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

Instructor: Srinivasa Narasimhan TAs: Adam Harley, Peiyun Hu, Rishi Madhok, Talha Siddiqui, Anuj Pahuja, Nitin Singh

Due: Tuesday, March 19th, 2019 11:59 p.m.

- Please pack your code and write-up into a single file `<andrewid>.zip`, 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.
- If you have any questions, please post in Piazza or visit the TAs during the office hours.

* * *

This homework consists of four sections. In the first section you will implement a simple Lucas-Kanade (LK) tracker with one single template; in the second section, the tracker will be generalized to accommodate for large appearance variance. The third 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 such as 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,

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

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 frame $\mathcal{I}_{t+1}$, so that the pixel squared difference in the two rectangles is minimized:

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

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

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 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,

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

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

- 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. To deal with fractional movement of the template, you will need to interpolate the image using the Scipy module `ndimage.shift` or something similar. You will also need to iterate the estimation until the change in $||\Delta\mathbf{p}||_2^2$ is below a threshold. In order to perform interpolation you might find `RectBivariateSpline` from the `scipy.interpolate` package. Read the documentation of defining the spline (`RectBivariateSpline`) as well as evaluating the spline using `RectBivariateSpline.ev` carefully.

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.

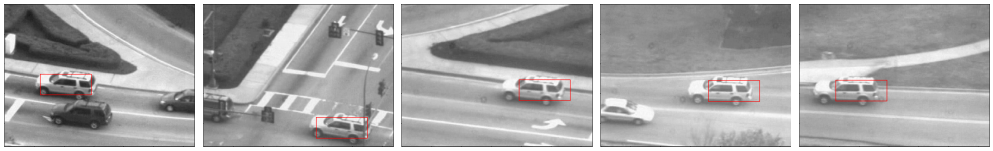

Figure 1: Lucas-Kanade Tracking with One Single Template

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. Iain Matthews et al. (2003, `https://www.ri.cmu.edu/publication_view.html?pub_id=4433`) suggested one possible approach. Write a script `testCarSequenceWithTemplateCorrection.py` with a similar functionality to **Q1.3**, but with a template correction routine incorporated. Save the resulting `rects` as `carseqrects-wcrt.npy`, and also report the performance at those frames. An example is given in Figure 2.

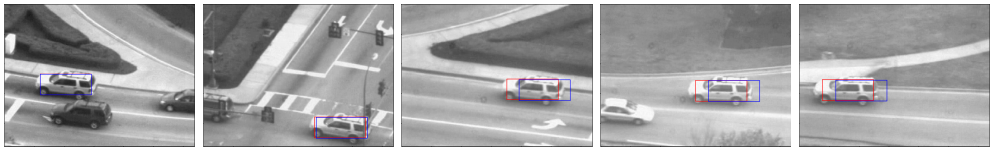

Figure 2: Lucas-Kanade Tracking with Template Correction

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 performance improvement can be easily visually inspected.

2 Lucas-Kanade Tracking with Appearance Basis

The tracker we have implemented in the first secion, with or without template drifting correction, may suffice if the object being tracked is not subject to drastic appearance variance. However, in real life, this can hardly be the case. We have prepared another sequence `sylvseq.npy` (the initial rectangle is $[101, 61, 155, 107]$), with exactly the same format as `carseq.mat`, on which you can test the baseline implementation and see what

would happen. In this section, you will implement a variant of the Lucas-Kanade tracker (see Section 3.4 in [2]), to model linear appearance variation in the tracking.

2.1 Appearance Basis

One way to address this issue is to use eigen-space approach (aka, principal component analysis, or PCA). The idea is to analyze the historic data we have collected on the object, and produce a few bases, whose linear combination would most likely constitute to the appearance of the object in the new frame. This is actually similar to the idea of having a lot of templates, but looking for too many templates may be expensive, so we only worry about the *principal* templates.

Mathematically, suppose we are given a set of k image bases $\{\mathcal{B}_k\}_{k=1}^K$ of the same size. We can approximate the appearance variation of the new frame $\mathcal{I}_{t+1}$ as a linear combination of the previous frame $\mathcal{I}_t$ and the bases weighted by $\mathbf{w} = [w_1, \ldots, w_K]^T$, such that

$$\mathcal{I}_{t+1}(\mathbf{x}) = \mathcal{I}_t(\mathbf{x}) + \sum_{k=1}^{K} w_k \mathcal{B}_k(\mathbf{x}) \tag{6}$$

Q2.1 (5 points) Express $\mathbf{w}$ as a function of $\mathcal{I}_{t+1}$, $\mathcal{I}_t$, and $\{\mathcal{B}_k\}_{k=1}^K$, given Equation 6. Note that since the $\mathcal{B}_k$'s are orthobases, thay are orthogonal to each other.

2.2 Tracking

Given K bases, $\{\mathcal{B}_k\}_{k=1}^K$, our goal is then to simultaneously find the translation $\mathbf{p} = [p_x, p_y]^T$ and the weights $\mathbf{w} = [w_1, \ldots, w_K]^T$ that minimizes the following objective function:

$$\min_{\mathbf{p},\mathbf{w}} = \sum_{\mathbf{x}\in\mathbb{N}} ||\mathcal{I}_{t+1}(\mathbf{x}+\mathbf{p}) - \mathcal{I}_t(\mathbf{x}) - \sum_{k=1}^{K} w_k \mathcal{B}_k(\mathbf{x})||_2^2 \ . \tag{7}$$

Again, starting with an initial guess of $\mathbf{p}$ (for instance, $\mathbf{p} = [0, 0]^T$), one can linearize $\mathcal{I}_{t+1}(\mathbf{x}+\mathbf{p}+\Delta\mathbf{p})$ with respect to $\Delta\mathbf{p}$. In a similar manner to Equation 5 one can incorporate these linearized approximations into a vectorized form such that,

$$\arg\min_{\Delta\mathbf{p},\mathbf{w}} ||\mathbf{A}\Delta\mathbf{p} - \mathbf{b} - \mathbf{B}\mathbf{w}||_2^2 \ . \tag{8}$$

As discussed in Section 3.4 of [2] (ignore the inverse compositional discussion) this can be simplified down to

$$\arg\min_{\Delta\mathbf{p}} ||\mathbf{B}^{\perp}(\mathbf{A}\Delta\mathbf{p} - \mathbf{b})||_2^2 \tag{9}$$

where $\mathbf{B}^{\perp}$ spans the null space of $\mathbf{B}$. Note that $||\mathbf{B}^{\perp}\mathbf{z}||_2^2 = ||\mathbf{z} - \mathbf{B}\mathbf{B}^T\mathbf{z}||_2^2$ when $\mathbf{B}$ is an orthobasis.

Q2.2 (15 points) Implement a function with the following signature

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

where `bases` is a three-dimensional matrix that contains the bases. It has the same format as `frames` as is described earlier and can be found in `sylvbases.npy`.

Q2.3 (15 points) Write a script `testSylvSequence.py` that loads the video frames from `sylvseq.npy` and runs the new Lucas-Kanade tracker to track the sylv (the toy). The bases are available in `sylvbases.npy` in the `data` directory. The rectangle in the first frame is $[x_1, y_1, x_2, y_2]^T = [101, 61, 155, 107]^T$. Please report the performance of this tracker at frames 1, 200, 300, 350 and 400 (the frame + bounding box), in comparison to that of the tracker in the first section. That is, there should be two rectangles for each frame, as exemplified in Figure 3. Also, create a `sylvseqrects.npy` for all the `rects` you have obtained for each frame. It should contain one single $N \times 4$ matrix named `rects`, where N is the number of frames, and each row contains $[x_1, y_1, x_2, y_2]^T$, where $[x1, y1]^T$ is the coordinate of the top-left corner of the tracking box, and $[x_2, y_2]^T$ the bottom-right corner.

Figure 3: Lucas-Kanade Tracking with Appearance Basis

3 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.

3.1 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 $\Delta\mathbf{p} = [p_1, \ldots, p_6]^T$,

$$\mathbf{x}' = \mathcal{W}(\mathbf{x}; \mathbf{p}) = \begin{bmatrix} 1+p_1 & p_2 \\ p_4 & 1+p_5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} p_3 \\ p_6 \end{bmatrix} \quad . \tag{10}$$

One can represent this affine warp in homogeneous coordinates as,

$$\tilde{\mathbf{x}}' = \mathbf{M}\tilde{\mathbf{x}} \tag{11}$$

where,

$$\mathbf{M} = \begin{bmatrix} 1+p_1 & p_2 & p_3 \\ p_4 & 1+p_5 & p_6 \\ 0 & 0 & 1 \end{bmatrix} \quad . \tag{12}$$

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.

Q3.1 (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 (you will probably also find `scipy.ndimage.affine_transform` helpful).

3.2 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`.

Q3.2 (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.

Q3.3 (10 points) Write a script `testAerialSequence.py` that loads the image sequence from `aerialseq.npy` and run the motion detection routine you have developed to detect the moving objects. Report the performance at frames 30, 60, 90 and 120 with the corresponding binary masks superimposed, as exemplified in Figure 4. 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.

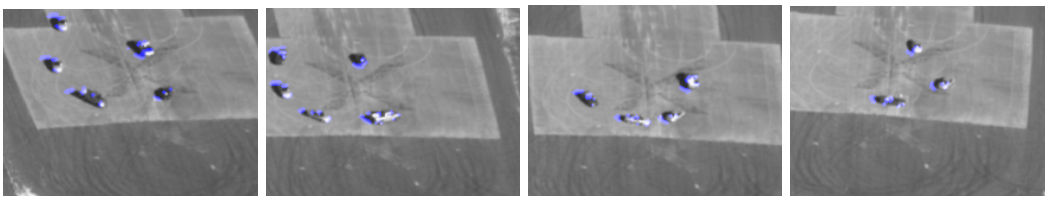

Figure 4: Lucas-Kanade Tracking with Motion Detection

4 Efficient Tracking

4.1 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 utilized within tracking it attempts to linearize the current frame as,

$$\mathcal{I}_t(\mathcal{W}(\mathbf{x};\mathbf{0}+\Delta\mathbf{p}) \approx \mathcal{I}_t(\mathbf{x}) + \frac{\partial \mathcal{I}_t(\mathbf{x})}{\partial \mathbf{x}^T}\frac{\partial \mathcal{W}(\mathbf{x};\mathbf{0})}{\partial \mathbf{p}^T}\Delta\mathbf{p} \quad . \tag{13}$$

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

$$\arg\min_{\Delta\mathbf{p}} ||\mathbf{A}'\Delta\mathbf{p} - \mathbf{b}'||_2^2 \tag{14}$$

for the specific case of an affine warp where $\mathbf{p} \leftarrow \mathbf{M}$ and $\Delta\mathbf{p} \leftarrow \Delta\mathbf{M}$ this results in the update $\mathbf{M} = \mathbf{M}(\Delta\mathbf{M})^{-1}$. Just to clarify, the notation $\mathbf{M}(\Delta\mathbf{M})^{-1}$ corresponds to $\mathbf{W}(\mathbf{W}(\mathbf{x};\Delta\mathbf{p})^{-1};\mathbf{p})$ in Section 2.2 from [2].

Q4.1 (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?

5 Deliverables

The assignment (code and writeup) should be submitted to canvas. The writeup should also be submitted to Gradescope named <AndrewId>_hw3.pdf. The code should be submitted as a zip named <AndrewId>.zip. The zip when uncompressed should produce the following files.

- `LucasKanade.py`
- `LucasKanadeAffine.py`
- `LucasKanadeBasis.py`
- `SubtractDominantMotion.py`
- `InverseCompositionAffine.py`
- `testCarSequence.py`

- `testSylvSequence.py`
- `testCarSequenceWithTemplateCorrection.py`
- `testAerialSequence.py`
- `carseqrects.npy`
- `carseqrects-wcrt.npy`
- `sylvseqrects.npy`

Do not include the data directory in your submission.