

Practical session : Geometric Transformations on Digital Images

In this session, we will program in **Python** using the **OpenCV** library¹(Open Source Computer Vision Library) for the practice of rigid motions on digital images that has been seen in class.

The OpenCV is a graphical and open-source library. It has many features for creating graphical interfaces as well as tools in the field of image and video processing. You will find complete documentation of the library at <https://docs.opencv.org/4.x>.

For programming environment, we will use the online **JupyterLab** : <https://jupyter.org/try-jupyter/lab> that does not require any installation or sign up, and it allows us to realize directly the exercises in this session.

Exercise 1 _____ A first program with OpenCV

We start with a simple OpenCV program that loads an image and displays it in a window. For this, we will use with the following functions :

```
#Create a window
cv.namedWindow( winname [, flags=WINDOW_AUTOSIZE] ) -> None
#Load an image from a file
cv.imread( filename[, flags=IMREAD_COLOR ] ) -> retval
#Display an image in the specified window.
cv.imshow( winname, mat ) -> None
#Save an image to a specified file.
cv.imwrite( filename, img[, params] ) -> retval
#Wait for a pressed key.
cv.waitKey( [delay] ) -> retval
#Destroy all of the opened windows
cv.destroyAllWindows( ) -> None
```

You can find a full description of the functions and their parameters on the documentation page of OpenCV library at <https://docs.opencv.org/4.x/>.

Now using these functions, complete the file `exo1.py` and run to test it with different input images such as : **FIXME**.

Exercise 2 _____ Image rotation with interpolation

Let do some rotations on digital images using an available function in `imutils` package with OpenCV. More precisely, we consider the following function :

1. <https://opencv.org>

```
#Rotate an image with angle of rotation (in degree)
cv.rotate( src,  rotation_angle ) -> dst
```

To make the program more interactive, we will add an `trackbar` to the window that allows to control the rotation angle and show the rotated image in the opened window. For this we need the following functions :

```
#Create a trackbar and attaches it to the specified window.
cv.createTrackbar( trackbarname, winname, slider_currnet_value,
                  max_slider_value, slider_callback [, params] )
#Define a callback function for the trackbar
def trackbarCallback( angle : int ) -> None :
    #Function to be executed every time the slider changes position
#Call the defined callback function
trackbarCallback( 0 )
```

Now using these functions, complete the file `exo2.py` and run to test it with different input images such as : `FIXME`.

We can easily observe the interpolation effects that appear on the images such as : `circles_500.png`, `strike_500.png` and `Retina_1.png`.

Exercise 3 _____ Digitized rotation in \mathbb{Z}^2

In this exercise, we will do simply rotations of point in the continuous space of \mathbb{R}^2 and the discrete space of \mathbb{Z}^2 that have been presented in the course. For this, different classes of object have been prepared in the file `exo3.py`, we have :

- Class `ImagePoint` : for points in \mathbb{Z}^2
- Class `RealPoint` : for points in \mathbb{R}^2
- Class `rotation` : for a rotation with a rotation angle `theta` and a rotation center, in this class there are the following methods :
 - `getRotationMatrix` : get the rotation matrix associated to the rotation angle `theta`
 - `getInverseRotationMatrix` : get the inverse rotation matrix which correspond to the rotation angle `-theta`
 - `forwardRotation` : realize a forward rotation on a real point
 - `digitizedForwardRotation` : realize a backward rotation on a real point then digitized the result to obtain a image point
 - `backwardRotation` : realize a forward rotation on a real point
 - `digitizedBackwardRotation` : realize a backward rotation on a real point then digitized the result to obtain a image point

Complete the file `exo3.py` and run to test the forward and backward rotations on points in \mathbb{R}^2 and \mathbb{Z}^2 . You obtain the following result.

```
RealPoint:( 5.2 , 3.5 )
ImagePoint:( 4 , 5 )
ImagePoint:( 4 , 5 )
RealPoint:( 1.2020815280171315 , 6.151828996322964 )
ImagePoint:( 6 , 1 )
ImagePoint:( -1 , 6 )
```

Exercise 4 — Digitized rotation of digital images

We will now realize the digitized rotation on digital image using the forward and backward models from the previous exercise for points in a digital image. To do this, different functions have prepared in the file `exo4.py`, we have :

- `getPixelValue` : get the value of a given pixel in image
- `setPixelValue` : set a value to a pixel in image
- `forwardRotation` : realize a forward rotation on a real point
- `createBlackImage` : create a black image of size width \times height
- `forwardRotationImage` : realize a digitized forward rotation on an input image for a rotation angle `thetha`
- `backwardRotationImage` : realize a digitized backward rotation on an input image for a rotation angle `thetha`

Complete the file `exo4.py` and run to test the forward and backward rotations on different input images such as : `FIXME`.

Hereafter, an example of result obtained on `church.png` for a rotation angle of 45° .



Forward rotation (Lagrangian model)




Backward rotation (Eulerian model)

Exercise 5 Well-composed image

We can easily observe the geometrical and topological issues appeared on the transformed images such as : `circles_500.png`, `strike_500.png` and `Retina_2.png`.

Hereafter, we apply the notion of *regularity* presented in the course to perform a topology-preserving rotation on digital images. To this end, we consider 4-connectivity for both object and background of binary images, namely *well-composed images*.

Let start with the verification of such images. We recall that, a binary image is *well-composed* if it does not contain the *forbidden configuration* (up to symmetries) : 

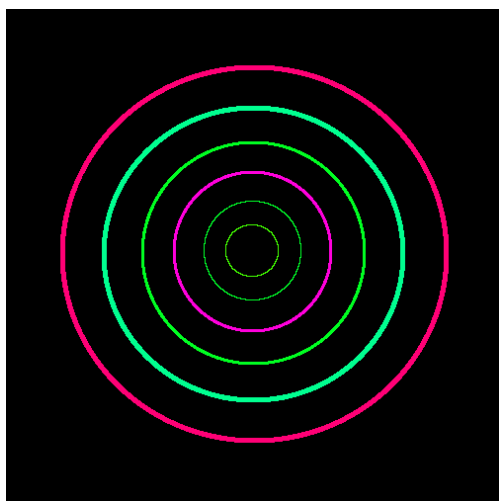
Complete the file `exo5.py` and run to test it. Here are the results you must obtain when running the program for the well-composed verification of the following images.

Image's filename	well-composed
<code>Retina_1.png</code>	✗
<code>Retina_2.png</code>	✓
<code>circles_500.png</code>	✗
<code>circles_500_2.png</code>	✓

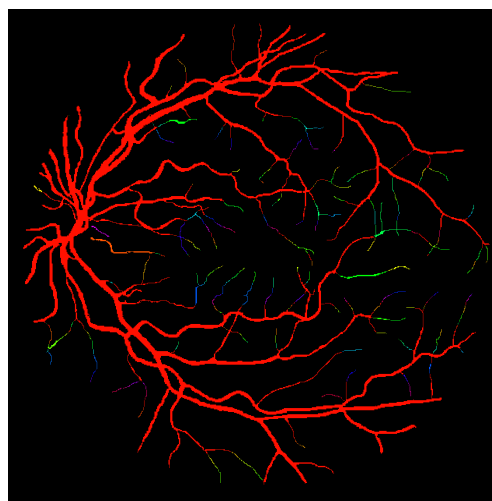
Optional : If you want to display the connected components in a given binary image, you can use the following function in OpenCV.

```
#computes the connected components labeled image of binary image.
cv.connectedComponents( image[, labels[, connectivity[, ltype]]])
                        -> retval, labels
```

Hereafter, examples of result obtained on running the connected components on two different images.



`circles_500_2.png`



`retina_1.png`

You can run the code in `exo4.py` to rotate the well-composed images and observe the number of connected components before and after the rotation. Normally, it is not preserved. To perform a topology-preserving rotation of digital images, we will need the notion of **regularity**.

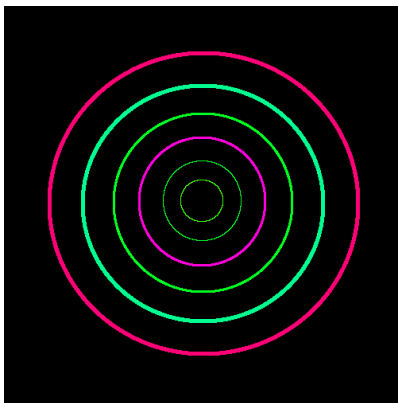
Exercise 6 — Regularization of well-composed image

As presented in the course, regular images allow to preserve their topology under any rigid motion and thus rotation. One solution to make a well-composed image I become a regular image is to upsample I by twice. More precisely, a pixel $p \in I$ turns to 2×2 pixel in the generated regular image. Note that the method works if and only if I is well-composed.

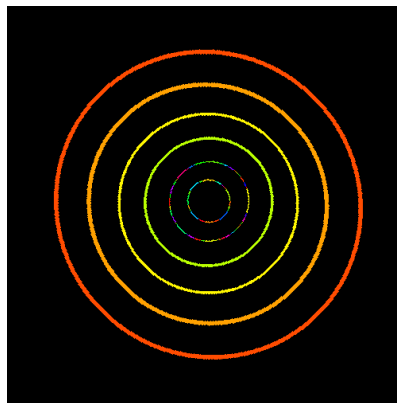
A function called `createRegularImage` is defined in `exo6.py` to upsample a well-composed image. Complete this file `exo6.py` and run to test it.

Optional : If you want to check the topology-preserving rotation, we can use the code in `exo4.py` to rotate an image and the `cv.connectedComponents` of OpenCV to show the connected component before and after the rotation. Note that, as mentioned in the course, to avoid the **incomplete and ambiguous issues of color** in the transformed image, we consider hereafter the backward rotation to generate the transformed results. The example code is given in `exo7.py`.

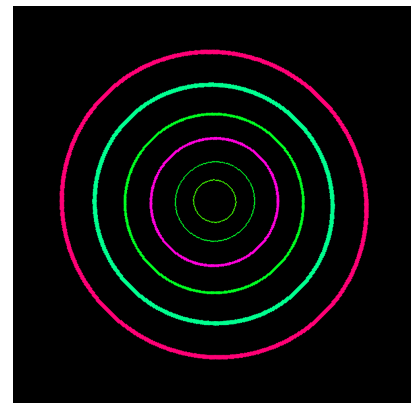
Here are the results you must obtain when testing the topology-preserving rotations in `exo7.py` on different images.



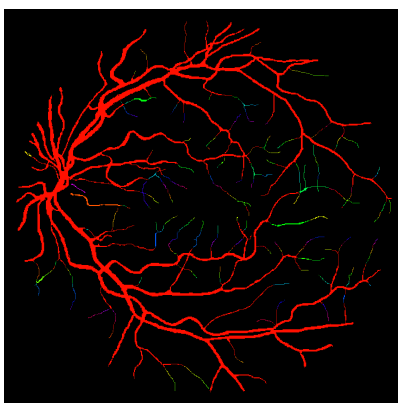
`circles_500_2.png`



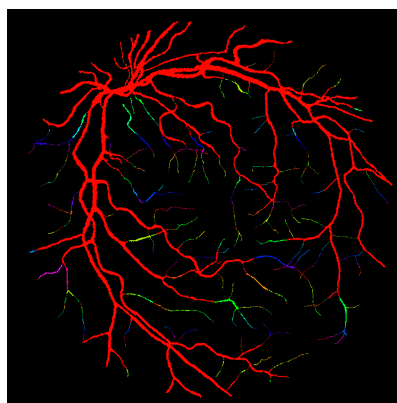
Backward transformation



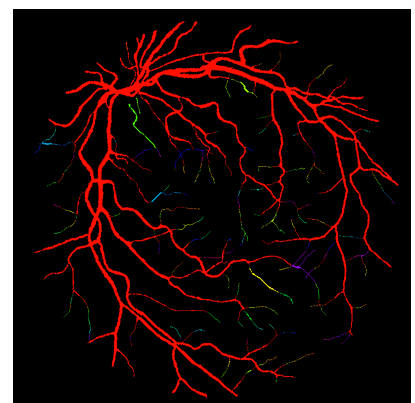
Transf. on regular image



`retina_2.png`



Backward transformation



Transf. on regular image