Problem Set 4: Motion Detection

February 20, 2026

ASSIGNMENT DESCRIPTION

Description

Problem Set 4 introduces optic flow as the problem of computing a dense flow field, where a flow field is a vector field <u(x,y), v(x,y)>. We discussed a standard method —Hierarchical Lucas and Kanade —for computing these vectors. This assignment will have you implement methods from simpler operations in order to understand more about array manipulation and the math behind them. We would like you to focus on movement in images and frame interpolation, using concepts that you will learn from modules 6A-6B: Optic Flow.

Learning Objectives

- Implement the Lucas-Kanade algorithm based on the concepts learned from the lectures.
- Learn how pixel movement can be seen as flow vectors.
- Create image resizing functions with interpolation.
- Implement the Hierarchical Lucas-Kanade algorithm.
- Understand the benefits of using a Pyramidal approach.
- Understand the theory of action recognition.

Problem Overview

Methods to be used

In this assignment, you will be implementing the Lucas-Kanade method to compute dense flow fields. Unlike previous problem sets, you will be coding them without using OpenCV functions dedicated to solving this problem.

Consider implementing a GUI (i.e. cv2.createTrackbar) to help you in finding the right parameters for each section.

Rules

You may use image processing functions to find color channels, load images, and find edges (such as with Canny). Don't forget that those have a variety of parameters and you may need to experiment with them. There are certain functions that may not be allowed and are specified in the assignment's autograder Ed Discussion post. Do not use OpenCV functions for finding optic flow or resizing images.

Refer to this problem set's autograder post for a list of banned function calls.

Please do not use absolute paths in your submission code. All paths should be relative to the submission directory. Any submissions with absolute paths are in danger of receiving a penalty!

INSTRUCTIONS

Programming Instructions

Your main programming task is to complete the API described in the file **ps4.py**. The driver program **experiment.py** helps to illustrate the intended use and will output the files needed for the writeup. Additionally there is a file **ps4_test.py** that you can use to test your implementation.

Write-up Instructions

Create **ps4_report.pdf** - a PDF file that shows all your output for the problem set, including images labeled appropriately (by filename, e.g. ps4-1-a-1.png) so it is clear which section they are for and the small number of written responses necessary to answer some of the questions (as indicated). For a guide as to how to showcase your results, please refer to the latex template for PS4.

How to Submit

Two assignments have been created on Gradescope: one for the report - PS4_report, and the other for the code - **PS4_code**.

- Report: the report (PDF only) must be submitted to the PS4_report assignment.
- Code: all files (**ps4.py** and **experiment.py**) must be submitted to the PS4_code assignment. DO NOT upload zipped folders or any sub-folders, please upload each file individually. Drag and drop all files into Gradescope.

Notes

- You can only submit to the autograder **10** times in an hour. You'll receive a message like "You have exceeded the number of submissions in the last hour. Please wait for 36.0 mins before you submit again." when you exceed those 10 submissions. You'll also receive a message "You can submit 8 times in the next 53.0 mins" with each submission so that you may keep track of your submissions.
- If you wish to modify the autograder functions, create a copy of those functions and DO NOT mess with the original function call.

YOU MUST SUBMIT your report and code separately, i.e., two submissions for the code and the report, respectively. Only your last submission before the deadline will be counted for each of the code and the report.

1. Optical Flow [25 points]

In this part you need to implement the basic Lucas Kanade step. You need to create gradient images and implement the Lucas and Kanade optic flow algorithm. Compute the gradients I_x and I_y using the Sobel operator (see cv2.Sobel). Set the scale parameter to one-eighth, ksize to 3, and use the default border type.

Recall that this method solves the following:

$$\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -\sum I_x I_t \\ -\sum I_y I_t \end{bmatrix}$$

The last component we need is I_t which is just the temporal derivative - the difference between the image at time t+1 and t:

$$I_t = I(x, y, t+1) - I(x, y, t)$$

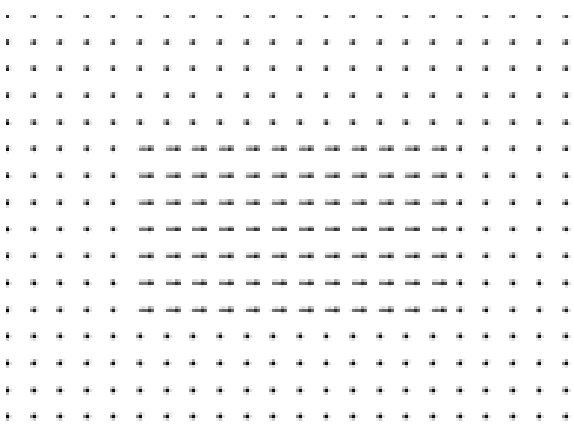
A weighted sum could be computed by just filtering the gradient image (or the gradient squared or product of the two gradients) by a function like a 5x5 or bigger (or smaller!) box filter or smoothing filter (e.g. Gaussian) instead of actually looping. Convolution is just a normalized sum. Additionally, think about what it means to solve for u and v in the equation above. Treat each sum as a component in a 2x2 matrix, and what it means when inverting that matrix. This will be very helpful in order to optimize your code.

1.a.

Write a function *optic_flow_lk()* to perform the optic flow estimation. Essentially, you solve the equation above for each pixel, producing two displacement images U and V that are the X-axis and Y-axis displacements respectively (u(x,y) and v(x,y)).

Show these displacements using a vector or quiver plot, though you may have to scale the values to see the dashes/arrows. An implementation of this function is provided in the utility code section of experiment.py.

For a pair of images that have a static background and a block that presents a movement of 2 pixels to the right at the center, the ideal result would be vector of zero-magnitude in the background and vectors of magnitude = 2 in the center area:



Use the base image labeled as Shift0.png and find the motion that the center block presents in the images ShiftR2.png, and ShiftR5U5.png. You should be able to get a large majority of the vectors pointing in the right direction.

Code:

- gradient_x(image)
- gradient_y(image)
- optic_flow_lk(img_a, img_b, k_size, k_type, sigma)

1.b.

Now try the code comparing the base image Shift0 with the remaining images of ShiftR10, ShiftR20, and ShiftR40, respectively. Remember LK only works for small displacements with respect to the gradients. Try blurring your images or smoothing your results, you should be able to get most vectors pointing in the right direction.

Report: Does LK still work? Does it fall apart in any of the pairs? Try using different parameters to get results closer to the ones above. Describe your results and what you tried.

2. Gaussian and Laplacian Pyramids [20 points]

Recall how a Gaussian pyramid is constructed using the REDUCE operator. Here is the original paper that defines the REDUCE and EXPAND operators: Burt, P. J., and Adelson, E. H. (1983). The Laplacian Pyramid as a Compact Image Code. Here you will also find convolution to help you optimize your code to interpolate the missing pixels. Use this paper as a reference, but follow the lecture content for your implementation.

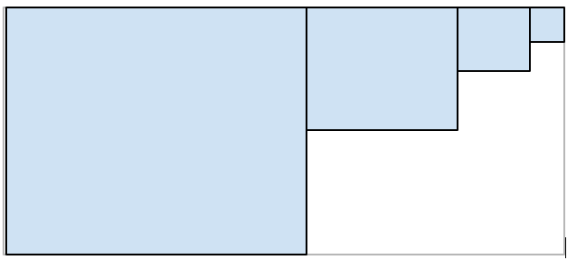
2.a.

Write a function to implement REDUCE, and one that uses it to create a Gaussian pyramid. Use this to produce a pyramid of 4 levels (0-3), applying it to the first frame of the DataSeq1 sequence. Here, you will also complete the function create_combined_img(...) which will output an image that looks like the example below. Normalize each subimage to [0, 255] before copying it into the output array, and use the utility function normalize_and_scale(...).

Code:

- reduce_image(image)
- gaussian_pyramid(image, levels)
- create_combined_img(img_list)

Report: Input: **yos_img_01.png**. Output: the four images that make up the Gaussian pyramid, side-by-side, large to small as **ps4-2-a-1.png**; the combined image should look like:



2.b.

Although the Lucas-Kanade method does not use the Laplacian Pyramid, you do need to expand the warped coarser levels (more on this in a minute). Therefore, you will need to implement the EXPAND operator. Once you have that, the Laplacian Pyramid is just some subtractions.

Write a function to implement EXPAND. Using it, write a function to compute the Laplacian pyramid from a given Gaussian pyramid. Apply it to create the 4-level Laplacian pyramid for the first frame of DataSeq1 (your output will have 3 Laplacian images and 1 Gaussian image).

Code:

- expand_image(image)
- laplacian_pyramid(g_pyr)

Output: - Input: **yos_img_01.png**. Output: the Laplacian pyramid images, side-by-side, large to small (3 Laplacian images and 1 Gaussian image), created from the first image of DataSeq1 as **ps4-2-b-1.png**

3. WARPING BY FLOW [20 POINTS]

The next task is to create a warp function that uses flow vectors to try to revert the apparent motion. This is going to be somewhat tricky. We suggest using the test sequence or some simple motion sequence you create where it's clear that a block is moving in a specific direction. Consider the case where an object in image A moves 2 pixels to the right, as shown in image B. This means that the pixel in B (5,7) is equal to the pixel in A(3,7), where we index using x,y and not row, column. To warp B back to A, create a new image C and set C(x,y) to the value of B(x+2,y). C would then align with A.

Write a function warp() that takes as input an image (e.g. B) and the U and V displacements, and returns a warped image C such that C(x,y)=B(x+U(x,y),y+V(x,y)). Ideally, C should be identical to the original image (A). Note: When writing code, be careful about x, y and rows, columns.

Implementation hints: - The NumPy function meshgrid() might be helpful in creating a matrix of coordinate values, e.g.:

```
A = np.zeros((4, 3))
M, N = A.shape
X, Y = np.meshgrid(range(N), range(M))
```

This produces X and Y such that (X(x,y),Y(x,y))=(x,y). Try printing X and Y to verify this. Now you can add displacement matrices (U,V) directly with (X,Y) to get the resulting locations.

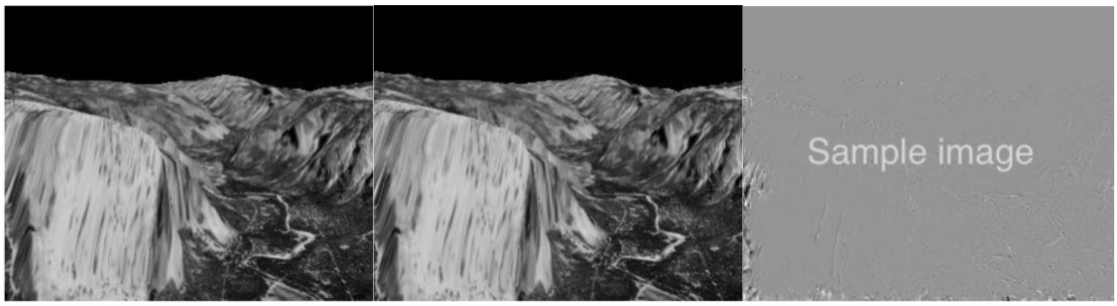
Also, OpenCV has a handy remap() function that can be used to map image values from one location to another. You simply need to provide the image, an X map, a Y map, and an interpolation method.

3.a.

Apply your single-level LK code to the DataSeq1 sequence (from 1 to 2 and 2 to 3). Because LK only works for small displacements, find a Gaussian pyramid level that works best for these.

You will show the output flow fields similar to what you did above and a warped version of image 2 to the coordinate system of image 1. That is, Image 2 is warped back into alignment with image 1. Do the same for images 2 and 3. Create a GIF (http://gifmaker.me/) with these three images to verify your results, you don't need to submit this. You will likely need to use a coarser level in the pyramid (more blurring) to work for this one. If you did this correctly, there should be no apparent motion. Note: For this question you are only comparing images at some chosen level of the pyramid. In the next section you'll do the hierarchy.

Once you have warped these images, you will subtract it from the original. After normalizing and scaling the resulting array, the ideal results should be a gray image with no visible edges. However, with just the single-level LK this may not be the case. Here is a sample output:



Code:

- warp(image, U, V, interpolation, border_mode)

Report:

- Input: **yos_img_01.jpg** and **yos_img_02.jpg**. Output: **ps4-3-a-1.png**
- Input: **yos_img_02.jpg** and **yos_img_03.jpg** . Output: **ps4-3-a-2.png**

4. Optical Flow with LARGE shifts [25 Points]

You may notice that for larger shifts, the Lucas-Kanade by itself fails to record the movement values accurately. Implement the Hierarchical Lucas-Kanade method to overcome this limitation. Complete this code in the hierarchical_lk() function.

4.a.

Compare this method with the single-level LK. Use the base image labeled as Shift0.png and find the motion that the center block presents in the images ShiftR10.png, ShiftR20.png, and ShiftR40.png. You should be able to get better results with this method.

Code:

- hierarchical_lk(img_a, img_b, levels, k_size, k_type, sigma, interpolation, border_mode)

4.b.

Use the Urban2 images to calculate the optic flow between two images. Warp the second image like you did in part 3. Show the flow image and the difference between the original and the

warped one. Reminder: the difference image should have almost no visible edges.

Report: - Input: **urban01.png** and **urban02.png**. Output: **ps4-4-b-1.png** (quiver plot) **ps4-4-b-2.png** (difference image)

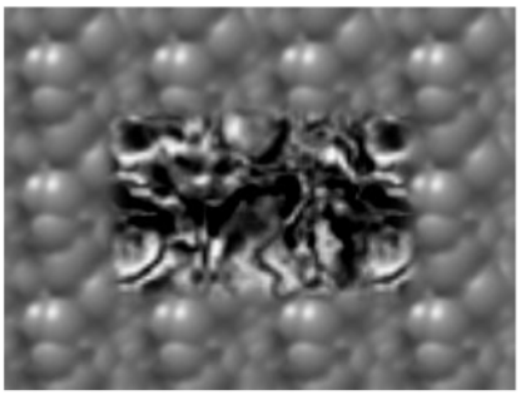
5. Frame Interpolation [10 Points]

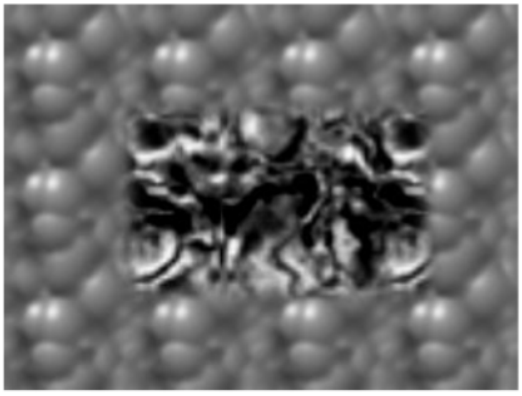
Optic flow can be used in Frame Interpolation (See Szelinski 2010 Section 8.5.1). With Optic Flow principles, we are able to (or at least attempt to) create missing frames. Given that new images are created, you need to obtain the dense optical flow, one vector per pixel. Consider two frames I_0 and I_1, if the same motion estimate u_0 is obtained at location x_0 in image I_0 and is also obtained at location $x_0 + u_0$ in image I_1, the flow vectors are said to be consistent. You will assume the initial flow is the same as the resulting flow. We can generate a third image I_t where $t \in (0,1)$, which will contain a pixel value for the motion vector in question:

$$I_t(x_0 + t * u_0) = (1-t)I_0(x_0) + t * I_1(x_0 + u_0)$$
$$I_t(x_0) = I_0(x_0 - t * u_0)$$

5.a.

You will test this method using two simple images:





Now you will insert 4 new images uniformly distributed between I0 and I1. This means your resulting sequence of images are: I_0, $I_0.2$, $I_0.4$, $I_0.6$, $I_0.8$, I_1 . Verify your results by creating a GIF from these six images. Create an image that contains all the images in the sequence. Organize them in 2 rows and 3 columns. The first row will show I_0, $I_0.2$, $I_0.4$ and the second one $I_0.6$, $I_0.8$, I_1.

The output should be created using a grayscale image. GIF Maker can be used to visualize the output (it is only for visualization purposes).

Report: - Input: **Shift0.png** (I_0) and **ShiftR10.png** (I_1). Output: **ps4-5-a-1.png**

5.b.

The next step is to try this method with real images. For this section, use the files in MiniCooper and insert 4 new images (similar to part a) for each pair of images.

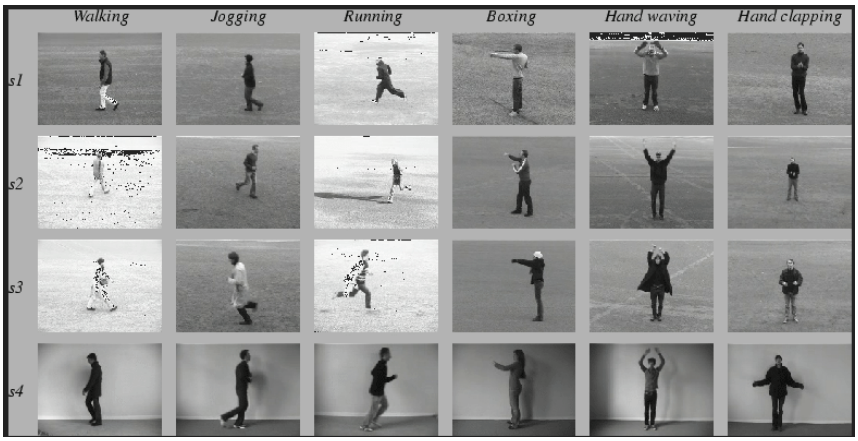
Include all images organized using the same layout as before (2 rows and 3 columns) for each image pair, i.e. (I_0, I_1), (I_1, I_2), etc.

Notice this method produces a great amount of artifacts in the resulting images. Use what you have learned so far to reduce them in order to create a smoother sequence of frames.

The output should be a grayscale image.

Report:
- Input: **mc01.png** (I_0) and **mc02.png** (I_1). Output: **ps4-5-b-1.png**
- Input: **mc02.png** (I_1) and **mc03.png** (I_2). Output: **ps4-5-b-2.png**

6. Challenge Problem (Extra Credit) - [20 Points]



Finally, for our challenge questions, we will consider the application of optical flow as a means of identifying specific actions. For example: how can you tell if someone is running or walking? Obviously, there are many different ways to do classification. Here, we simply ask that, given the motion analysis from optical flow, you write an algorithm (preferably a simple, rule-based algorithm) that classifies whether each video represents one of the three following classes:

- Running
- Walking
- Clapping

Where a "1" is the class output for "running", "2" is the class output for "walking", and "3" is the class output for clapping. The velocity and direction information should be ample to identify each. You do not have to be 100% correct on these videos, credit will be given for any result that is better than random (3 or more out of 6 correct). You don't have to use every frame the video, just as many as you need to classify the action. We will check to make sure that you aren't using metadata, file ordering, or any other means besides optical flow to classify the actions, and points will only be awarded if the optical flow is the only input to the classifier.

For reading in the video, we will use the read_video() function and pass the output of this function to the function you will have to write: classify_video().

If you don't want to use your own rule-based classifier, consider using the sklearn.DecisionTreeClassifier() from Scikit-Learn, but don't spend too much time on this. This is not a machine learning question!

Note: Apple computers might face an issue running cv2.VideoCapture(video_file). It might not work with .avi file, the solution is to transcode them to .m4v files and then proceed.

Data

- person01_running_d2_uncomp.avi
- person08_running_d4_uncomp.avi
- person04_walking_d4_uncomp.avi
- person06_walking_d1_uncomp.avi
- person14_handclapping_d3_uncomp.avi
- person10_handclapping_d4_uncomp.avi

Code:

- classify_video(images) -> int

Report:

- Output: For every video, provide the class and confidence (if the method gives it), and explain in depth what you did to attempt this problem.

- Discussion Question: Optical flow is only one approach to activity recognition and classification. Many modern methods do not require optical flow at all, but simply learn to classify actions directly from the pixels. Can you think of a situation in which calculating the optical flow would be a vital part of action classification? What about a situation in which optical flow might be unhelpful in action classification?