16-720A Computer Vision: Homework 3
Lucas-Kanade Tracking

Instructor: Srinivasa G Narasimhan
TAs: Abhijat Biswas, Adam Villaflor, Akankshya Kar, Dai Li, Ritwick Chaudhry, Ziyan Wang and Harsh Sharma

Due: Thursday, October 22nd, 2020 11:59 p.m.

- Please submit your code **and** write-up to Gradescope in accordance with the complete submission checklist at the end of this document.
- All tasks marked with a **Q** require a submission.
- Please stick to the provided function signatures, variable names, and file names.
- **Start early**! This homework cannot be completed within two hours!
- **Verify your implementation as you proceed**: otherwise you will risk having a huge mess of malfunctioning code that can go wrong anywhere.

* * *

This homework consists of four sections. In the first section you will implement a simple Lucas-Kanade (LK) tracker with one single template. The second section requires you to implement a motion subtraction method for tracking moving pixels in a scene. In the final section you shall study efficient tracking using inverse composition. Other than the course slide decks, the following references may also be helpful:

1. 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
2. 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

Both are available at:
`https://www.ri.cmu.edu/pub_files/pub3/baker_simon_2002_3/baker_simon_2002_3.pdf`.
`https://www.ri.cmu.edu/publications/lucas-kanade-20-years-on-a-unifying-framework-part-2/`.

1 Lucas-Kanade Tracking

In this section you will be implementing a simple Lucas & Kanade tracker with one single template. In the scenario of two-dimensional tracking with a pure translation warp function,

$$\mathcal{W}(\mathbf{x};\mathbf{p}) = \mathbf{x} + \mathbf{p} \ . \tag{1}$$

The problem can be described as follows: starting with a rectangular neighborhood of pixels $\mathbb{N} \in \{\mathbf{x}_d\}_{d=1}^{D}$ on frame $\mathcal{I}_t$, the Lucas-Kanade tracker aims to move it by an offset

$\mathbf{p} = [p_x, p_y]^T$ to obtain another rectangle on the next frame $\mathcal{I}_{t+1}$, so that the pixel squared difference in the two rectangles is minimized:

$$\mathbf{p}^* = \quad \arg\min_{\mathbf{p}} \quad = \sum_{\mathbf{x}\in\mathbb{N}} ||\mathcal{I}_{t+1}(\mathbf{x}+\mathbf{p}) - \mathcal{I}_t(\mathbf{x})||_2^2 \tag{2}$$

$$= \left|\left| \begin{bmatrix} \mathcal{I}_{t+1}(\mathbf{x}_1+\mathbf{p}) \\ \vdots \\ \mathcal{I}_{t+1}(\mathbf{x}_D+\mathbf{p}) \end{bmatrix} - \begin{bmatrix} \mathcal{I}_t(\mathbf{x}_1) \\ \vdots \\ \mathcal{I}_t(\mathbf{x}_D) \end{bmatrix} \right|\right|_2^2 \tag{3}$$

Q1.1 (5 points) Starting with an initial guess of $\mathbf{p}$ (for instance, $\mathbf{p} = [0, 0]^T$), we can compute the optimal $\mathbf{p}^*$ iteratively. In each iteration, the objective function is locally linearized by first-order Taylor expansion,

$$\mathcal{I}_{t+1}(\mathbf{x}' + \Delta\mathbf{p}) \approx \mathcal{I}_{t+1}(\mathbf{x}') + \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial \mathcal{W}(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T} \Delta\mathbf{p} \tag{4}$$

where $\Delta\mathbf{p} = [\Delta p_x, \Delta p_y]^T$, is the change in template offset. Further, $\mathbf{x}' = \mathcal{W}(\mathbf{x};\mathbf{p}) = \mathbf{x} + \mathbf{p}$ and $\frac{\partial \mathcal{I}(\mathbf{x}')}{\partial \mathbf{x}'^T}$ is a vector of the $x-$ and $y-$ image gradients at pixel coordinate $\mathbf{x}'$. In a similar manner to Equation 3 one can incorporate these linearized approximations into a vectorized form such that,

$$\arg\min_{\Delta\mathbf{p}} ||\mathbf{A}\Delta\mathbf{p} - \mathbf{b}||_2^2 \tag{5}$$

Then, we update with the minimizer as $\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p}$ at each iteration. We repeat until convergence.

- What is $\frac{\partial \mathcal{W}(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T}$?
- What is $\mathbf{A}$ and $\mathbf{b}$?
- What conditions must $\mathbf{A}^T\mathbf{A}$ meet so that a unique solution to $\Delta\mathbf{p}$ can be found?

Q1.2 (15 points) Implement a function with the following signature

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

that computes the optimal local motion from frame $\mathcal{I}_t$ to frame $\mathcal{I}_{t+1}$ that minimizes Equation 3. Here `It` is the image frame $\mathcal{I}_t$, `It1` is the image frame $\mathcal{I}_{t+1}$, `rect` is the 4-by-1 vector that represents a rectangle describing all the pixel coordinates within $\mathbb{N}$ within the image frame $\mathcal{I}_t$, and p_0 is the initial parameter guess $(\delta x, \delta y)$. The four components of the rectangle are $[x_1, y_1, x_2, y_2]^T$, where $[x_1, y_1]^T$ 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 the four corners. You will also need to iterate the estimation until the change in $||\Delta\mathbf{p}||_2^2$ is below a threshold.

To deal with fractional movement of the template, you will need to interpolate the image. We suggest using `RectBivariateSpline` from the `scipy.interpolate` package. Read the documentation of defining the spline (`RectBivariateSpline`) as well as evaluating the spline using `RectBivariateSpline.ev` carefully. Please note that `RectBivariateSpline.ev` can be used to evaluate the image values and gradients.

Q1.3 (10 points) Write a script `testCarSequence.py` that loads the video frames from `carseq.npy`, and runs the Lucas-Kanade tracker that you have implemented in the previous task to track the car. `carseq.npy` can be located in the `data` directory and it contains one single three-dimensional matrix: the first two dimensions correspond to the *height* and *width* of the frames respectively, and the third dimension contain the indices of the frames (that is, the first frame can be visualized with `imshow(frames[:, :, 0])`). The rectangle in the first frame is $[x_1, y_1, x_2, y_2]^T = [59, 116, 145, 151]^T$. Report your tracking performance (image + bounding rectangle) at frames 1, 100, 200, 300 and 400 in a format similar to Figure 1. Also, create a file called `carseqrects.npy`, which contains one single $n \times 4$ matrix, where each row stores the `rect` that you have obtained for each frame, and n is the total number of frames. You are encouraged to play with the parameters defined in the scripts and report the best results.

Similarly, write a script `testGirlSequence.py` that loads the video from `girlseq.npy` and runs your Lucas-Kanade tracker on it. The rectangle in the first frame is $[x_1, y_1, x_2, y_2]^T = [280, 152, 330, 318]^T$. Report your tracking performance (image + bounding rectangle like in Figure 1) at frames 1, 20, 40, 60 and 80 in a format similar to Figure 1. Also, create a file called `girlseqrects.npy`, which contains one single $n \times 4$ matrix, where each row stores the `rect` that you have obtained for each frame.

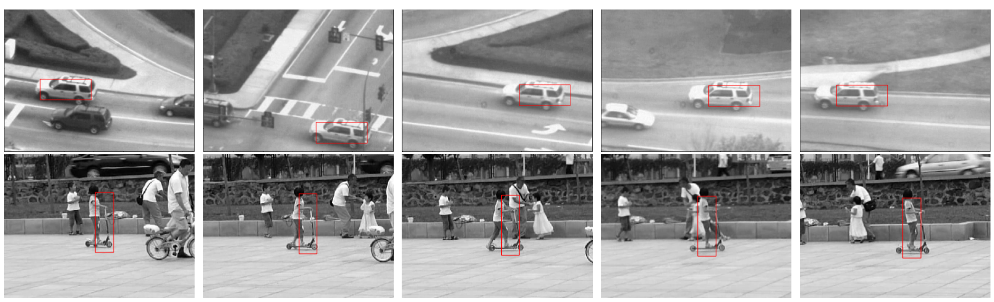

Figure 1: Lucas-Kanade Tracking with One Single Template

Q1.4 (20 points) As you might have noticed, the image content we are tracking in the first frame differs from the one in the last frame. This is understandable since we are updating the template after processing each frame and the error can be accumulating. This problem is known as *template drifting*. There are several ways to mitigate this problem. One possible approach comes from Iain Matthews et al.(2003).

`https://www.ri.cmu.edu/publication_view.html?pub_id=4433`

We want you to implement their approach (strategy 3 in section 2.1), so please read the paper carefully. Write two scripts with a similar functionality to **Q1.3** but with this template correction routine incorporated: `testCarSequenceWithTemplateCorrection.py` and `testGirlSequenceWithTemplateCorrection.py`. Save the `rects` as `carseqrects-wcrt.npy` and `girlseqrects-wcrt.npy`, and also report the performance at those frames. An example is given in Figure 2. Again, you are encouraged to play with the parameters defined in the scripts to see how each parameter affects the tracking results.

Here the blue rectangles are created with the baseline tracker in **Q1.3**, the red ones with the tracker in **Q1.4**. The tracking performance has been improved non-trivially. Note that you do not necessarily have to draw two rectangles in each frame, but make sure that the

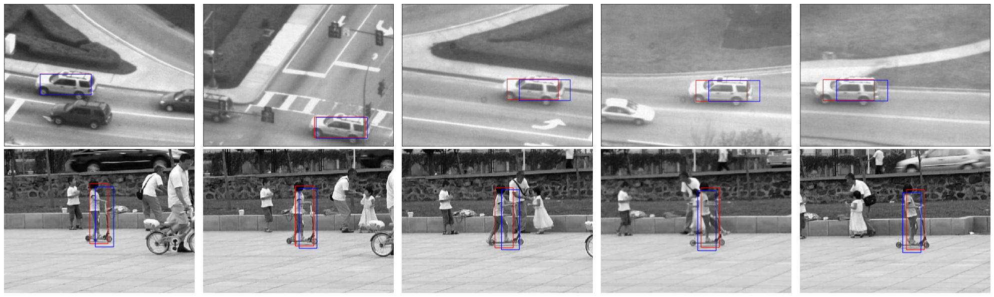

Figure 2: Lucas-Kanade Tracking with Template Correction

performance improvement can be easily visually inspected.

2 Affine Motion Subtraction

In this section, you will implement a tracker for estimating dominant affine motion in a sequence of images and subsequently identify pixels corresponding to moving objects in the scene. You will be using the images in the file `aerialseq.npy`, which consists of aerial views of moving vehicles from a non-stationary camera.

2.1 Dominant Motion Estimation

In the first section of this homework we assumed the the motion is limited to pure translation. In this section you shall implement a tracker for affine motion using a planar affine warp function. To estimate 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 the stationary objects in the scene whose depth variation is small relative to their distance from the camera.

Using a planar affine warp function you can recover the vector $\Delta\mathbf{p} = [p_1, \ldots, p_6]^T$,

$$\mathbf{x}' = \mathcal{W}(\mathbf{x}; \mathbf{p}) = \begin{bmatrix} 1+p_1 & p_2 \\ p_4 & 1+p_5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} p_3 \\ p_6 \end{bmatrix} \quad . \tag{6}$$

One can represent this affine warp in homogeneous coordinates as,

$$\tilde{\mathbf{x}}' = \mathbf{M}\tilde{\mathbf{x}} \tag{7}$$

where,

$$\mathbf{M} = \begin{bmatrix} 1+p_1 & p_2 & p_3 \\ p_4 & 1+p_5 & p_6 \\ 0 & 0 & 1 \end{bmatrix} \quad . \tag{8}$$

Here $\mathbf{M}$ represents $\mathbf{W}(\mathbf{x}; \mathbf{p})$ in homogeneous coordinates as described in [1]. Also note that $\mathbf{M}$ will differ between successive image pairs. Starting with an initial guess of $\mathbf{p} = \mathbf{0}$ (i.e. $\mathbf{M} = \mathbf{I}$) you will need to solve a sequence of least-squares problem to determine $\Delta\mathbf{p}$

such that $\mathbf{p} \rightarrow \mathbf{p} + \Delta\mathbf{p}$ at each iteration. Note that unlike previous examples where the template to be tracked is usually small in comparison with the size of the image, image $\mathcal{I}_t$ will almost always not be contained fully in the warped version $\mathcal{I}_{t+1}$. Hence, one must only consider pixels lying in the region common to I_t and the warped version of I_{t+1} when forming the linear system at each iteration. **Hint** Use Equation 4 and determine $\frac{\partial \mathcal{W}(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T}$ for the affine transform.

Q2.1 (15 points) Write a function with the following signature

```
LucasKanadeAffine(It, It1)
```

which returns the affine transformation matrix `M`, and `It` and `It1` are I_t and I_{t+1} respectively. `LucasKanadeAffine` should be relatively similar to `LucasKanade` from the first section and again we suggest you use `RectBivariateSpline`.

2.2 Moving Object Detection

Once you are able to compute the transformation matrix $\mathbf{M}$ relating an image pair $\mathcal{I}_t$ and $\mathcal{I}_{t+1}$, a naive way for determining pixels lying on moving objects is as follows: warp the image $\mathcal{I}_t$ using $\mathbf{M}$ so that it is registered to $\mathcal{I}_{t+1}$ and subtract it from $\mathcal{I}_{t+1}$; the locations where the absolute difference exceeds a threshold can then be declared as corresponding to locations of moving objects. To obtain better results, you can check out the following scipy.morphology functions: `binary_erosion`, and `binary_dilation`.

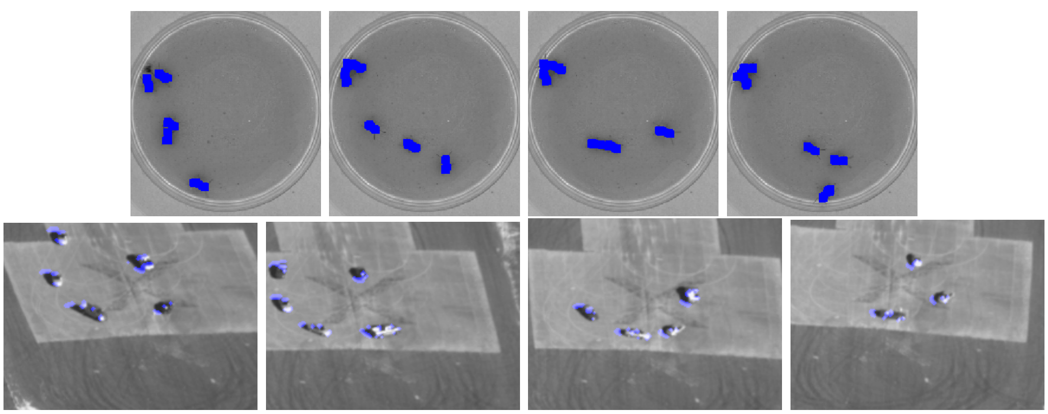

Figure 3: Lucas-Kanade Tracking with Motion Detection

Q2.2 (10 points) Using the function you have developed for dominant motion estimation, write a function with the following signature

```
SubtractDominantMotion(image1, image2)
```

where `image1` and `image2` form the input image pair, and the return value `mask` is a binary image of the same size that dictates which pixels are considered to be corresponding to moving objects. You should invoke `LucasKanadeAffine` in this function to derive the transformation matrix $\mathbf{M}$, and produce the aforementioned binary mask accordingly.

Q2.3 (10 points) Write two scripts `testAntSequence.py` and `testAerialSequence.py` that load the image sequence from `antseq.npy` and `aerialseq.npy` and run the motion detection routine you have developed to detect the moving objects. Try to implement `testAntSequence.py` first as it involves little camera movement and can help you debug your mask generation procedure. Report the performance at frames 30, 60, 90 and 120 with the corresponding binary masks superimposed, as exemplified in Figure 3. 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

3.1 Inverse Composition

The inverse compositional extension of the Lucas-Kanade algorithm (see [1]) has been used in literature to great effect for the task of efficient tracking. When used in tracking, the main idea is that in each iteration we are trying to find the inverse warping $\mathcal{W}(\mathbf{x};\Delta\mathbf{p})^{-1}$ that best aligns our current tracking rectangle $\mathcal{I}_{t+1}(\mathcal{W}(\mathbf{x};\mathbf{p})$ in the next frame with the template in the current frame $\mathcal{I}_t(x)$. This is equivalent to finding the warping on the template $\mathcal{I}_t(\mathcal{W}(\mathbf{x};\Delta\mathbf{p})$ that best aligns it with $\mathcal{I}_{t+1}(\mathcal{W}(\mathbf{x};\mathbf{p})$. Thus, we can view the Inverse Compositional Algorithm for tracking as attempting to minimize

$$\mathbf{p}^* = \quad \arg\min_{\mathbf{p}} \quad = \sum_{\mathbf{x}\in\mathbb{N}} ||\mathcal{I}_t(\mathcal{W}(\mathbf{x};\Delta\mathbf{p}) - \mathcal{I}_{t+1}(\mathcal{W}(\mathbf{x};\mathbf{p})||_2^2 \tag{9}$$

where we linearize just around the current frame $\mathcal{I}_t$ (instead of around the next frame $\mathcal{I}_{t+1}$) as

$$\mathcal{I}_t(\mathcal{W}(\mathbf{x};\mathbf{0}+\Delta\mathbf{p}) \approx \mathcal{I}_t(\mathbf{x}) + \frac{\partial \mathcal{I}_t(\mathbf{x})}{\partial \mathbf{x}^T}\frac{\partial \mathcal{W}(\mathbf{x};\mathbf{0})}{\partial \mathbf{p}^T}\Delta\mathbf{p} \quad . \tag{10}$$

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

$$\arg\min_{\Delta\mathbf{p}} ||\mathbf{A}'\Delta\mathbf{p} - \mathbf{b}'||_2^2 \tag{11}$$

Then, we update our parameters using $\mathbf{W}(\mathbf{x};\mathbf{p}) \leftarrow \mathbf{W}(\mathbf{W}(\mathbf{x};\Delta\mathbf{p})^{-1};\mathbf{p})$. Note that for our specific case of using an affine warp where $\mathbf{p} \Leftrightarrow \mathbf{M}$ and $\Delta\mathbf{p} \Leftrightarrow \Delta\mathbf{M}$ this results in the update $\mathbf{M} \leftarrow \mathbf{M}(\Delta\mathbf{M})^{-1}$. You can refer to the Lucas Kanade Lecture or section 2.2 of [2]:

`https://www.ri.cmu.edu/pub_files/pub3/baker_simon_2003_3/baker_simon_2003_3.pdf`

Q3.1 (15 points) Reimplement the function `LucasKanadeAffine(It,It1)` as `InverseCompositionAffine(It,It1)` using the inverse compositional method. In your own words please describe why the inverse compositional approach is more computationally efficient than the classical approach? **Hint** Is there anything special we can do if $\mathbf{A}'$ doesn't depend on $\mathbf{p}$?

4 HW3 Submission Checklist

The assignment (code and writeup) should be submitted to Gradescope with each part on its own page. The code should be submitted as a zip named <AndrewId>.zip. The zip when uncompressed should produce the following files.

- `code/LucasKanade.py`
- `code/LucasKanadeAffine.py`
- `code/SubtractDominantMotion.py`
- `code/InverseCompositionAffine.py`
- `code/testCarSequence.py`
- `code/testCarSequenceWithTemplateCorrection.py`
- `code/testGirlSequence.py`
- `code/testGirlSequenceWithTemplateCorrection.py`
- `code/testAntSequence.py`
- `code/testAerialSequence.py`
- `result/carseqrects.npy`
- `result/carseqrects-wcrt.npy`
- `result/girlseqrects.npy`
- `result/girlseqrects-wcrt.npy`

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

5 Frequently Asked Questions (FAQs)

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

A1: When moving the rectangle template with $\Delta\mathbf{p}$, the new points can have fractional coordinates. So, you need to use `RectBivariateSpline` to sample the image intensity at those fractional coordinates.

Q2: What's the right way of computing the image gradients $\mathcal{I}_x(\mathbf{x})$ and $\mathcal{I}_y(\mathbf{x})$?

A2: You should first compute the entire image gradients $\mathcal{I}_x$ and $\mathcal{I}_y$ and then sample them at $\mathbf{x}$.

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.

Q4: How can I debug my inverse compositional Lucas Kanade?

A4: The easiest way is by checking whether you are getting similar results as **Q2.3** on `antseq.npy` and `aerialseq.npy`. Additionally, `InverseCompositionAffine(It,It1)` should be running much faster than `LucasKanadeAffine(It,It1)`. For our implementation, `InverseCompositionAffine(It,It1)` runs about 2 to 3 times faster.