_x[in_region]
        new_y = new_y[in_region]
        It_warp=mode1(new_y,new_x,grid=False)
        # get error image
        # tmplate in region
        b_tmp=It.reshape(in_region.shape)[in_region]
        b_tmp=b_tmp.reshape(-1,1)
        b_new_I = It_warp.reshape(-1,1)
        b=b_new_I-b_tmp

        # make A and b in region -> dimension matching
        A_new = A[in_region]
        # use formula-> get H
        H = A_new.T @ A_new
        # inverse H
        H_inv = np.linalg.inv(H)
        # formula to get delta_p
        tmp = A_new.T @ b
        delta_p = H_inv @ tmp
        # update delta_m
        delta_M =
        ↪  np.array([[delta_p[0][0]+1,delta_p[1][0],delta_p[2][0]],
        [delta_p[3][0],1+delta_p[4][0],delta_p[5][0]],
        [0,0,1]])
        #Update the w
        M = M @ np.linalg.inv(delta_M)
        delta_p_th=delta_p.reshape(1,-1)[0]
        delta_threshold=np.linalg.norm(delta_p_th)
        times+=1
#print(delta_p_th,delta_threshold)
# ---------------------
return M
```

Notes:

1. The notation $\mathbf{M}(\Delta\mathbf{M})^{-1}$ corresponds to $\mathcal{W}(\mathcal{W}(\mathbf{x};\Delta\mathbf{p})^{-1};\mathbf{p})$ in Section 2.2 in [1].

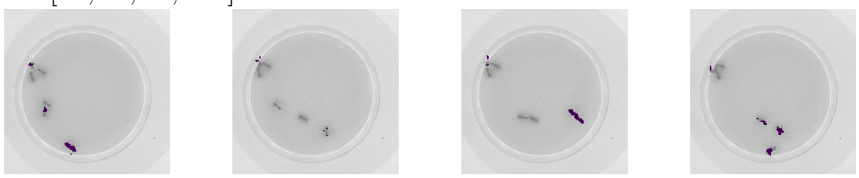
3.2 Tracking with Inverse Composition (5 points)

You will now re-use your `SubtractDominantMotion` and `TrackSequenceAffineMotion` implemented in **Q2.2** and **Q2.3** to provide the visualizations for the ant and aerial sequences.

Thus, for the **ant sequence** follow the same to-do list as in **Q2.3**. When reporting your run-time performance, please also include the percentage gain with respect to **Q2.3**.

Q3.2 Ant sequence visualizations

visualizations:for[30, 60, 90, 120]



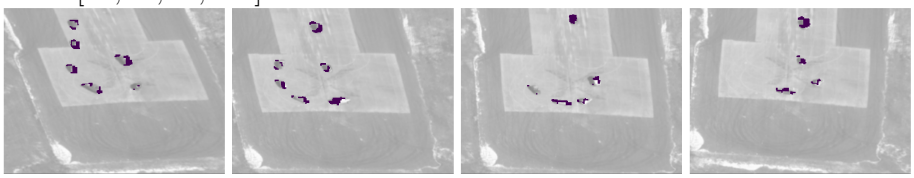
Q3.2 Ant sequence run-time performance and performance gain

124/124 [00:43¡00:00, 2.84it/s] percentage gain: 61.61%

For the **aerial sequence** follow the same to-do list as in **Q2.3**. When reporting your run-time performance, please also include the percentage gain with respect to **Q2.3**.

Q3.2 Aerial sequence visualizations

visualizations:for[30, 60, 90, 120]



Q3.2 Aerial sequence run-time performance and performance gain

149/149 [01:09¡00:00, 2.14it/s] percentage gain: 77.45%

Finally, in your own words, please describe briefly why the inverse compositional approach is more computationally efficient than the classical approach.

Q3.2 Anwer

It is more computationally efficient for not to computer steepest descent image(matrix A) and inverse Hessian matrix in every iteration but only once in pre-compution. So lots of time can be saved for only computing matrix A for one time and more computationally efficient.

4 Extra credit

Find a 10s video clip of your choice and run Lucas-Kanade tracking on a salient foreground object in this video. Needless to say, the object you track should undergo considerable amount of motion in the scene and should not be static. It is even better if this object undergoes an occlusion and your algorithm is able to track it across this occlusion. To report your results in the write-up, capture 6 frames of the tracked video at 0s, 2s, 4s, 6s, 8s, 10s and include these.

main function code

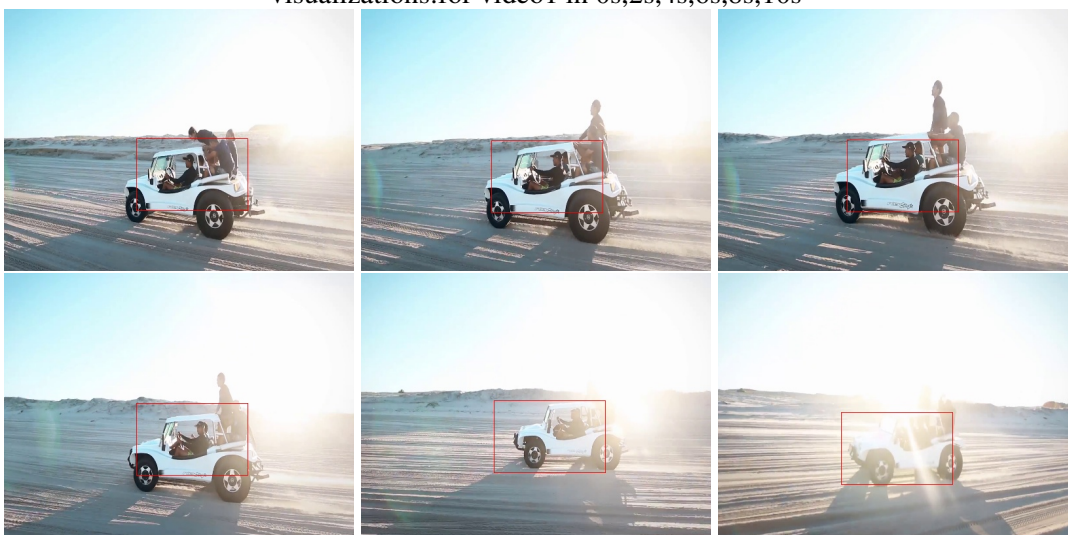
```
import cv2
import time

def video_process(rect_0,name,num,drift_threshold = 5):
        cap = cv2.VideoCapture(name)
        fps = int(cap.get(cv2.CAP_PROP_FPS))
        count = 0
        start = time.time()
        #rect_0 =[260.0, 240.0, 430.0, 350.0]
        pn=np.zeros(2)
        # initial rect
        rect_it=np.copy(rect_0)
        rect_it0=np.copy(rect_0)

        num_iters = 1e4
        threshold = 1e-2

        rect_it=np.copy(rect_0)
        rect_it0=np.copy(rect_0)
        first_frame = cv2.imread("video"+str(num)+"_0.jpg")
        first_frame = cv2.cvtColor(first_frame, cv2.COLOR_BGR2GRAY)
        It0 = first_frame.copy()
        It = first_frame.copy()
        seconds=[0,2,4,6,8,10]
        index=0
        i=0

        while cap.isOpened():
                ret,frame = cap.read()
                frame = cv2.resize(frame,(640,480))
                show_frame = frame.copy()
                frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
                frame=np.array(frame)
                #print(frame)

                #print(frame.shape)
                pn,delta_p = LucasKanade(It, frame, rect_it, threshold,
                ↪  num_iters, p0=np.copy(pn))
                # append result to results list
                rect_new=np.copy(rect_it)
                rect_new[0] += pn[0]
                rect_new[1] += pn[1]
                rect_new[2] += pn[0]
                rect_new[3] += pn[1]
                # second formula, start with pn, use T1
(see rest in next page)
```

main function code

```
# second formula, start with pn, use T1
pn_star,delta_p_star=LucasKanade(It0, frame, rect_it0, threshold,
↪  num_iters, p0=np.copy(pn))
# update with p_star
p_diff = pn_star-pn
if np.linalg.norm(p_diff) <= drift_threshold and delta_p <
↪  threshold:
        rect_it=np.copy(rect_it0)
        rect_it[0] += pn_star[0]
        rect_it[1] += pn_star[1]
        rect_it[2] += pn_star[0]
        rect_it[3] += pn_star[1]
        It=frame
show_frame = cv2.rectangle(show_frame,
↪  (int(rect_new[0]),int(rect_new[1])),
↪  (int(rect_new[2]),int(rect_new[3])), (0, 0, 255), 1)
if i==fps*seconds[index]:
        cv2.imwrite("video"+str(num)+"_"+str(seconds[index])+".jpg",
        ↪  show_frame)
        index+=1

cv2.imshow('LK tracking', show_frame)
#cv2.imwrite("frame%d.jpg" % count, frame)
i+=1
if (cv2.waitKey(10) & 0xFF == ord('q')) or (i>(fps*10)):
        break

cap.release()
cv2.destroyAllWindows()
```

Visualizations for the corresponding frames

visualizations:for video1 in 0s,2s,4s,6s,8s,10s



visualizations:for video2 in 0s,2s,4s,6s,8s,10s

Explain any changes you had to make in the algorithm to get it to work on this video.

Answer

1. In LucasKanade function, return the delta p value. if p value is larger than threshold of delta p, it means the tracking is poor and situation like occlusion(trees block some part vehicles in video2) or the picture brightness is too high(video) happens. So if continue use this frame as template is bad, so if this happens, not update tmplate by adding another judge:

```
if np.linalg.norm(p_diff) <= drift_threshold and delta_p < threshold:
        rect_it=np.copy(rect_it0)
        rect_it[0] += pn_star[0]
        rect_it[1] += pn_star[1]
        rect_it[2] += pn_star[0]
        rect_it[3] += pn_star[1]
        It=frame
```

2. Use cv2 to get frame in video, and get FPS to count time. Also use cv2.rectangle to show rectangle and result.

```
cap = cv2.VideoCapture(name)
fps = int(cap.get(cv2.CAP_PROP_FPS))
...
show_frame = cv2.rectangle(show_frame, (int(rect_new[0]),int(rect_new[1])),
↪  (int(rect_new[2]),int(rect_new[3])), (0, 0, 255), 1)
```

5 Deliverables

The assignment (code and writeup) should be submitted to Gradescope. The write-up should be submitted to Gradescope named <AndrewId>_hw2.pdf. The code should be submitted as a zip named <AndrewId>_hw2.zip to Gradescope. The zip when uncompressed should produce the following files.

- `LucasKanade.ipynb`
- `LukasKanadeAffine.ipynb`
- `LukasKanadeEfficient.ipynb`
- (Optional) `ExtracCredit.ipynb`
- `carseqrects.npy`
- `carseqrects-wtcr.npy`
- `girlseqrects.npy`
- `girlseqrects-wtcr.npy`

***Do not include the data directory in your submission.**

6 Frequently Asked Questions (FAQs)

Q1: Why do we need to use `ndimage.shift` or `RectBivariateSpline` for moving the rectangle template?

A1: When moving the rectangle template with $\Delta\mathbf{p}$, you can either move the points inside the template or move the image in the opposite direction. If you choose to move the points, the new points can have fractional coordinates, so you need to use `RectBivariateSpline` to sample the image intensity at those fractional coordinates. If you instead choose to move the image with `ndimage.shift`, you don't need to move the points and you can sample the image intensity at those points directly. The first approach could be faster since it does not require moving the entire image.

Q2: What's the right way of computing the image gradients $\mathcal{I}_x(\mathbf{x})$ and $\mathcal{I}_y(\mathbf{x})$. Should I first sample the image intensities $\mathcal{I}(\mathbf{x})$ at $\mathbf{x}$ and then compute the image gradients $\mathcal{I}_x(\mathbf{x})$ and $\mathcal{I}_y(\mathbf{x})$ with $\mathcal{I}(\mathbf{x})$? Or should I first compute the entire image gradients $\mathcal{I}_x$ and $\mathcal{I}_y$ and sample them at $\mathbf{x}$?

A2: The second approach is the correct one.

Q3: Can I use pseudo-inverse for the least-squared problem $\arg\min_{\Delta\mathbf{p}} ||\mathbf{A}\Delta\mathbf{p} - \mathbf{b}||_2^2$?

A3: Yes, the pseudo-inverse solution of $\mathbf{A}\Delta\mathbf{p} = \mathbf{b}$ is also $\Delta\mathbf{p} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$ when $\mathbf{A}$ has full column ranks, i.e., the linear system is overdetermined.

Q4: For inverse compositional Lucas-Kanade, how should I deal with points outside out the image?

A4: Since the Hessian in inverse compositional Lucas Kanade is precomputed, we cannot simply remove points when they are outside the image since it can result in dimension mismatch. However, we can set the error $\mathcal{I}_{t+1}(\mathcal{W}(\mathbf{x};\mathbf{p})) - \mathcal{I}_t(\mathbf{x})$ to 0 for $\mathbf{x}$ outside the image.

Q5: How to find pixels common to both It1 and It?

A5: If the coordinates of warped It1 is within the range of It.shape, then we consider the pixel lies in the common region.

References

[1] Simon Baker, Ralph Gross, Iain Matthews, and Takahiro Ishikawa. Lucas-kanade 20 years on: A unifying framework: Part 2. Technical Report CMU-RI-TR-03-01, Carnegie Mellon University, Pittsburgh, PA, February 2003.

[2] Simon Baker and Iain Matthews. Lucas-kanade 20 years on: A unifying framework part 1: The quantity approximated, the warp update rule, and the gradient descent approximation. *International Journal of Computer Vision - IJCV*, 01 2004.

[3] Iain Matthews, Takahiro Ishikawa, and Simon Baker. The template update problem. In *Proceedings of British Machine Vision Conference (BMVC '03)*, September 2003.