Problem Set 6: Optic Flow

Description

We discussed 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 this field. The basic idea is to compute a single translational vector over a window centered about a point that best predicts the change in intensities over that window by looking at the image gradients. For this problem set you will implement the necessary elements to compute the LK optic flow field. This will include the necessary functions to create a Gaussian pyramid.

Some sequences are provided in the input directory. First, there is a test sequence called `TestSeq` that just translates a textured rectangle over a textured background. In that directory you will find the images `Shift0, ShiftR2, ShiftR10, ShiftR20, ShiftR40` and `ShiftR5U5`. You need to use these images to test your code and to see the effect of the hierarchy. `DataSeq1` has 3 frames of the so-called *"Yosemite"* sequence and `DataSeq2` has larger displacements. Also, there is an extra credit sequence called `Juggle`.

Reading: Burt, P. J., and Adelson, E. H. (1983). The Laplacian Pyramid as a Compact Image Code (question 2)

What to submit

Download and unzip: ps6.zip

`ps6/`

- input/ - input images, videos or other data supplied with the problem set
- output/ - directory containing output images and other files your code generates
- ps6.py - code for completing each part, esp. function calls
- *.py - Python modules, any utility code
- ps6_report.pdf - a PDF file with all output images and text responses

Zip it as `ps6.zip`, and submit on T-Square.

Guidelines

1. Include all the required images in the report to avoid penalty.
2. Include all the textual responses, outputs and data structure values (if asked) in the report.
3. Make sure you submit the correct (and working) version of the code.
4. Include your name and GTID on the report.
5. Even if the code is not working, submit the code as the instructors can read through the algorithms to give partial credit. Comment your code appropriately.

Questions

1. Lucas Kanade Optic Flow

In this part, you need to implement the basic LK step. You need to write code to create gradient images and implement the Lucas and Kanade optic flow algorithm. Recall that we compute the gradients I_x and I_y and then over a window centered around each pixel we solve the following:

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

Remember 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.

a. Write a function `optic_flow_LK()` to do 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)`). Save the images with a false-color colormap (e.g. Jet) so that the displacement across each image is visible clearly.
- Pseudocolor plotting tutorials: OpenCV, matplotlib
Note: In this case, combine U and V side-by-side to make a composite twice-wide image.

You can also show these displacements using a vector or *quiver* plot, though you may have to scale the values to see the dashes/arrows.
- MATLAB has a built-in `quiver` function which does this - you can refer to it to see what your output should look like.
- Matplotlib quiver plot tutorials: matplotlib 1, matplotlib 2 (check image aspect ratio and Y-axis direction)
- OpenCV code snippet (you may have to adapt this to work correctly):

```
stride = 25  # plot every so many rows, columns
scale = 2  # scale up vector lengths by this factor
color = (0, 255, 0)  # green
img_out = np.zeros((V.shape[0], U.shape[1], 3), dtype=np.uint8)
for y in xrange(0, V.shape[0], stride):
    for x in xrange(0, U.shape[1], stride):
        cv2.line(img_out, (x, y), (x + int(U[y, x] * scale), y +
int(V[y, x] * scale)), color, 1)
```

`TestSeq` has images of a smoothed image texture with a different texture center rectangle displaced by a set number of pixels. `Shift0` is the "base" image; images listed as `ShiftR2` have the center portion shifted to the right by 2 pixels; `ShiftR5U5` have the center portion shifted to the right by 2 pixels and up 5, etc.

When you try your code on the images `TestSeq` you should get a simple translating rectangle

in the middle and everything else zero. Try your LK on the base image and the `ShiftR2` and between the base image and `ShiftR5U5`. Remember LK only works for small displacements with respect to the gradients so you might have to smooth your images a little to get it to work; try to find a blur amount that works for both cases but keep it as little as possible.

Function: `optic_flow_LK(A, B) -> U, V`

Output: Show U and V (the X and Y displacements) either as side-by-side false-color image, or as a quiver plot, when computing motion between:
- the base `Shift0` and `ShiftR2` as `ps6-1-a-1.png`
- the base `Shift0` and `ShiftR5U5` as `ps6-1-a-2.png`

Output (textual response):
- If you blur (smooth) the images say how much you did

b. Now try the code comparing the base image `Shift0` with the remaining images of `ShiftR10`, `ShiftR20` and `ShiftR40`. Use the same amount of blurring as you did in the previous section. Does it still work? Does it fall apart on any of the pairs?

Output: Show U and V (the X and Y displacements) either as side-by-side false-color image, or as a quiver plot, when computing motion between:
- the base `Shift0` and `ShiftR10` as `ps6-1-b-1.png`
- the base `Shift0` and `ShiftR20` as `ps6-1-b-2.png`
- the base `Shift0` and `ShiftR40` as `ps6-1-b-3.png`

Output (textual response):
- Describe your results.

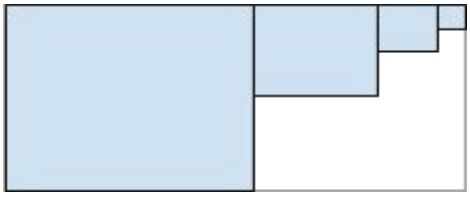
2. Gaussian and Laplacian Pyramids

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

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 `DataSeq1` sequence.

Functions:
- `reduce(image) -> reduced_image`
- `gaussian_pyramid(image, levels) -> g_pyr`

Output:
- the 4 images that make up the Gaussian pyramid, side-by-side, large to small as `ps6-2-a-1.png`; the combined image should look like:



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` *(which has 3 Laplacian images and 1 Gaussian image)*.

Functions:
- `expand(image) -> expanded_image`
- `laplacian_pyramid(g_pyr) -> l_pyr`

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 `ps6-2-b-1.png`

3. Warping by flow

Next you'll try it on the two Data sequences. You'll need to determine a level of the pyramid where it seems to work. To test your results you should use the recovered motion field to *warp* the second image back to the first (or the first to the second).

The challenge in this is to create a warp function and then use it correctly. This is going to be somewhat tricky. I suggest you try and use the test sequence or some simple motion sequence you create where it's clear that a block is moving in a specific direction. The first question is - are you recovering the amount you need to move to bring the second image to the first or the first to the second. If you look carefully at the slides you'll see we're solving for the amount that is the change from *A* to *B* . Consider the case where the image moves 2 pixels to the right. This means that $B(5,7) = A(3,7)$ where I am indexing by x,y and not by row and column. So, to warp *B* back to *A* to create a new image *C*, would 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 X and Y displacements (U, V), 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.

Think: Besides differences due to image variations, in which regions would you expect to see differences between A & C? Are there any regions in C where values are unknown or missing?

Function: `warp(image, U, V) -> warped`

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(xrange(N), xrange(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 displacements 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.

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 the best for these. To show that it works, you will show the output flow fields as above and you'll show a warped version of image 2 to the coordinate system of image 1. That is, you'll warp Image 2 back into alignment with image 1. If you flicker (rapidly show them back and forth) the warped image 2 and the original image 1, you should see almost no motion: the second image pixels have been "moved back" to the location they came from in the first image. Same is true for image 2 to 3. Next, try this for `DataSeq2` where the displacements are greater. You will likely need to use a coarser level in the pyramid (more blurring) to work for this one**.**
Note: For this question you are only comparing between images at some chosen level of the pyramid! In the next section you'll do the hierarchy.

Output:
- the images showing the x and y displacements for `DataSeq1` *(either as images or as arrows)* `as ps6-3-a-1.png`
- show the *difference image* between warped image 2 and original image 1 for DataSeq1 `as ps6-3-a-2.png` (zero difference should map to neutral gray, max -ve to black, max +ve to white)
- the images showing the x and y displacements for `DataSeq2` *(either as images or as arrows)* `as ps6-3-a-3.png`
- show the *difference image* between warped image 2 and original image 1 for DataSeq2 `as ps6-3-a-4.png` (zero difference should map to neutral gray, max -ve to black, max +ve to white)

4. Hierarchical LK optic flow

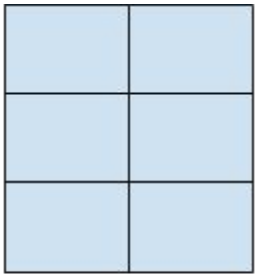
Recall the basic steps of the hierarchical method:
1. Given input images A and B. Initialize k = n where n is the max level.
2. REDUCE both input images to level k. Call these images A_k and B_k.
3. If k = n initialize U and V to be zero images the size of A_k; otherwise expand the flow field and double to get to the next level: U = 2*EXPAND(U), V = 2* EXPAND(V).
4. Warp B_k using U and V to form C_k.
5. Perform LK on A_k and C_k to yield two incremental flow fields D_x and D_y.
6. Add these to original flow: U = U + D_x and V = V + D_y.
7. If k > 0 let k = k – 1 and goto (2).
8. Return U and V

When developing this code you should try it on `TestSeq` between the base and the larger shifts, or you might try and create some test sequences of your own. Take a textured image and displace a center square by a fixed translation amount. Vary the amount and make sure your hierarchical method does the right thing. In principle, for a displacement of δ pixels, you'll need to set n to (at least) $\log_2(\delta)$.

Function: `hierarchical_LK(A, B) -> U, V`

a. Write the function `hierarchical_LK()` to compute the hierarchical LK optic flow. Apply it to `TestSeq` for the displacements of 10, 20 and 40 pixels (`ShiftR10.png, ShiftR20.png` and `ShiftR40.png`).

Output:
- the displacement images between *B* and the original *A* for each of the cases; create a stacked side-by-side false-color image showing these results together as `ps6-4-a-1.png`
- the difference images between the warped *B* and the original *A* for each of the cases; create a stacked image showing these results together as `ps6-4-a-2.png`



Displacement image pairs (U, V), stacked

Difference images, stacked

b. Apply your function to `DataSeq1` for the images `yos_img_01.jpg, yos_img_02.jpg` and `yos_img_03.jpg` .

Output:
- the displacement images between *B* and the original *A* for each of the cases; create a stacked side-by-side false-color image showing these results together as `ps6-4-b-1.png`
- the difference images between the warped *B* and the original *A* for each of the cases; create a stacked image showing these results together as `ps6-4-b-2.png`

c. Apply your function to `DataSeq2` for the images `0.png, 1.png and 2.png`.

Output:
- the displacement images between *B* and the original *A* for each of the cases; create a stacked side-by-side false-color image showing these results together as `ps6-4-c-1.png`
- the difference images between the warped *B* and the original *A* for each of the cases; create a stacked image showing these results together as `ps6-4-c-2.png`

5. The sequence `Juggle` has significant displacement between frames — the juggled balls move significantly. Try your hierarchical LK on that sequence and see if you can warp frame 2 back to frame 1.

a. Apply your hierarchical LK to the `Juggle` sequence.

Output:
- the displacement images between *B* and the original *A* for each of the cases
create a side-by-side false-color image showing these results in one image as `ps6-5-a-1.png`
- the difference images between the warped *B* and the original *A* for each of the cases
create a side-by-side false-color image showing these results in one image as `ps6-5-a-2.png`