PR1 Part (a) *50 points*

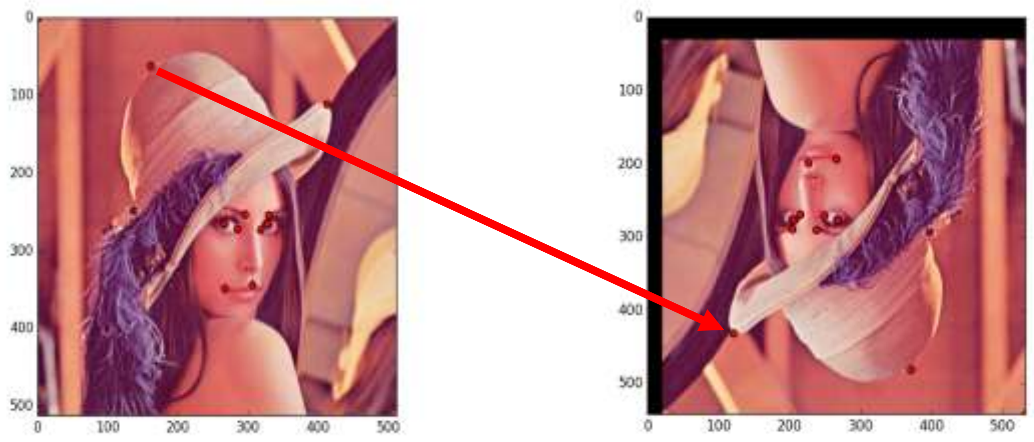
GIVEN:

We will provide you with five images: computer.png, lena.png, mario.jpg, mountain.jpg, water.jpg
These images will be referred to as *original* images in the remainder of the PR.

For an *original* image, we will also provide two additional images, and they will be referred to as *transformed* images in the remainder of the PR. *Transformed* images are obtained by separately applying ***affine*** and ***perspective*** transformations to the *original* images. Finally, you will be given ten corresponding points between the *original* and *transformed* images. Or in other words, the correspondences between the 2D coordinates of features for an *original* image and a *transformed* image are pre-established and constrained through either an ***affine*** transformation or a ***perspective*** transformation.

These *original* images are in the folder called "PartA/original." The *transformed* images are in folders called "PartA/affine" and "PartA/perspective." The correspondences between the *original* image and a *transformed* image are in a folder called "PartA/correspondences." Images are either in the .png or .jpg image format. Correspondences are in the .csv format.

Refer to the example given below to get a visual intuition. You are given the *original* image (shown on the left-hand side), the *transformed image (*shown on the right-hand side), and the corresponding points between them (overlayed on images as red points with a red arrow in between them).



original image *transformed image*

SOLVE:

Given a set of corresponding ***points*** between an *original* and a *transformed* image, estimate:

X1. *The **Affine** transformation matrix*
X2. *The **Perspective** transformation matrix*

More simply put, we are asking you:

Given a **few (or minimum)** corresponding points between the *original* and *transformed* image, solve for the transformation matrix that summarizes the transformations (scale, rotate, shear, etc.) applied to the **entire** *original* image, by way of estimating the affine/perspective transformation matrix.

This problem can be expressed as a linear system of the form A***X*** = B, where ***X*** is the transformation matrix you are trying to estimate.
The built-in functions in OpenCV (or MATLAB) can estimate the unknown transformation matrix. The OpenCV function *getAffineTransform()* can estimate matrix ***X1***. The OpenCV function *getPerspectiveTransform()* can estimate matrix ***X2***. Official documentation of these functions can be found here. These functions are a great way to validate your answer. **However, we want you to code these functions by yourself!**

Thus, you will use the corresponding points to solve matrices ***X1*** and ***X2*** with the Least Squares method **OR** the SVD method. You can code these methods by either referring to the slides or/and the lectures for this. Your approach to solving matrices ***X1*** and ***X2*** must follow the requirements given below. (**Make sure to remove the mean from each set of points**):

A. Solve for the matrices ***X1*** and ***X2*** with minimal number of correspondences (3 for ***affine***, 4 for ***perspective***)
B. More than minimal required correspondences (over-constrained set of equations).

   For both matrices, provide with unit-vectors.

Compute the error between **your** estimate of matrices ***X1*** and ***X2*** *against* the **built-in** functions. Also, display the transformed images using **your** estimate of matrices ***X1*** and ***X2***. You can use the OpenCV function *warpAffine()* and *warpPerspective()* for this. These matrices and the transformed image can be considered as your initial guess.

Ideally, adding more correspondences should give a better estimation of ***X1*** and ***X2***. **Find and add** ten corresponding extra points to your existing list of corresponding points by manually clicking corresponding points between the *original* and the *transformed* images (*transformed* images provided by us and not your initial guess!). Does the error reduce further compared to your first guess?

Deliverables:

X1:	the ***affine*** transformation matrix
X2:	the ***perspective*** transformation matrix
Method A:	Solve matrices using minimal number of correspondences
Method B:	Solve matrices using more than minimum correspondences

For each of the 5 *original* images, you need to report:
- **your** estimated ***X1*** using method A, and also **your** *transformed* image
- **your** estimated ***X1*** using method B, and also **your** *transformed* image
- **your** estimated ***X2*** using method A, and also **your** *transformed* image
- **your** estimated ***X2*** using method B, and also **your** *transformed* image
- the error between **your** ***X1*** using method A against the ***X1*** generated using built-in function
- the error between **your** ***X1*** using method B against the ***X1*** generated using built-in function
- the error between **your** ***X2*** using method A against the ***X2*** generated using built-in function
- the error between **your** ***X2*** using method B against the ***X2*** generated using built-in function

For adding the 10 extra correspondences (only for the computer.png), you need to
- manually pick 10 correspondences for [*original*– ***affine***] and [*original* - ***perspective***]
- save the correspondences in a new .csv file
- report the estimated **X1** and **X2** along with the errors.
- turn in **your** *transformed* images

README file: tells how to run your codes (i.e. python versions, library version, etc.)
A text file that stores all your estimated matrices, which should be well named and organized

Note: **your** *transformed* images should also be well named and organized

PR1 Part (b) *50 points*

This part of the assignment involves implementing various functionalities discussed in class regarding the Gaussian and Laplacian pyramids. The images required for this assignment are stored in the folder called PartB. For questions 1-6, use the image called lena.png stored in PartB/lena.png to show your results. For questions 4 to 6, you can refer to the OpenCV documentation here. However, you **CANNOT** use the built-in OpenCV functions provided in this documentation (except display functionalities such as *imread(), imshow(), imwrite(), waitKey()*, etc.)

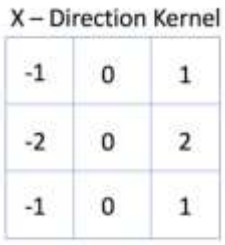
1. Write a function to convolve an input image ($\mathbf{I}_{inp}$) with a kernel (**K**) to produce a convolved image ($\mathbf{I}_{out}$), i.e,

$$\mathbf{I}_{out} = \text{Convolve}\ (\mathbf{I}_{inp}, \mathbf{K})$$

K can be of varying sizes. $\mathbf{I}_{inp}$ is in *RGB* format. Display $\mathbf{I}_{out}$. Note that you are expected to write your function to convolve the image and **NOT** use OpenCV *filter2D()* function. However, you can use *filter2D()* to verify if your code works as intended.

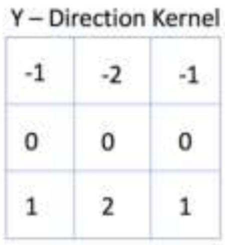
Example:

$\mathbf{I}_{inp}$



X – Direction Kernel

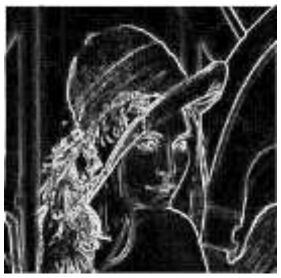
-1	0	1
-2	0	2
-1	0	1



Y – Direction Kernel

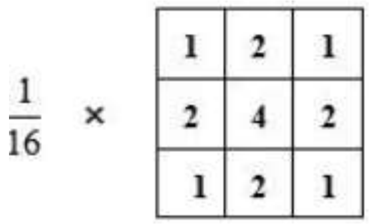
-1	-2	-1
0	0	0
1	2	1

K



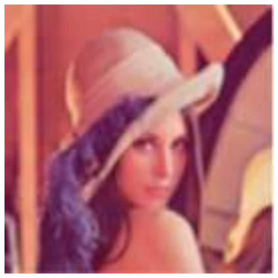
$\mathbf{I}_{out}$

$\mathbf{I}_{inp}$



$$\frac{1}{16} \times \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

K



$\mathbf{I}_{out}$

2. Write a function to reduce an input image ($\mathbf{I}_{inp}$) by half its width and height ($\mathbf{I}_{out}$), i.e,

$$\mathbf{I}_{out} = \text{Reduce}\ (\mathbf{I}_{inp}),$$

such that,

$$\text{height}(\mathbf{I}_{out}) = \text{height}(\mathbf{I}_{inp})/2$$
$$\text{width}(\mathbf{I}_{out}) = \text{width}\ (\mathbf{I}_{inp})/2$$

Remember to Gaussian filter the image before reducing it; use separable 1D Gaussian kernels. $\mathbf{I}_{inp}$ is in *RGB* format. Display $\mathbf{I}_{out}$. Note: An ***interpolation algorithm*** will be required to assign pixel values in $\mathbf{I}_{inp}$ to $\mathbf{I}_{out}$. You are expected to write your function to reduce the image and **NOT** use OpenCV *resize()* function.

3. Write a function to expand an input image ($\mathbf{I}_{inp}$) by twice its width and height ($\mathbf{I}_{out}$), i.e,

$$\mathbf{I}_{out} = \text{Expand}(\mathbf{I}_{inp}),$$

such that,

$$\text{height}(\mathbf{I}_{out}) = \text{height}(\mathbf{I}_{inp}) * 2$$
$$\text{width}(\mathbf{I}_{out}) = \text{width}(\mathbf{I}_{inp}) * 2$$

$\mathbf{I}_{inp}$ is in *RGB* format. Display $\mathbf{I}_{out}$. Note: An ***interpolation algorithm*** will be required to assign pixel values in $\mathbf{I}_{inp}$ to $\mathbf{I}_{out}$. You are expected to write your function to expand the image and **NOT** use OpenCV *resize()* function.

4. Use the "Reduce" function to write the GaussianPyramid($\mathbf{I}_{inp}$, **n**) function, where **n** is the no. of levels. Note that a higher level (lower resolution) in a Gaussian Pyramid is formed by removing consecutive rows and columns in a lower level (higher resolution) image.

5. Use the above functions to write LaplacianPyramids($\mathbf{I}_{inp}$, **n**) that produces **n** level Laplacian pyramid of I. Note that a level in Laplacian Pyramid is formed by the difference between that level in the Gaussian Pyramid and expanded version of its upper level in the Gaussian Pyramid.

6. Write the Reconstruct(**LI**, **n**) function, which collapses the Laplacian pyramid **LI** of **n** levels to generate the original image. Report the error in reconstruction using image difference.

7. Finally, you will be mosaicking images using Laplacian plane based reconstruction (Note that your program should handle color images). Let the user pick the blend boundaries of all images by mouse. Blend the *left* image with the *right* image using**: i)** affine unwarping **OR** perspective unwarping, **ii)** no unwarping. Note the *left* and *right* images share a joint region. Find points for unwarping manually. Submit four results of mosaicking. Images can be from the web. Show results on **i)** and **ii)** for each of the four mosaics. Each mosaic can be comprised of 2-4 individual images from different cameras/viewpoints. So, in total we are expecting 8 combinations to mosaic the images.

Extra Credit for Part(b): *25 points*

The extra credit analyses the use of corner detectors to compute correspondences between images and the possibility of the automatic creation of image mosaics. There will be 25 points for a completely automated solution and 15 points for a semi-automatic solution. Apply the Harris corner detector (MATLAB/OpenCV library) to compute point correspondences between two image feature (corner) points. Image matching can be done using normalized cross-correlation (or any other method such as SSD). Submit the results on the same images as above.

Deliverables:

For questions 1~6:

- show output images for each of the functions
- the output images should be well named and organized

For question 7:

- provide 8 mosaicked images using all combinations of **i)** affine unwarping **OR** perspective unwarping, **ii)** no unwarping for all 4 of your *left* and *right* images.
- at least 1 affine unwarping and 1 perspective unwarping

For Extra Credit:

- provide 4 mosaicked images with perspective unwarping

Programming Instructions

1. File locations/paths:

Hardcoding paths to file locations should be avoided in python. Relative paths should be used instead wherever possible. Hardcoded paths should be declared in a separate configuration file (usually .yaml) and imported as string variables in the programming logic using. This separates programming logic from trivial details of finding where the paths are. Using argparser is also acceptable but it becomes cumbersome since arguments are parsed from the command line. Finally declared path should be independent of the operating system that is used.

2. Modularity and testing

Python allows developers to create and import their own packages. Instead of creating a single file with all functions, all required functions should be declared in a separate file and then imported into the main file. Not only this allows for better code readability but also allows the developer to unit test these codes separately from the combined logic. This makes debugging code easier as well.

3. Function declarations

Each function should have a description of what a function does, what is the input to the function, and what is the expected output. The description should be short. Default arguments should be provided to the keyword arguments of the functions.

4. Checking for value errors:

Developers should explicitly tell the functions about what input types it should expect (int, float, arrays, etc..). They should write if statements to check for NoneType errors instead of python throwing this error at run time. Assert statements should be used to test the validity of output types (E.g: Check if output shape out convolution has three channels as well)

5. Edge cases

A developer should write programs knowing different logical edge cases to their own functions.

6. Reproducibility:

Before submitting on Canvas students should run their code and check if their program does not give any errors. A ReadMe file should always be provided. ReadMe is a simple text file that contains instructions about what third party libraries are required to run their program, the commands to run their program, etc.