HW2: Optical Flow and Video Stabilization

Due date: See in Moodle

Section 1 – General Questions

In this part you will need to answer a few questions. Keep your answers short. Sometimes there is more than one correct answer, so please explain your answer. All answers in part 1 should be submitted in a PDF file. The first line of the PDF should include your IDs.

Let's recall the original optical flow formula:

$$0 = I_t + I_x u + I_y v$$

Note that for each pixel we have two unknowns (u and v), and only one equation.

1. What is the constant brightness constraint? In you answer relate to:
    a. How does it help us to solve the optical flow between two images?
    b. Is this assumption correct in real world scenarios? (what happens when there are reflective objects in the image?).
2. What is the aperture problem and what can we do about it?
3. How did Lucas-Kanade solve the optical flow problem? (what did they assume about the movement of each pixel?). Hint: it is related to the movement of the neighbourhood of each pixel.
4. Is the Lucas-Kanade assumption true around the object boundaries? Why?
5. Propose a general idea how to correctly find the optical flow on the object's boundaries, given you can get any input you desire (except from the true movement of each pixel). For example, you can get a depth map / label image (you know for each pixel to which object it belongs to or what is its depth). Write which inputs you assume to have and the general idea of your solution.

Section 2 – Lucas-Kanade Optical Flow

Part 2.1 - Optical Flow one Step

In this part you are required to compute the velocity fields (u,v) between images I1 and I2. Using those values, you are required to back-project from I2 to I1 (the result should be very similar to I1, in the sense that there should be little movement between the two frames).

The Optical Flow (velocity field) will be computed using the Lucas-Kanade Algorithm (as you learned in class). For this assignment you will use a 2D-translation warp.

In this part, we run the **main_river.py** file and implement functions in the **lucas_kanade.py** file.

I1 in this part is loaded from **river1.png**

I2 in this part is loaded from **river2.png**

Note that there are constants in the head of the main_river.py that you need to finetune / edit.

1. Implement the function **build_pyramid** in **lucas_kanade.py.** The function signature is:

   An image pyramid is a list containing downsampled versions of the original image. Here, we first create a list with a copy of the original image. Then, iterate over the levels. In each level, we convolve the PYRAMID_FILTER with the image from the previous level. Then, we decimate the result using indexing: simply pick every second entry of the result. Finally, we append the filtered->decimated result to the end of the list. The list length should be num_levels+1.

   Hint: Use signal.convolve2d with boundary='symm' and mode='same'.

   You are not allowed to use cv2 PyrDown here (or any other cv2 method), since we use a slightly different decimation process from this function.

build_pyramid		Type and shape	Description
Inputs			
	image	np.ndarray array of shape: (h, w).	input image.
	num_levels	int	The number of times we will apply the filter on the image
Output:			
	pyramid	A list of np.ndarray of images. The list length should be num_levels+1	Image pyramid is a list of images. The first image is a copy of the original image. The second image is a decimated version of the first image after we applied a filter to it. And so on.

2. Implement the function **lucas_kanade_step** in **lucas_kanade.py.**

   This method receives two images as inputs and a window_size. It calculates the per-pixel shift in the x-axis and y-axis. That is, it outputs two maps of the shape of the inupt images. The first map encodes the per pixel optical flow parameters in the x-axis and the second in the y-axis.

   Open the file for further documentation.

lucas_kanade_step		Type and shape	Description
Inputs			
	I1	np.ndarray array of shape: (h, w).	Image at time t.
	I2	np.ndarray array of shape: (h, w).	Image at time t+1.
	window_size	int	The window is of size: window_sizeXwindow_size.
Output:			
	(du, dv)	tuple of two np.ndarray-s. Each np.ndarray is of shape (h, w).	Each one is of the shape of the original image. dv encodes the optical flow parameters in rows and du in columns.

3. Implement the function **warp_image** in **lucas_kanade.py.** This function back projects the image I2 using u and v. The expected result should be very similar to I1.

warp_image		Type and shape	Description
Inputs			
	image	np.ndarray array of shape: (h, w).	Image at time t+1.
	u	np.ndarray : 2d array	Optical flow parameters corresponding to the columns.
	v	np.ndarray : 2d array	Optical flow parameters corresponding to the rows.
Output:			
	image_warp	np.ndarray array of shape: (h, w).	Warped image

For a reason which will be clearer later, this method needs to support the case where u and v shapes do not share the same shape as of the image. We will update u and v to the shape of the image. The way to do it, is to:

1. cv2.resize to resize the u and v to the shape of the image.
2. Then, normalize the shift values according to a factor. This factor is the ratio between the image dimension and the shift matrix (u or v) dimension (the factor for u should take into account the number of columns in u and the factor for v should take into account the number of rows in v).

As for the warping, use `scipy.interpolate`'s `griddata` method:

1. Define the grid-points using a flattened version of the `meshgrid` of 0:w-1 and 0:h-1.
2. The values here are simply image.flattened().
3. The points you wish to interpolate are, again, a flattened version of the `meshgrid` matrices - don't forget to add them v and u.
4. Use `np.nan` as `griddata`'s fill_value.

Finally, fill the nan holes with the source image values.

Hint: For the final step, use np.isnan(image_warp).

4. Now, go to the **main_river.py**. Run the script up to the comment line:

```
##########################################################################################
######################### ONE STEP LUCAS KANADE ENDS HERE ################################
##########################################################################################
```

4.1. Put the image created in river_results/0_river_one_LK_step_result.png in your PDF and explain the result.

4.2 Look at the gifs created in **river_results/2_after_one_lk_step.gif** and **river_result/1_original.gif** and answer the following question in the report (PDF): Why is the result imperfect?

5. Implement the function **lucas_kanade_optical_flow** in **lucas_kanade.py**. This function calculates the LK Optical Flow for max iterations in num-levels of an image pyramid. See detailed algorithm flow after this table.

lucas_kanade_optical_flow		Type and shape	Description
Inputs			
	I1	np.ndarray array of shape: (h, w).	Image at time t.
	I2	np.ndarray array of shape: (h, w).	Image at time t+1.
	window_size	int	The window is of size: window_sizeXwindow_size.
	max_iter	int	Maximal number of LK-steps for each level of the pyramid.
	num_levels	int	Number of pyramid levels.
Output:			
	(u, v)	tuple of two np.ndarray-s. Each np.ndarray is of shape (h, w).	v encodes the optical flow parameters in rows and u in columns.

1. Since the image is going through a series of decimations, we would like to resize the image shape to:

   [K * (2^(num_levels - 1))] X [M * (2^(num_levels - 1))]. Where:

   a. K is the ceil(h / (2^(num_levels - 1)),
   b. and M is ceil(h / (2^(num_levels - 1)).

   You can use cv2.resize here.

2. Build pyramids for the two images.
3. Initialize u and v as all-zero matrices in the shape of I1.
4. For every level in the image pyramid (start from the smallest image and finish in the full size image):
   a. Warp I2 from that level according to the current u and v.
   b. Repeat for num_iterations:
      i. Perform a Lucas Kanade Step with the I1 decimated image of the current pyramid level and the current I2_warp to get the new I2_warp.
   c. For every level which is not the image's level, perform an image resize (using cv2.resize) to the next pyramid level resolution (bigger image) and scale u and v accordingly.

6. Run the entire **main_river.py** script. Put all png outputs in your PDF report. Explain all results. Explain why your gif **3_after_full_lk.gif** looks better now.

Section 3 – Video Stabilization

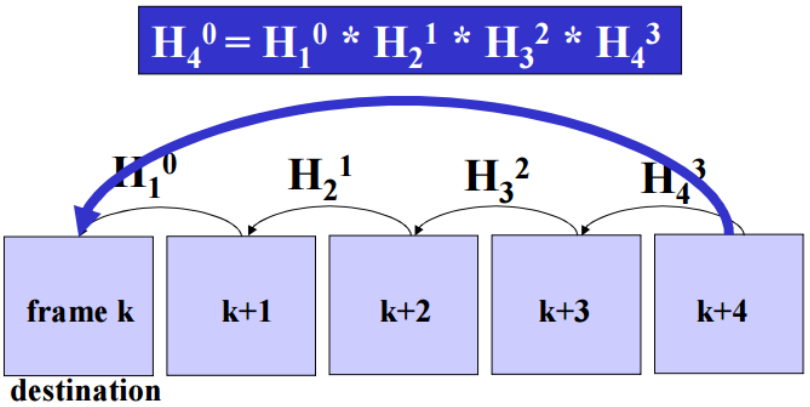
Part 3.1:

In this part you are required to stabilize a video file using Lucas-Kanade Optical Flow. We will use the functions from Section to do that.

Given a video, for each frame compute the LK Optical Flow with respect to the previous frame (and the destination frame).

For example, suppose you have warped frame k to frame 1. Now, compute the warp from frame k+1 to frame k, add the warp you computed from frame k to frame 1 and use the combined warp, to warp frame k+1 to the coordinate system of frame 1.
Frame 1 should be copied as is to the output video.

Graphical representation:



We will continue to implement functions in the **lucas_kanade.py** file. The file we will run in this section is **main_tau_video.py**. We will process the video **input.avi**.

We will have 3 output videos as a result of running this section:

ID1_ID2_stabilized_video.avi

ID1_ID2_faster_stabilized_video.avi

ID1_ID2_fixed_borders_stabilized_video.avi

7. Implement the function **lucas_kanade_video_stabilization** in **lucas_kanade.py**. This function uses LK Optical Flow to stabilize the video and save it to file.

   Detailed explanation follows this table.

lucas_kanade_video_stabilization		Type and shape	Description
Inputs			
	input_video_path	str	path to input video.
	output_video_path	str	path to output stabilized video.
	window_size	int	The window is of size: window_sizeXwindow_size.
	max_iter	int	Maximal number of LK-steps for each level of the pyramid.
	num_levels	int	Number of pyramid levels.
Output:			
	None		
Effect	Writes the stabilized video to output_video_path		

Steps:

1. Open a VideoCapture object of the input video and read its parameters.
2. Create an output video VideoCapture object with the same parameters as in (1) in the path given here as input.
   a. **Use fourcc = cv2.VideoWriter_fourcc(*'XVID')**
3. Convert the first frame to grayscale and write it as-is to the output video.
4. Resize the first frame as in the Full-Lucas-Kanade function to

K * (2^(num_levels - 1)) X M * (2^(num_levels - 1)).

Where: K is the ceil(h / (2^(num_levels - 1)),

and M is ceil(h / (2^(num_levels - 1)).

5. Create a u and a v which are og the size of the image.
6. Loop over the frames in the input video (use tqdm to monitor your progress) and:
   a. Resize them to the shape in (4).
   b. Feed them to the lucas_kanade_optical_flow with the previous frame.
   c. Use the u and v maps obtained from (6.2) and compute their mean values over the region that the computation is valid (exclude half window borders from every side of the image).
   d. Update u and v to their mean values inside the valid computation region.
   e. Add the u and v shift from the previous frame diff such that frame in the t is normalized all the way back to the first frame.

f. Save the updated u and v for the next frame (so you can perform step 6.e. for the next frame.
g. Finally, warp the current frame with the u and v you have at hand.
h. We highly recommend you to save each frame to a directory for your own debug purposes. Erase that code when submitting the exercise.

7. Do not forget to gracefully close all VideoCapture and to destroy all windows.

8. Run **main_tau_video.py** to create ID1_ID2_stabilized_video.avi. Explain the result you obtained. What happened to the MSE of the video frames? It is ok if you do not handle border effects as we will take care of them by the end of the exercise. This means that the video may contain frames such as this:

But, other than that - the video should look stabilized (not still but stable).

The MSE between frames should be low relative to the original video MSE (lower by at least 40%).

Our numbers are (for your validation):

Mean MSE between frames for original video: 60.23

Mean MSE between frames for Lucas Kanade Stabilized output video: 26.73

Yours can be better (but we do not expect 0 MSE, right? [If worse, not worse than the margin noted up here in percentage])

Part 3.2: Faster LK Implementation

9. Implement the function **faster_lucas_kanade_step** in **lucas_kanade.py**. This function implements LK Optical Flow to two images. This function should run faster than **lucas_kande_step** from Section 2.

   The main trick of this function is to compute u and v only in interest points (corners) when the pyramid resolution is big enough. That means that for small sized levels of the image pyramid the LK-step will be calculated as before but for higher levels it will be computed only for corners.

   You can choose if you'd like to use your Harris corner detector from the first exercise or opencv's cv2.cornerHarris.

   Detailed explanation follows this table.

faster_lucas_kanade_step		Type and shape	Description
Inputs			
	I1	str	Image at time t.
	I2	str	Image at time t+1.
	window_size	int	The window is of size: window_sizeXwindow_size.
Output:			
	(du, dv)	tuple of two np.ndarray-s. Each np.ndarray is of shape (h, w).	Each one is of the shape of the original image. dv encodes the optical flow parameters in rows and du in columns.

Follow the steps:

(1) If the image is small enough (you need to design what is good enough), simply return the result of the good old lucas_kanade_step function.

(2) Otherwise, find corners in I2 and calculate u and v only for these pixels.

(3) Return maps of u and v which are all zeros except for the corner pixels you found in (2).

10. Implement **faster_lucas_kanade_optical_flow** this should be a copy of **lucas_kanade_optical_flow** but replace the call for **lucas_kanade_step** with a call to: **faster_lucas_kanade_step**.
11. Implement **lucas_kanade_faster_video_stabilization** this should be a copy of **lucas_kanade_video_stabilization** but replace the call for **lucas_kanade_optical_flow** with a call to: **faster_lucas_kanade_optical_flow**.

Use *fourcc = cv2.VideoWriter_fourcc(*'XVID')* for the output video.

Again, it is OK if you have border effects as in the first video you created.

The runtime should decrease by at least 30%. The MSE between frames should be low relative to the original video MSE (lower by at least 30%).

For your validation, these are our numbers:

Mean MSE between frames for original video: 60.23

Mean MSE between frames for Lucas Kanade Stabilized output video: 26.73

Mean MSE between frames for Lucas Kanade Stabilized output FASTER Implementation video: 29.03

Yours can be better (but we do not expect 0 MSE, right? [If worse, not worse than the margin noted up here in percentage])

12. Implement **lucas_kanade_faster_video_stabilization_fix_effects**. This function fixes border effects of the Lucas Kanade implementation by cutting out a constant portion of the image and uses the **faster_lucas_kanade_optical_flow.**

Use *fourcc = cv2.VideoWriter_fourcc(*'XVID')* for the output video.

lucas_kanade_faster_video_stabilization_fix_effects		Type and shape	Description
Inputs			
	input_video_path	str	path to input video.
	output_video_path	str	path to output stabilized video.
	window_size	int	The window is of size: window_sizeXwindow_size.
	max_iter	int	Maximal number of LK-steps for each level of the pyramid.
	num_levels	int	Number of pyramid levels.
	start_rows	int	The number of lines to cut from top
	start_cols	int	The number of lines to cut from bottom.
	end_rows	int	The number of columns to cut from left.
	end_cols	int	The number of columns to cut from right.
Output:			
	None		
Effect	Writes the stabilized video to output_video_path.		

For your validation, these are our numbers:

Mean MSE between frames for original video: 60.23

Mean MSE between frames for Lucas Kanade Stabilized output FASTER Implementation + BORDERS CUT video: 24.39

Yours can be better (but we do not expect 0 MSE, right? [If worse, not worse than the margin noted up here in percentage]).

General Notes

- In all parts, you may add functions as you please, which will be called from the functions you were given.
- Don't change the main files (**main_tau_video.py** and **main_river.py**). The only things that you need to do is determine the values of the window-size, number of iterations and pyramid level parameters. Choose values that give the best result. We will test your functions from your main file.
- If creating each video takes more than an hour on Tochna Computers - points will be decreased.
- Files that you need to submit:
    - PDF file (**ex2_ID1_ID2.pdf**) with answers to questions which are not programming.
        - The first line in this document should contain your ID2.
    - **ID1_ID2_stabilized_video.avi**
    - **ID1_ID2_faster_stabilized_video.avi**
    - **ID1_ID2_fixed_borders_stabilized_video.avi**
    - **lucas_kanade.py**
    - **main_river.py**
    - **main_tau_video.py**

Any compilation errors of any sort equal zero points for that respective question. Please double check your solutions before submitting them!

ALL FILES WILL BE PLACED IN ONE ZIP FILE CALLED: **vp2022_ex2_ID1_ID2.zip** , where ID1&ID2 should be your ID's.

Only one of each student pair should submit the HW.

Enjoy and Good Luck! ☺