Carnegie Mellon University
16-720: Computer Vision
Lucas-Kanade Tracking

- **Due date.** Please refer to course schedule for the due date for **HW2**.
- **Gradescope submission.** You will need to submit both (1) a `YourAndrewName.pdf` and (2) a `YourAndrewName.zip` file containing your code, either as standalone python (`.py` or colab notebooks (`.ipynb`). We suggest you create a PDF by `print`ing your colab notebook from your browser. However, this can improperly cutoff images that straddle multiple pages. In this case, we suggest you download such images separately and explicitly append them to your PDF using online tools such as https://combinepdf.com/. You may also find it useful to look at this post: https://askubuntu.com/questions/2799/how-to-merge-several-pdf-files. Remember to use Gradescope's functionality to mark pages that provide answers for specific questions, such as code or visualizations.

The starter code can be found at the course gdrive folder (you will need your andrew account to access):
https://drive.google.com/drive/folders/1JZGnpUG6Ca1o0q47PCxsQVQASQwJTmUu

We strongly recommend editing and running your code in Google Colab, although you are welcome to use your local machine instead.

1 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 to estimate the motion of pixels in subsequent images. This homework explores the core aspects of implementing the LK tracking method and efficiency improvements.

What you'll be doing:

This homework is worth 100 points and consists of three parts:

1. In Section 2, you will implement a Lucas-Kanade (LK) tracker for **pure translation with a single template**. You will test your implementation on two sequences `carseq.npy` and `girlseq.npy`, which will be downloaded when running from Google Colab. This section is worth 40 points.
2. In Section 3, you will modify the tracker to account for **affine motion**. Additionally, you will implement a motion subtraction method to track moving pixels in a scene. You will test your implementation on two sequences: `antseq.npy` and `aerialseq.npy`, which will be downloaded when running from Google Colab. This section is worth 35 points.

3. In Section 4, you will implement **efficient** tracking via inverse composition. You will also test your implementation on `antseq.npy` and `aerialseq.npy`. This section is worth 25 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]

2 Lucas-Kanade Tracking (40 total points)

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

Questions for this section must be implemented in `LucasKanade.ipynb`. For coding questions, 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 Figure 1) and `girlseq.npy` (bottom row in Figure 1). These files will be downloaded when running from Google Colab. For non-coding questions, please fill in the corresponding parts in `LucasKanade.ipynb`.

Problem Formulation. Following the notation in [2], 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,

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

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,

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

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

2.1 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,

$$\mathcal{I}_{t+1}(\mathbf{x}' + \Delta\mathbf{p}) \approx \mathcal{I}_{t+1}(\mathbf{x}') + \frac{\partial \mathcal{I}_{t+1}(\mathbf{x}')}{\partial \mathbf{x}'^T} \frac{\partial \mathcal{W}(\mathbf{x}; \mathbf{p})}{\partial \mathbf{p}^T} \Delta\mathbf{p} \tag{3}$$

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 2 into a vectorized form,

$$\arg\min_{\Delta\mathbf{p}} ||\mathbf{A}\Delta\mathbf{p} - \mathbf{b}||_2^2 \tag{4}$$

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

Please answer the following questions in the corresponding snippets of `LucasKanade.ipynb`:

Q2.1.1 What is $\frac{\partial \mathcal{W}(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}^T}$? (**Hint:** It should be a 2x2 matrix)

Q2.1.2 What is $\mathbf{A}$ and $\mathbf{b}$?

Q2.1.3 What conditions must $\mathbf{A}^T\mathbf{A}$ meet so that a unique solution to $\Delta\mathbf{p}$ can be found?

2.2 Lucas-Kanade (20 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 of $[\Delta p_x, \Delta p_y]^T$. 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 Equation 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 (**Q2.1.2**);
3. Run least-squares optimization (Equation 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 5) for more details.

2.3 Tracking with template update (15 points)

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,

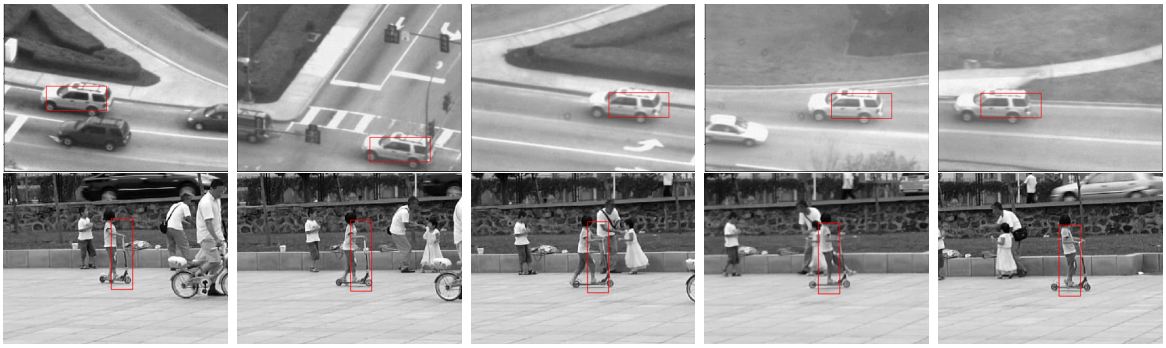

Figure 1: Lucas-Kanade Tracking with One Single Template

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

which receives a sequence of frames, the initial coordinates for the template to be tracked, the number of iterations, and the threshold for running the LK optimization. The function will use the LK tracker you implemented in **Q2.2** 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, you can test your Lucas-Kanade Tracking algorithm on the car and girl sequences.

Please run the corresponding snippets in `LucasKanade.ipynb`. You should see the tracking in both sequences. Note that there might be a slight drift towards the end of each sequence, which is perfectly normal.

3 Affine Motion Subtraction (35 total points)

In the previous 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**. Specifically, as the camera undergoes movement, the entire scene will undergo an affine motion. Our goal is to find both the affine motion of the scene and the moving objects within the scene. For implementing such tracker, you will be working on two main parts: 1) estimating the dominant affine motion in subsequent images (Section 3.1) ; and 2) identifying pixels corresponding to moving objects in the scene (Section 3.2). For testing it, as before, you will run code and visualize the results of your implementation (Section 3.3).

Questions for this section must be implemented in `LucasKanadeAffine.ipynb`. For coding questions, we provide you with starter code for each question, as well as default parameters. To test your affine motion detection, you will run your code on the ant sequence `antseq.npy` (top row in Figure 2), and an aerial sequence of moving vehicles `aerialseq.npy` (bottom row in Figure 2). These files will be downloaded when running from Google Colab.

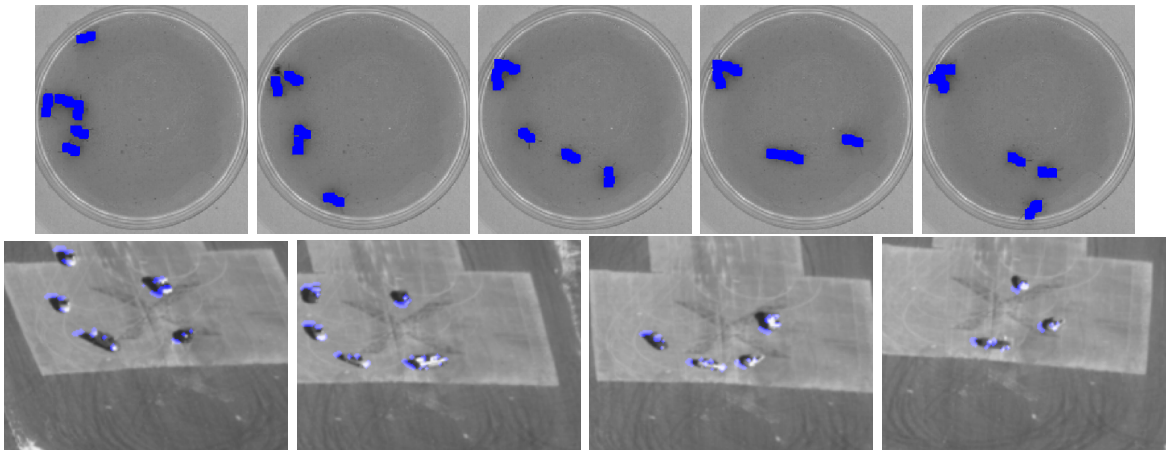

Figure 2: Lucas-Kanade Tracking with Motion Detection

3.1 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,

$$\mathbf{x}' = \mathcal{W}(\mathbf{x}; \mathbf{p}) = \begin{bmatrix} 1+p_1 & p_2 \\ p_4 & 1+p_5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} p_3 \\ p_6 \end{bmatrix} \quad . \tag{5}$$

As described in [1], one can represent this affine warp in homogeneous coordinates as,

$$\mathbf{x}' = \mathbf{M}\tilde{\mathbf{x}} \tag{6}$$

where

$$\mathbf{M} = \begin{bmatrix} 1+p_1 & p_2 & p_3 \\ p_4 & 1+p_5 & p_6 \end{bmatrix} \quad . \tag{7}$$

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

Similar as before, the algorithm starts with an initial guess of $\mathbf{p} = [0, 0, 0, 0, 0, 0]^T$. 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, threshold, num_iters)
```

where the input parameters are similar to those of `LucasKanade` in **Q2.2**, but note that we are **not** using `rect` since we will use the entire image as the template. The function should now return a 2×3 affine transformation matrix `M`.

Note: `LucasKanadeAffine` should be relatively similar to `LucasKanade`. In **Q2.2**, 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 $\mathcal{I}_{t+1}$.

3.2 Moving Object Detection (10 points)

Once you are 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.
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.ndimage.binary_erosion`, and `scipy.ndimage.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.

3.3 Tracking with affine motion (10 points)

Similar to **Q2.3**, 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, the threshold for running the optimization, and the tolerance for the motion subtraction. The function must return `masks`, a matrix that contains the binary outputs from the motion subtraction method you implemented in the previous section.

Once you implement the above function, you can test your Lucas-Kanade Affine algorithm on the aerial and ant sequences.

Please run the corresponding snippets in `LucasKanadeAffine.ipynb`. The snippets also report the runtime of the algorithms, which will be used in the next section. You should see the object movements in both sequences. The **ant sequence** involves little camera movement and can help you debug your mask generation procedure. Note that the ants may not always move in every frame, so you may observe that the mask does not include all ants in all frames.

4 Efficient Tracking (25 total points)

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

Questions for this section must 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`. For non-coding questions, please fill in the corresponding parts in `LucasKanadeEfficient.ipynb`.

4.1 Inverse Composition (15 points)

The inverse compositional extension of the Lucas-Kanade algorithm [1] has been used in literature to great effect for the task of efficient tracking. When utilized within tracking, this method attempts to linearize the current frame as:

$$\mathcal{I}_t(\mathcal{W}(\mathbf{x}; \mathbf{0} + \Delta\mathbf{p}) \approx \mathcal{I}_t(\mathbf{x}) + \frac{\partial \mathcal{I}_t(\mathbf{x})}{\partial \mathbf{x}^T} \frac{\partial \mathcal{W}(\mathbf{x}; \mathbf{0})}{\partial \mathbf{p}^T} \Delta\mathbf{p} \quad . \tag{8}$$

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

$$\arg\min_{\Delta\mathbf{p}} ||\mathbf{A}'\Delta\mathbf{p} - \mathbf{b}'||_2^2 \tag{9}$$

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,

$$\Delta\mathbf{M} = \begin{bmatrix} 1 + \Delta p_1 & \Delta p_2 & \Delta p_3 \\ \Delta p_4 & 1 + \Delta p_5 & \Delta p_6 \\ 0 & 0 & 1 \end{bmatrix} \tag{10}$$

With this in mind, write the function,

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

which re-implements function `LucasKanadeAffine` from **Q3.1**, but now using the aforementioned inverse compositional method.

Note: 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 [2].

4.2 Tracking with Inverse Composition (10 points)

You will now re-use your `SubtractDominantMotion` and `TrackSequenceAffineMotion` implemented in **Q3.2** and **Q3.3** to provide the visualizations for the ant and aerial sequences.

Please run the corresponding snippets in `LucasKanadeEfficient.ipynb`. You should see the object movements in both sequences in the same way as you get in **Q3.3**. The snippets also report the run time of the algorithms.

Please answer the following questions in the corresponding snippets of `LucasKanadeEfficient.ipynb`:

Q4.2.1 Compare the runtime of the algorithm using inverse composition (as described in this section) with its runtime without inverse composition (as detailed in the previous section) in the context of the ant and aerial sequences.

Q4.2.2 In your own words, please describe briefly why the inverse compositional approach is more computationally efficient than the classical approach.

5 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] S. Baker and I. 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.

[2] S. Baker, R. Gross, I. Matthews, and T. 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.