Exercise 3: Pyramids and Optic Flow

Due date: DD/MM/YYYY

In the lectures we discussed optic flow, in this exercise you will implement the Lucas Kanade algorithm.

All functions should be able to accept both gray-scale and color images.

In the optical-flow methods, you should accept a color image, but work on the gray-scale copy of the image.

1 Lucas Kanade Optical Flow

Write a function which takes an image and returns the optical flow by using the LK algorithm.

```
def opticalFlow(im1: np.ndarray, im2: np.ndarray, step_size=10,
            win_size=5) -> (np.ndarray, np.ndarray):
    """
    Given two images, returns the Translation from im1 to im2
    :param im1: Image 1
    :param im2: Image 2
    :param step_size: The image sample size
    :param win_size: The optical flow window size (odd number)
    :return: Original points [[x,y]...], [[dU,dV]...] for each points
    """
```

In order to compute the optical flow, you will first need to compute the gradients I_x and I_y and then over a window centered around each pixel we calculate

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

Remember (1): the checks for the $\lambda_{1/2}$:

- $\lambda_1 \geq \lambda_2 > 1$
- $\frac{\lambda_1}{\lambda_2} < 100$

For some points you will not be able to get a good optical flow, meaning they won't pass the λ constraints. That why you need to return only the points that are good, since only they will have a matching $\Delta X, \Delta y$.

Remember (2): LK algorithm is used for sub-pixel movements! The best way to check your self is to take an image, apply a transformation (i.e. translation) and test

Use the method *displayOpticalFlow* to look at the results, you can also check the mean of the $\Delta X, \Delta Y$ to see that it matches the initial transformation. Even if you had a simple translation, not all points will give the same optical flow, there will be some outliers (*"I accept chaos, I'm not sure whether it accepts me." — Bob Dylan*)

2 Hierarchical Lucas Kanade optical flow

The point of the exercise is to capture large movements using OpticalFlow.
The steps are as follows:

- The input of all the above functions can be either color or gray-scale images.
- Create pyramids with depth of k out of both of the images (image1, image2).
- Find the optical flow of the smallest images in the pyramid (U,V) **using iterative method**.
- Move upwards in the pyramid while adding the the current optical flow (U_i, V_i) the optical flow from the previous layer, but double the U,V as follows: $U_i = U_i + 2 * U_{i-1}, V_i = V_i + 2 * V_{i-1}$.
- Return the U_k, V_K.

```
def opticalFlowPyrLK(img1: np.ndarray, img2: np.ndarray, k: int,
                     stepSize: int, winSize: int) -> np.ndarray:
    """
    :param img1: First image
    :param img2: Second image
    :param k: Pyramid depth
    :param stepSize: The image sample size
```

```
:param winSize: The optical flow window size (odd number)
:return: A 3d array, with a shape of (m, n, 2),
where the first channel holds U, and the second V.
"""
```

3 Image Alignment And Warping

In this section you should find the parameters alignment between 2 input images: im_1, im_2, where im_2 was received after performing parametric motion of im_1.
Finding the parameters should be done by assuming a specific form of transformation and minimizing the error function. You should implement two methods for finding the parameters: Lucas-Kanade and Correlation. The functions should have the following interface, according to the assumed transformation and required method:

1. **Translation** :

   ```
   findTranslationLK(im1: np.ndarray, im2: np.ndarray) -> np.ndarray:
   ```

2. **Rigid (translation+rotation)** :

   ```
   findRigidLK(im1: np.ndarray, im2: np.ndarray) -> np.ndarray:
   ```

3. **Translation** :

   ```
   findTranslationCorr(im1: np.ndarray, im2: np.ndarray) -> np.ndarray:
   ```

4. **Rigid (translation + rotation)** :

   ```
   findRigidCorr(im1: np.ndarray, im2: np.ndarray) -> np.ndarray:
   ```

5. **Image Warpping** :

   ```
   def warpImages(im1: np.ndarray, im2: np.ndarray, T: np.ndarray) -> np.ndarray:
       """
       :param im1: input image 1 in grayscale format.
       :param im2: input image 2 in grayscale format.
       :param T: is a 3x3 matrix such that each pixel in image 2
   ```

```
        is mapped under homogenous coordinates to image 1 (p2=Tp1).
        :return: warp image 2 according to T and display both image1
        and the wrapped version of the image2 in the same figure.
        """
```

6. **Testing:** In order to test your solution you should generate your own test images. You can create im2 from im1 by applying a known transformation and testing your functions.
   Submit your testing images under the following names:
   imTransA1.jpg, imTransA2.jpg, imTransB1.jpg, imTransB2.jpg
   and imRigidA1, imRigidA2, imRigidB1, imRigidB2
   (A and B represent the image pairs, and all images in jpg format).

4 Gaussian and Laplacian Pyramids

Remarks:

1. For all pyramids, use Gauusian kernel with a size of 5×5 and $\sigma = 0.3 * ((k_{size} - 1) * 0.5 - 1) + 0.8$ (you can use cv2.getGaussianKernel).

2. Remember, the sum of the kernel when down-sampling should be 1, and when up-sampling the sum should be 4.

3. Each level in the pyramids, the image shape is cut in half, so for x levels, crop the initial image to $2^x \cdot \lfloor img_{size}/2^x \rfloor$

4.1 Gaussian Pyramids

Write a function that returns a Gaussian pyramid for a given image.

```
def gaussianPyr(img: np.ndarray, levels: int = 4) -> List[np.ndarray]:
    """
    Creates a Gaussian Pyramid
    :param img: Original image
    :param levels: Pyramid depth
    :return: Gaussian pyramid (list of images)
    """
```

4.2 Laplacian Pyramids

Write a function that returns a Laplacian pyramid for a given image.

```
def laplaceianReduce(img: np.ndarray, levels: int = 4) -> List[np.ndarray]:
    """
    Creates a Laplacian pyramid
    :param img: Original image
    :param levels: Pyramid depth
    :return: Laplacian Pyramid (list of images)
    """


def laplaceianExpand(lap_pyr: List[np.ndarray]) -> np.ndarray:
    """
    Restores the original image from a laplacian pyramid
    :param lap_pyr: Laplacian Pyramid
    :return: Original image
    """
```

4.3 Pyramid Blending

```
def pyrBlend(img_1: np.ndarray, img_2: np.ndarray,
             mask: np.ndarray, levels: int) -> (np.ndarray, np.ndarray):
    """
    Blends two images using PyramidBlend method
    :param img_1: Image 1
    :param img_2: Image 2
    :param mask: Blend mask
    :param levels: Pyramid depth
    :return: (Naive blend, Blended Image)
    """
```

The Naive blend, is blending without using the pyramid.

5 Important Comments

- Add test to the following tasks:

  1. Hierarchical Lucas Kanade optical flow
  2. Compare the result of the functions *opticalFlow* (task 1) and *opticalFlowPyrLK* (task 2).
  3. Image Wrapping (task 3).

- The input of all the above functions can be either color or gray-scale images.
- Submit the two python files *ex3_main.py,ex3_utils.py* and any other python files you feel are necessary to use, along with all the images you use in your code.
- Your code should work 'out of the box', make sure your paths are correct and no other libraries are added.
- Do not change the file *ex3_main.py* except change the images path if you feel creative, and please do!!
- Only ZIP files!

6 Good Luck! :)