CS 83/183, Winter 2024

# Programming Assignment 4
# Video Tracking

Due Date: (Sunday) March 03, 2024 23:59 (ET)

## Instructions

1. **Integrity and collaboration:** Students are encouraged to work in groups but each student must write their own code and write-up. If you work as a group, include the names of your collaborators in your write up. Code should **NOT** be shared or copied. Please **DO NOT** use external code or large-language models such as chatgpt unless permitted. Plagiarism is strongly prohibited and may lead to failure of this course.

2. **Start early!** Running the code on the videos can take a lot of time, making debugging very slow.

3. **Verify your implementation as you proceed!** Otherwise you risk having a huge mess of malfunctioning code that can go wrong anywhere.

4. **Questions:** If you have any question, please look at Piazza first. Other students may have encountered the same problem, and might have been solved already. If not, post your question on the discussion board. TAs will respond as soon as possible.

5. **Write-up:** Your write-up should mainly consist of two parts, your answers to theory questions, and your insights as you attempt the programming questions. Specific items to be included in the write-up are mentioned in each question.

6. **Code:** Stick to the function prototypes mentioned in the handout. This makes verifying code easier for the TAs. If you do want to change a function prototype or add an extra parameter, please talk to the TAs.

7. **Submission:** Your submission for this assignment should be a zip file, `<DartmouthID.zip>`, composed of your write-up, your python implementations (including helper functions), and your implementations, results for extra credit (optional). Please make sure to remove the data/ folder and any other files that are not required. Ensure that your submission is of a reasonable size. You may want to use video compression if your videos are huge.

Your final upload should have the files arranged in this layout:

- <DartmouthID>.zip
  - <DartmouthID>.pdf
  - python/
    - LucasKanade.py
    - LucasKanadeAffine.py
    - InverseCompositionAffine.py
    - test_lk.py
    - test_lk_affine.py
    - test_ic_affine.py
    - file_utils.py
  - ec/
    - LucasKanade_Robust.py *(optional)*
    - LucasKanade_Pyramid.py *(optional)*

# 1 Overview

One incredibly important aspect of human and animal vision is the ability to follow objects and people in our view. Whether it is a tiger chasing its prey, or you trying to catch a basketball, tracking is so integral to our everyday lives that we forget how much we rely on it. In this assignment, you will be implementing an algorithm that will track an object in a video.

You will first implement the Lucas-Kanade tracker (with translation only and with affine transformation), and then a more computationally efficient version called the Matthews-Baker method [1]. This method is one of the most commonly used methods in computer vision due to its simplicity and wide applicability.

To initialize the tracker, you need a template represented by a bounding box around the object to be tracked in the first frame of the video (we have provided this template for all videos). For each of the subsequent frames the tracker will update an affine transform that warps the current frame so that the template in the first frame is aligned with the warped current frame.

For extra credit, we will also look at ways to make tracking more robust, by incorporating illumination invariance and implementing the algorithm in a pyramid fashion.

## 1.1 Preliminaries

An image transformation or warp is an operation that acts on pixel coordinates and maps pixel values from one place to another in an image. Translation, rotation and scaling are all examples of warps. We will use the symbol $\mathbf{W}$ to denote warps. A warp function $\mathbf{W}$ has a set of parameters $\mathbf{p}$ associated with it and maps a pixel with coordinates $\mathbf{x} = [u \;\; v]^T$ to $\mathbf{x}' = [u' \;\; v']^T$.

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

An affine transform is a warp that can include any combination of translation, anisotropic scaling and rotations. An affine warp can be parametrized in terms of 6 parameters $\mathbf{p} = [p_1 \; p_2 \; p_3 \; p_4 \; p_5 \; p_6]^T$. One of the convenient things about an affine transformation is that it is linear; its action on a point with coordinates $\mathbf{x} = [u \;\; v]^T$ can be described as a matrix operation

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \mathbf{W}(\mathbf{p}) \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{2}$$

Where $\mathbf{W}(\mathbf{p})$ is a $3 \times 3$ matrix such that

$$\mathbf{W}(\mathbf{p}) = \begin{bmatrix} 1 + p_1 & p_3 & p_5 \\ p_2 & 1 + p_4 & p_6 \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

Note that for convenience when we want to refer to the warp as a function we will use $\mathbf{W}(\mathbf{x};\mathbf{p})$ and when we want to refer to the matrix for an affine warp we will use $\mathbf{W}(\mathbf{p})$. Table 1 contains a summary of the variables used in the next two sections. It will be useful to keep these in mind.

Table 1: Summary of Variables

| Symbol | Vector/Matrix Size | Description |
|---|---|---|
| $u$ | $1 \times 1$ | Image horizontal coordinate |
| $v$ | $1 \times 1$ | Image vertical coordinate |
| $\mathbf{x}$ | $2 \times 1$ or $1 \times 1$ | pixel coordinates: $(u, v)$ or unrolled |
| $\mathbf{I}$ | $m \times 1$ | Image unrolled into a vector ($m$ pixels) |
| $\mathbf{T}$ | $m \times 1$ | Template unrolled into a vector ($m$ pixels) |
| $\mathbf{W}(\mathbf{p})$ | $3 \times 3$ | Affine warp matrix |
| $\mathbf{p}$ | $6 \times 1$ | parameters of affine warp |
| $\frac{\partial \mathbf{I}}{\partial u}$ | $m \times 1$ | partial derivative of image wrt $u$ |
| $\frac{\partial \mathbf{I}}{\partial v}$ | $m \times 1$ | partial derivative of image wrt $v$ |
| $\frac{\partial \mathbf{T}}{\partial u}$ | $m \times 1$ | partial derivative of template wrt $u$ |
| $\frac{\partial \mathbf{T}}{\partial v}$ | $m \times 1$ | partial derivative of template wrt $v$ |
| $\nabla\mathbf{I}$ | $m \times 2$ | image gradient $\nabla\mathbf{I}(\mathbf{x}) = \left[\frac{\partial \mathbf{I}(\mathbf{x})}{\partial u}\ \frac{\partial \mathbf{I}(\mathbf{x})}{\partial v}\right]$ |
| $\nabla\mathbf{T}$ | $m \times 2$ | image gradient $\nabla\mathbf{T}(\mathbf{x}) = \left[\frac{\partial \mathbf{T}(\mathbf{x})}{\partial u}\ \frac{\partial \mathbf{T}(\mathbf{x})}{\partial v}\right]$ |
| $\frac{\partial \mathbf{W}}{\partial \mathbf{p}}$ | $2 \times 6$ | Jacobian of affine warp wrt its parameters |
| $\mathbf{J}$ | $m \times 6$ | Jacobian of error function $L$ wrt $\mathbf{p}$ |
| $\mathbf{H}$ | $6 \times 6$ | Pseudo Hessian of $L$ wrt $\mathbf{p}$ |

## Lucas-Kanade: Forward Additive Alignment

A Lucas Kanade tracker maintains a warp $\mathbf{W}(\mathbf{x};\mathbf{p})$ which aligns a sequence of images $\mathbf{I}_t$ to a template $\mathbf{T}$. We denote pixel locations by $\mathbf{x}$, so $\mathbf{I}(\mathbf{x})$ is the pixel value at location $\mathbf{x}$ in image $\mathbf{I}$. For the purposes of this derivation, $\mathbf{I}$ and $\mathbf{T}$ are treated as column vectors (think of them as unrolled image matrices). $\mathbf{W}(\mathbf{x};\mathbf{p})$ is the point obtained by warping $\mathbf{x}$ with a transform that has parameters $\mathbf{p}$. $\mathbf{W}$ can be any transformation that is continuous in its parameters $\mathbf{p}$. Examples of valid warp classes for $\mathbf{W}$ include translations (2 parameters), affine transforms (6 parameters) and full projective transforms (8 parameters). The Lucas Kanade tracker minimizes the pixel-wise sum of square difference between the warped image $\mathbf{I}(\mathbf{W}(\mathbf{x};\mathbf{p}))$ and the template $\mathbf{T}$.

In order to align an image or patch to a reference template, we seek to find the parameter vector $\mathbf{p}$ that minimizes $L$, where:

$$L = \sum_{\mathbf{x}} \left[\mathbf{T}\left(\mathbf{x}\right) - \mathbf{I}\left(\mathbf{W}\left(\mathbf{x};\mathbf{p}\right)\right)\right]^2 \tag{4}$$

In general this is a difficult non-linear optimization, but if we assume we already have a close estimate $\mathbf{p}$ of the correct warp, then we can assume that a small linear change $\Delta\mathbf{p}$ is enough to get the best alignment. This is the forward additive form of the warp. The objective can then be written as:

$$L = \sum_{\mathbf{x}} \left[\mathbf{T}\left(\mathbf{x}\right) - I\left(\mathbf{W}\left(\mathbf{x};\mathbf{p}+\Delta\mathbf{p}\right)\right)\right]^2 \tag{5}$$

Using linearization, we end up with the following expression for the *approximate* minimizer $\Delta\mathbf{p}$ of $L$:

$$\Delta\mathbf{p}^* = \mathbf{H}^{-1}\mathbf{J}^T\mathbf{b}, \tag{6}$$

where:

$$\mathbf{H} = \sum_{\mathbf{x}} \left[\nabla\mathbf{I}\frac{\partial\mathbf{W}}{\partial\mathbf{p}}\right]^T \left[\nabla\mathbf{I}\frac{\partial\mathbf{W}}{\partial\mathbf{p}}\right] \tag{7}$$

$$\mathbf{J} = \left[\nabla\mathbf{I}\frac{\partial\mathbf{W}}{\partial\mathbf{p}}\right] \tag{8}$$

$$\mathbf{b} = \mathbf{T}\left(\mathbf{x}\right) - \mathbf{I}\left(\mathbf{W}\left(\mathbf{x};\mathbf{p}\right)\right), \tag{9}$$

and $\nabla\mathbf{I}(\mathbf{x}) = \left[\frac{\partial\mathbf{I}(\mathbf{x})}{\partial u}\,\frac{\partial\mathbf{I}(\mathbf{x})}{\partial v}\right]$ is the vector containing the horizontal and vertical gradient at pixel location $\mathbf{x}$.

Once $\Delta\mathbf{p}$ is computed, the best estimate warp can be updated $\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p}$, and the whole procedure can be repeated again, stopping when $\Delta\mathbf{p}$ is less than some threshold.

## Matthews-Baker Alignment (Inverse Compositional Alignment)

While Lucas-Kanade alignment works very well, it is computationally expensive. The Matthews-Baker method is similar, but requires less computation, as the Hessian $\mathbf{H}$ and Jacobian $\mathbf{J}$ only need to be computed once. One caveat is that the warp needs to be invertible. Since affine warps are invertible, we can use this method.

In the previous section, we combined two warps by simply adding one parameter vector to another parameter vector, and produce a new warp $\mathbf{W}(\mathbf{x}, \mathbf{p}+\mathbf{p}')$. Another way of combining warps is through composition of warps. After applying a warp $\mathbf{W}(\mathbf{x};\mathbf{p})$ to an image, another warp $\mathbf{W}(\mathbf{x};\mathbf{q})$ can be applied to the warped image. The resultant (combined) warp is

$$\mathbf{W}(\mathbf{x};\mathbf{q}) \circ \mathbf{W}(\mathbf{x};\mathbf{p}) = \mathbf{W}(\mathbf{W}(\mathbf{x};\mathbf{p}), \mathbf{q}) \tag{10}$$

Since affine warps can be implemented as matrix multiplications, composing two affine warps reduces to multiplying their corresponding matrices

$$\mathbf{W}(\mathbf{x};\mathbf{q}) \circ \mathbf{W}(\mathbf{x};\mathbf{p}) = \mathbf{W}(\mathbf{W}(\mathbf{x};\mathbf{p}),\mathbf{q}) = \mathbf{W}(\mathbf{W}(\mathbf{p})\mathbf{x},\mathbf{q}) = \mathbf{W}(\mathbf{q})\mathbf{W}(\mathbf{p})\mathbf{x} \tag{11}$$

An affine transform can also be inverted. The inverse warp of $\mathbf{W}(\mathbf{p})$ is simply the matrix inverse of $\mathbf{W}(\mathbf{p})$, $\mathbf{W}(\mathbf{p})^{-1}$. In this assignment it will sometimes be simpler to consider an affine warp as a set of 6 parameters in a vector $\mathbf{p}$ and it will sometimes be easier to work with the matrix version $\mathbf{W}(\mathbf{p})$. Fortunately, switching between these two forms is easy (Equation 3).

The minimization is performed using an iterative procedure by computing a small parameter vector ($\Delta\mathbf{p}$) at each iteration. It is computationally more efficient to do the minimization by finding the $\Delta\mathbf{p}$ that helps align the template to the image, than applying the inverse warp to the image. This is because the image will change with each frame of the video, but the template is fixed at initialization. We will see soon that doing this allows us to write the Hessian and Jacobian in terms of the template, and so this can be computed once at the beginning of the tracking. Hence at each step, we want to find the $\Delta\mathbf{p}$ to minimize

$$L = \sum_{\mathbf{x}} \left[\mathbf{T}\left(\mathbf{W}\left(\mathbf{x};\Delta\mathbf{p}\right)\right) - \mathbf{I}\left(\mathbf{W}\left(\mathbf{x};\mathbf{p}\right)\right)\right]^2 \tag{12}$$

For tracking a patch template, the summation is performed only over the pixels lying inside the template region. Using linearization once agan, we end up with the following expression for the *approximate* minimizer $\Delta\mathbf{p}$ of $L$:

$$\Delta\mathbf{p}^* = \mathbf{H}^{-1}\mathbf{J}^T\left[\mathbf{I}(\mathbf{W}(\mathbf{x};\mathbf{p})) - \mathbf{T}\right] \tag{13}$$

where $\mathbf{J}$ is the Jacobian of $\mathbf{T}(\mathbf{W}(\mathbf{x};\Delta\mathbf{p}))$, $\mathbf{J} = \nabla\mathbf{T}\frac{\partial\mathbf{W}}{\partial\mathbf{p}}$, $\mathbf{H}$ is the approximated Hessian $\mathbf{H} = \mathbf{J}^T\mathbf{J}$, $\mathbf{I}\left(\mathbf{W}\left(\mathbf{x};\mathbf{p}\right)\right)$ is the warped image, and $\nabla\mathbf{T}(\mathbf{x}) = \left[\frac{\partial\mathbf{T}(\mathbf{x})}{\partial u}\ \frac{\partial\mathbf{T}(\mathbf{x})}{\partial v}\right]$. Note that for a given template, the Jacobian $\mathbf{J}$ and Hessian $\mathbf{H}$ are independent of $\mathbf{p}$. This means they only need to be computed once and then they can be reused during the entire tracking sequence.

Once $\Delta\mathbf{p}$ has been solved for, its corresponding warp needs to be inverted and composed with $\mathbf{W}(\mathbf{x};\mathbf{p})$ to get the new warp function for the next iteration.

$$\mathbf{W}(\mathbf{x};\mathbf{p}) \leftarrow \mathbf{W}(\mathbf{x};\mathbf{p}) \circ \mathbf{W}(\mathbf{x};\Delta\mathbf{p})^{-1} \tag{14}$$

The next iteration solves Equation 13 starting with the new value of $\mathbf{p}$. Possible termination criteria include the absolute value of $\Delta\mathbf{p}$ falling below some value or running for some fixed number of iterations.

## 2 Theory Questions (20 points)

**Type down your answers for the following questions in your write-up.** Each question should only take a couple of lines. In particular, the "proofs" do not require any lengthy calculations. If you are lost in many lines of complicated algebra you are doing something much too complicated (or wrong).

### Q2.1: Calculating the Jacobian (10 points)

Assuming the affine warp model defined in Equation 3, derive the expression for the Jacobian Matrix $\mathbf{J}$ in terms of the warp parameters $\mathbf{p} = [p_1\ p_2\ p_3\ p_4\ p_5\ p_6]'$.

### Q2.2: Computational complexity (10 points)

Find the computational complexity (Big O notation) for the initialization step (Precomputing $\mathbf{J}$ and $\mathbf{H}^{-1}$) and for each runtime iteration (Equation 13) of the Matthews-Baker method. Express your answers in terms of $n$, $m$ and $p$ where $n$ is the number of pixels in the template $\mathbf{T}$, $m$ is the number of pixels in an input image $\mathbf{I}$ and $p$ is the number of parameters used to describe the warp $W$. How does this compare to the run time of the regular Lucas-Kanade method?

# 3 Lucas-Kanade Tracker (80 points)

In this section, you will implement two versions of the Lucas-Kanade forward additive alignment (one with the transformation $W$ being the translation only and the other with the $W$ being the full affine transformation) and also the Matthews-Baker inverse compositional alignment with affine transformation. You will run all three algorithms on provided videos, which are provided in the `data` directory. To that end, we have provided script files `test_lk.py`, `test_lk_affine.py` and `test_ic_affine.py` to run three algorithms which can handle reading in images, template region marking, making tracker function calls, displaying output onto the screen and saving results to the disk. As a result, you only need to implement the actual algorithms. The function prototypes provided are guidelines. Please make sure that your code runs functionally with the original script and generates the outputs we are looking for (a frame sequence with the bounding box of the target being tracked on each frame) so that we can replicate your results.

Please include results of three algorithms on all three videos we have provided in your writeup. Specifically, please include results on five frames with an reasonable interval (e.g., for `landing.npy` with only 50 frames in total, you can use an interval of 10 frames and for `car1.npy` with 260 frames, you can use an interval of 50 frames) in your writeup. Also, please describe the difference between three algorithms in terms of performance (e.g., accuracy and speed) and explain why.

### Q3.1: Lucas-Kanade Forward Additive Alignment with Translation (20 points)

Write the function with the following function signature:

```
dx, dy = LucasKanade(It, It1, rect)
```

that computes the optimal local motion represented by only translation (motion in x and y directions) from frame $\mathbf{I}_t$ to frame $\mathbf{I}_{t+1}$ that minimizes Equation 4. Here `It` is the image frame $\mathbf{I}_t$, `It1` is the image frame $\mathbf{I}_{t+1}$, and `rect` is the $4 \times 1$ vector that represents a rectangle on the image frame `It`. The four components of the rectangle are `[x1, y1, x2, y2]`, where `(x1, y1)` is the top-left corner and `(x2, y2)` is the bottom-right corner of the bounding box. To deal with fractional movement of the template, you will need to interpolate the image using the function `RectBivariateSpline` from scipy.interpolate. Also, the `RectBivariateSpline.ev` function can be used to compute the gradient of an image at a point location. You will also need to iterate the estimation in Equation 6 until the change in warp parameters `(dx, dy)` is below a threshold or the number of iterations is too large. To evaluate Equation 6, you can use the function `lstsq` from np.linalg.

### Q3.2: Lucas-Kanade Forward Additive Alignment with Affine Transformation (20 points)

Write the function with the following function signature:

```
M = LucasKanadeAffine(It, It1, rect)
```

Different from Q3.1, here we compute the optimal local motion $M$ represented by an affine transformation with 6 parameters. In other words, the output $M$ should be a 2 × 3 affine transformation matrix.

### Q3.3: Inverse Compositional Alignment with Affine Transformation (20 points)

```
M = InverseCompositionAffine(It, It1, rect)
```

Same with Q3.2, the function should output a 2 × 3 affine matrix so that it aligns the current frame with the template. But you need to implement the inverse compositional alignment following the equations in the preliminaries or the slides from class.

### Q3.4: Testing Your Algorithms (20 points)

Test your three algorithms on the provided video sequence (`car1.npy`), (`car2.npy`) and (`landing.npy`) using our provided scripts. How do your algorithms perform on each video? Which are the differences of three algorithms in terms of performance and why do we have those differences? At what point does the algorithm break down and why does this happen? Remember that the algorithms you are implementing are very basic tracking algorithms so that it might not work at some situations. As long as it is working reasonably (not necessarily perfect), you will receive full credits.

**In your write-up:** Submit your results of three algorithms on all three videos (as mentioned above, please include results on frames with a reasonable interval), analyze your results and answer above questions.

# 4 Extra Credit (20 points)

CS 83—Extra credit (optional)
CS183—Extra credit (optional)

### Q4.1x: Adding Illumination Robustness (10 points)

The LK tracker as it is formulated now breaks down when there is a change in illumination because the sum of squared distances error it tries to minimize is sensitive to illumination changes. There are a couple of things you could try to do to fix this. The first is to scale the brightness of pixels in each frame so that the average brightness of pixels in the tracked region stays the same as the average brightness of pixels in the template. The second is to use a more robust M-estimator instead of least squares (so, e.g. a Huber or a Tukey M-estimator) that does not let outliers adversely affect the cost function evaluation. Note that doing this will modify our least squares problem to a weighted least squares problem, i.e. for each residual term you will have a corresponding weight $\mathbf{\Lambda}_{ii}$ and your minimization function will look like

$$L = \sum_{\mathbf{x}} \mathbf{\Lambda}_{ii} \left[ \mathbf{T}\left(\mathbf{W}\left(\mathbf{x}; \Delta\mathbf{p}\right)\right) - \mathbf{I}\left(\mathbf{W}\left(\mathbf{x}; \mathbf{p}\right)\right) \right]^2 \tag{15}$$

leading your jacobian computation to be a weighted jacobian instead of what you have seen earlier (Eq. 7)

$$A^T \mathbf{\Lambda} A \Delta \mathbf{p} = A^T \mathbf{\Lambda} b \tag{16}$$

$$\Rightarrow \Delta \mathbf{p} = \left(A^T \mathbf{\Lambda} A\right)^{-1} A^T \mathbf{\Lambda} b \tag{17}$$

Here $\mathbf{\Lambda}$ is a diagonal matrix of weights corresponding to the residual term computed as per the choice of the robust M-estimator used, $A$ is the jacobian matrix for each pixel of the template considered in the cost function, and $b$ is the corresponding vector of residuals. Implement these two methods and test your new tracker on the provided video sequences again.

Implement these modifications for the Lucas-Kanade tracker that you implemented for Q3.1. Use the same function names with '`Robust`' appended.

### Q4.2x: LK Tracking on an Image Pyramid (10 points)

If the target being tracked moves a lot between frames, the LK tracker can break down. One way to mitigate this problem is to run the LK tracker on a set of image pyramids instead of a single image. The Pyramid tracker starts by performing tracking on a higher level (smaller image) to get a course alignment estimate, propagating this down into the lower level and repeating until a fine aligning warp has been found at the lowest level of the pyramid (the original sized image). In addition to being more robust, the pyramid version of the tracker is much faster because it needs to run fewer gradient descent iterations on the full scale image due to its coarse to fine approach.

Implement these modifications for the Lucas-Kanade tracker that you implemented for Q3.1. Use the same function names with '`Pyramid`' appended.

## 5 Submission Summary

- **Q2.1** Derive the expression for the Jacobian Matrix
- **Q2.2** What is the computational complexity of Matthews-Baker method?
- **Q3.1** Write the Lucas-Kanade tracker with translation
- **Q3.2** Write the Lucas-Kanade tracker with affine transformation
- **Q3.3** Write the Matthews-Baker tracker with affine transformation
- **Q3.4** Test three algorithms on the provided sequences and analyze the results.
- **Q4.1x** Add illumination robustness
- **Q4.2x** LK Tracking on an image pyramid.

# References

[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

[3] Bouguet, Jean-Yves. Pyramidal Implementation of the Lucas Kanade Feature Tracker: Description of the algorithm, Intel Corporation, 2001