Xin Zhong

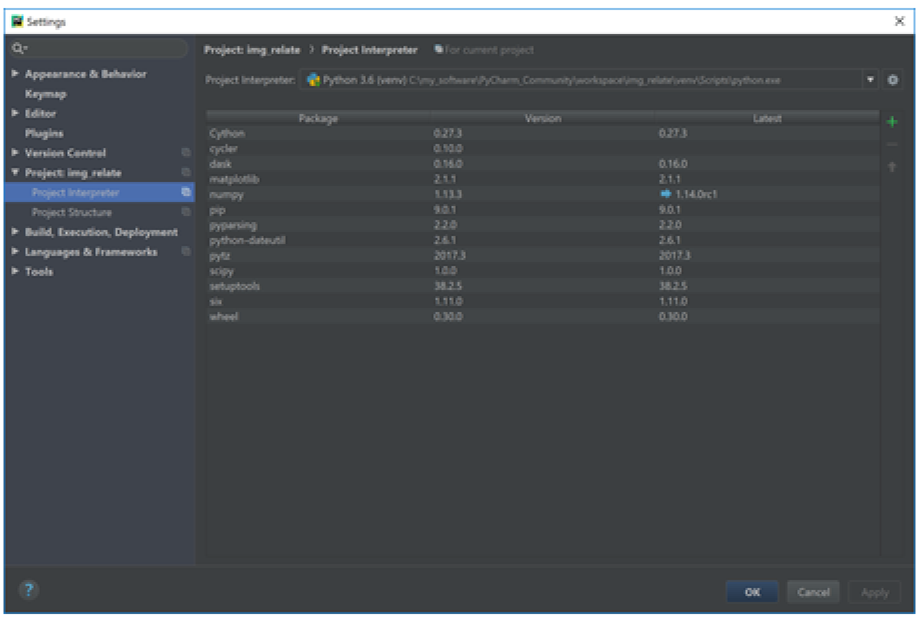
Log of Python for Image Processing

Matlab is convenient for image processing. However, Matlab is not open source, and mainly focus on research. Codes on Matlab can be hardly turned into software. So here we use Python scripts for image processing tasks.

Installation

1. Install python3.
2. Similar to Matlab toolbox, we want python library and package for image processing purpose, libraries examples

   - openCV: mainly on C++, python API updating too slow.
   - scikit-image: based on scipy, an image is a numpy array, this looks good, let's choose scikit-image for image work. So we need scipy, numpy, matplotlib(for image display), basic packages:



Install Anaconda (https://www.anaconda.com/download/) to easily get more packages.

Test:

Show Lena and its red channel as gray map:

```
from skimage import io
img = io.imread('lena.png')
io.imshow(img)
io.show()
```

```
img_c = img[:,:,0]
io.imshow(img_c)
io.show()
```

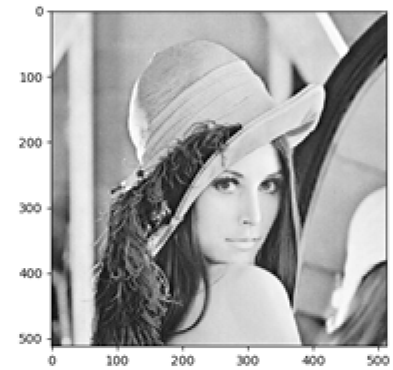


Image Basic

1. read in

An image returned by imread() is a [column, row, 3] numpy array in RGB.

```
img = io.imread('image_name.extension')
```

2. display

```
io.imshow(image_variable)
io.show()
```

3. write

```
io.imsave('image_name.extension',image_variable)
```

4. image information

```
print(type(img))
print(img.shape)  # as matlab size()    r,c,dim = img.shape
print(img.shape[0])  # rows
print(img.size)   # total pixel number, as matlab numel()
print(img.max())  # maximum value
print(img.mean())
```

5. pixel retrieval

As an numpy array, the retrieval is as img[row, column, channel]. e.g. get the value of row 20, col 30, green

```
print(img[20,30,1])
```

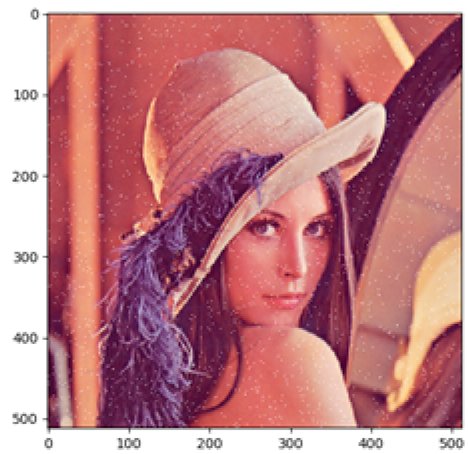
like Matlab, “:” means all, RGB channels are

```
img[:,:,0] # red        img[:,:,1] # green      img[:,:,2] # blue
```

6. pixel modification

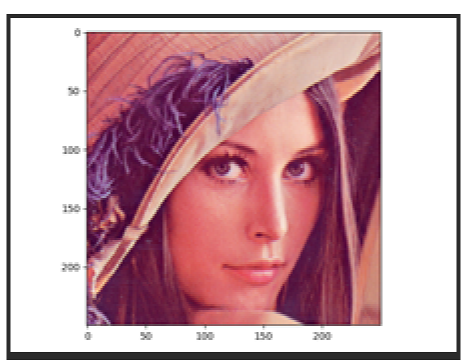
e.g. 1: salt and pepper noise

```
from skimage import io
import numpy as np
img = io.imread('lena.png')
r,c,d = img.shape
for i in range(3000):
        x,y = np.random.randint(0,r), np.random.randint(0,c)
        img[x,y,:] = 255
io.imshow(img)
io.show()
```



e.g. 2 cropping

```
io.imshow(img[150:400,150:400,:])
io.show()
```

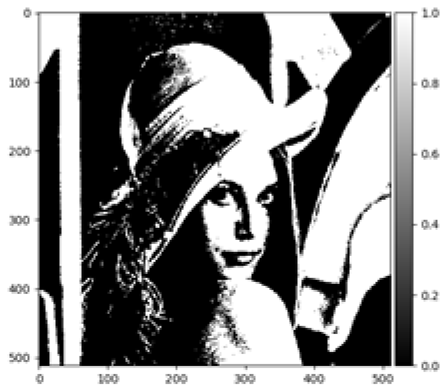


e.g. 3 local computation

```python
print(img[1:3,1:3,:].sum())
print(img[1:3,1:3,:].mean())
```

7. Binarization, 128 as the threshold (first bit plane), vectorized

```python
from skimage import io,color
img = io.imread('lena.png')
img_g = color.rgb2gray(img)    # rgb2gray normalize the image to [0,1]
img_b = np.zeros((img_g.shape),dtype=np.int8)
img_b[img_g[:,:] > (128/255)] = 1
io.imshow(img_b, cmap='gray')
io.show()
```



Type and color space

1. Memory space

an image, the numpy array has type of
uint8 [0, 255], uint16 [0, 65535], float [0,1], int8 [-128 127], etc.
typical conversion in skimage:

img_as_float	64 bit [0 1]
img_as_ubyte	8 bit [0 255]
img_as_uint	16 bit [0, 65535]

2. Color space

Note: any conversion in color space change the type to float [0, 1]

Frequent use in package skimage.color (more on http://scikit-image.org/docs/dev/api/skimage.color.html)

rgb2gray	gray scale
gray2rgb	amazingly change back…
rgb2lab	Lab space
rgb2xyz	XYZ spcae
xyz2lab	CieLab space

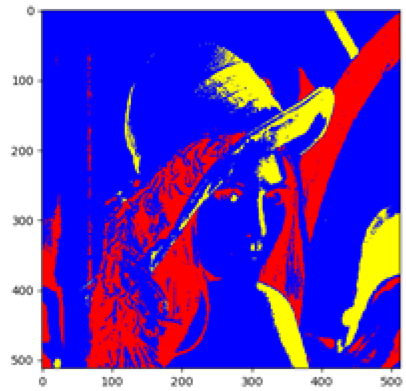
“skimage.color.convert_colorspace” function could call some conversion functions conveniently:

```
from skimage import io, color
img = io.imread('lena.png')
cie = color.convert_colorspace(img,'RGB','RGB CIE')
io.imshow(cie)
io.show()
```

conversion color space can be:
['RGB', 'HSV', 'RGB CIE', 'XYZ', 'YUV', 'YIQ', 'YPbPr', 'YCbCr']

Labels to color

```
from skimage import io, color
import numpy as np
img = io.imread('lena.png')
lg = color.rgb2gray(img)
r, c = lg.shape
labels = np.zeros([r,c])
labels[lg[:,:] > 0.75] = 2
labels[np.logical_and(lg[:,:] < 0.75,lg[:,:] > 0.25)] = 1
lc = color.label2rgb(labels)
io.imshow(lc)
io.show()
```



Matplotlib

Skimage.io.imshow(), ~.show() and ~.imread() essentially use matplotlib library as the plugin for image display, like Matlab plot series, matplotlib is good for matrices display.
so

```
from skimage import io, color
img = io.imread('lena.png')
io.imshow(img)
io.show()
```

is essentially

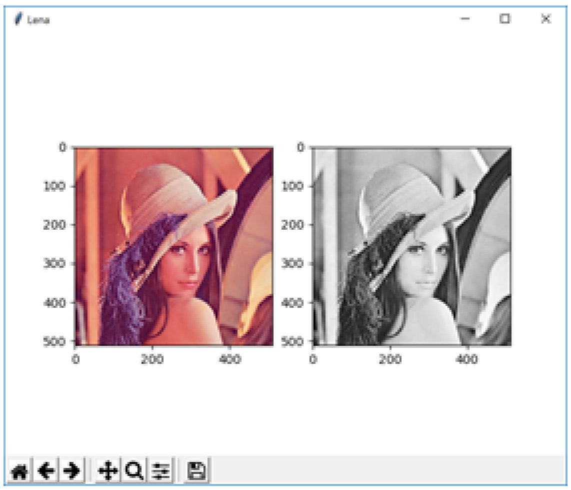
```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
img = mpimg.imread('lena.png')
plt.imshow(img)
plt.show()
```

1. Figure window and subplot
plt.figure(num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None)
plt.subplot(row, col, position), position row wisely

plt.show() displays all pending figures, so use it at the end to pop out all the figures one time.

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
img = mpimg.imread('lena.png')
plt.figure(num='Lena')
plt.subplot(1,2,1)
```

```
plt.imshow(img)
plt.subplot(1,2,2)
plt.imshow(img[:,:,0],cmap='gray')
plt.figure(num='Lena2')
plt.subplot(1,2,1)
plt.imshow(img[:,:,1],cmap='gray')
plt.subplot(1,2,2)
plt.imshow(img[:,:,2],cmap='gray')
plt.show()
```



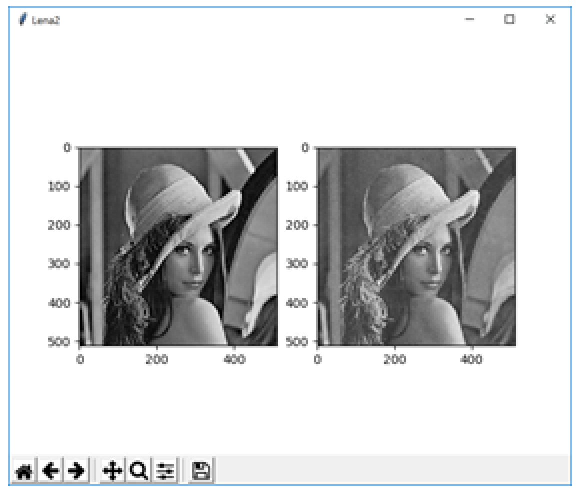


Image Distortion

Skimage.transform http://scikit-image.org/docs/dev/api/skimage.transform.html

1. resize, change to arbitrary size

```
ts = transform.resize(img,(64,64))
```

skimage.transform.resize(image, output_shape)

2. rescale, change to a factor

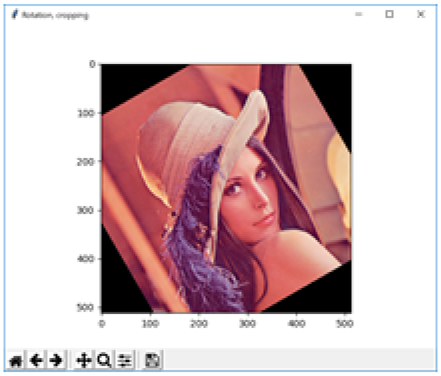
```
tsc = transform.rescale(img,[0.1,0.1])
```

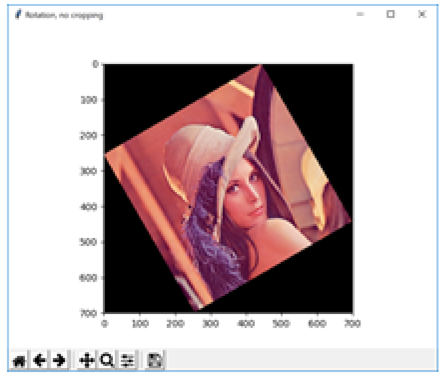
skimage.transform.rescale(image, scale[, …])

3. rotation

```
tr = transform.rotate(img,30)  # with cropping
trf = transform.rotate(img,30,resize=True)  # without cropping
```

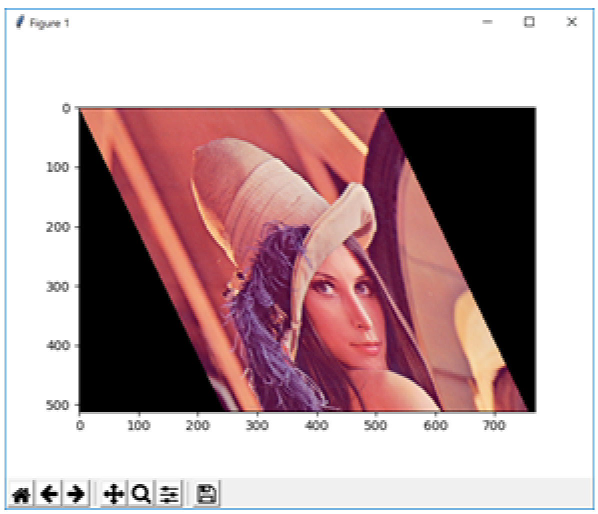
skimage.transform.rotate(image, angle[, …])





4. shearing

```
r,c,d = img.shape
sh_f = 0.5
tsh_m = transform.AffineTransform(shear=sh_f)
tsh = transform.warp(img, inverse_map=tsh_m, output_shape=(r,c*(1+sh_f)))
```



5. Affine Distortion

First we try

skimage.transform.AffineTransform(matrix=None, scale=None, rotation=None, shear=None, translation=None), However this class **only supports shearing along x direction**.

the transformation matrix is

    [sx * math.cos(rotation), -sy * math.sin(rotation + shear), 0]
    [sx * math.sin(rotation), sy * math.cos(rotation + shear), 0]
    [0,   0, 1]

where sx, sy are scale factors in the x and y direction.

To program general affine distortions (at least supporting y shearing), we first formulate it as: for each coordinate [x, y, 1]', its target is multiplying with the given matrix H:

$$\begin{bmatrix} A & B & C \\ D & E & F \\ 0 & 0 & 1 \end{bmatrix}$$

For translate, scale, rotation, and shearing the A – F respectively are:

$$\begin{bmatrix} 1 & 0 & Tx \\ 0 & 1 & Ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos & -sin & 0 \\ sin & cos & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & SHx & 0 \\ SHy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

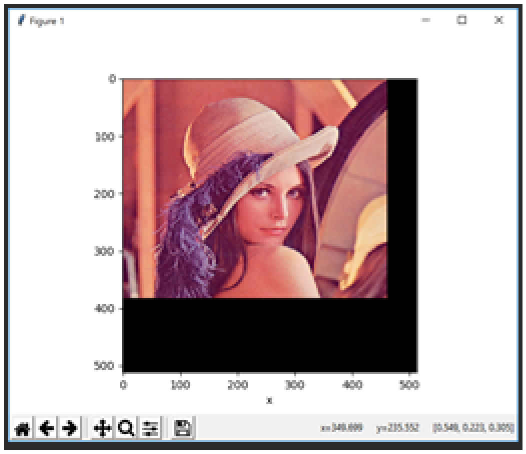
Having these, we apply skimage.transform.ProjectiveTransform (matrix=None) to seal the matrix

(1) translation, x shift 20 and y shift 30

```
matrix = np.array([[1,0,20],[0,1,30],[0,0,1]])
tform = transform.ProjectiveTransform(matrix)
img_trans = transform.warp(img, inverse_map=tform)
```

(2) scale, x to 0.9 and y to 0.75

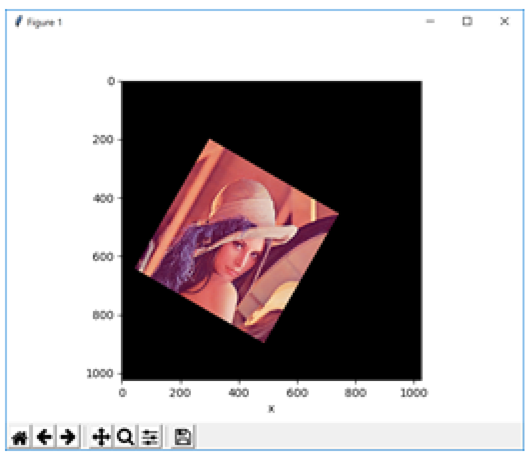
```
matrix = np.array([[0.9,0,0],[0,0.75,0],[0,0,1]])
tform = transform.ProjectiveTransform(matrix=matrix)
img_trans = transform.warp(img, tform.inverse)
plt.imshow(img_trans)
plt.xlabel('x')
```



Note: function warp accepts the inverse transformation matrix.

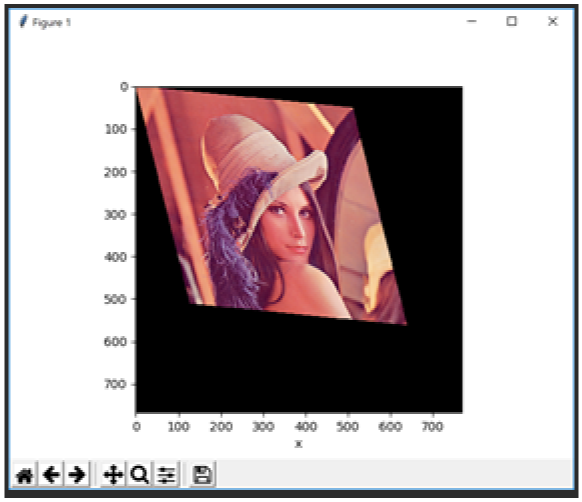
(3) rotation, 30 degree clock-wisely

```
matrix =
np.array([[math.cos(math.radians(30)),-1*math.sin(math.radians(30)),300],[math.sin(math.radian
s(30)),math.cos(math.radians(30)),200],[0,0,1]])
tform = transform.ProjectiveTransform(matrix=matrix)
img_trans = transform.warp(img, tform.inverse,output_shape=(1024,1024))
```



(4) shearing, x 0.25 y 0.1

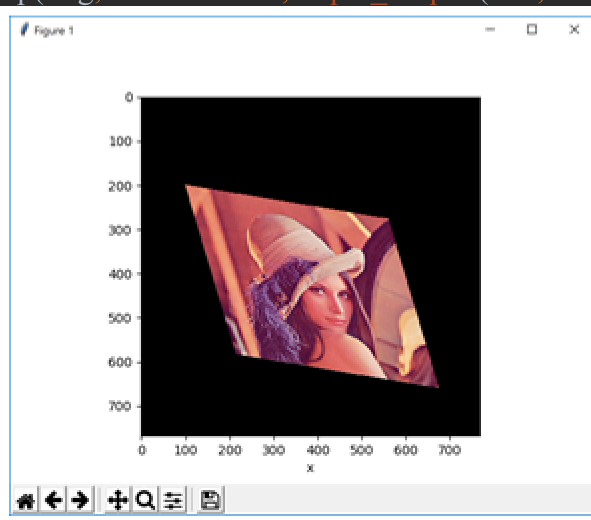
```
matrix = np.array([[1,0.25,0],[0.1,1,0],[0,0,1]])
tform = transform.ProjectiveTransform(matrix=matrix)
img_trans = transform.warp(img, tform.inverse,output_shape=(768,768))
```



With ProjectiveTransform and warp that doing the multiplication between H and each coordinate, we achieved the shearing along y axis 😊, actually this implementation can use arbitrary transform matrices.

(5) Put some transformations together, e.g. translation + scale + shearing

```
matrix1 = np.array([[1,0,100],[0,1,200],[0,0,1]])  # trans
matrix2 = np.array([[0.9,0,0],[0,0.75,0],[0,0,1]]) # scale
matrix3 = np.array([[1,0.25,0],[0.2,1,0],[0,0,1]]) # shear
matrix = matrix1 @ matrix2 @ matrix3        # matrix = matrix1.dot(matrix2).dot(matrix3) for python 3.5 lower
tform = transform.ProjectiveTransform(matrix=matrix)
img_trans = transform.warp(img, tform.inverse,output_shape=(768,768))
```



Thresholding and Filtering

1. Ostu

Core function: skimage.filters.threshold_otsu(image, nbins=256)

```
from skimage import io,filters,color,transform
import matplotlib.pyplot as plt
img = transform.rescale(color.rgb2gray(io.imread('lena.png')),[0.5,0.5],mode='constant')
thresh = filters.threshold_otsu(img)
img_bin =(img > thresh)*1
plt.figure('thresh')
plt.subplot(121)
plt.title('original image')
plt.imshow(img,plt.cm.gray)
plt.subplot(122)
plt.title('binary image')
plt.imshow(img_bin,plt.cm.gray)
plt.show()
```

Also try
skimage.filters.threshold_adaptive(image, block_size, method='gaussian') that returns a binary image with local thresholding.

2. Global thresh

Algorithm:

(1) Compute the global mean -> M

(2) Threshold the image with M -> Iu, Id; Compute the mean of Iu, and Id -> Mu, Md

(3) Repeat until (Mu + Md) / 2 == M

- Update M as (Mu + Md) / 2
- Threshold the image with M -> Iu, Id, compute the mean of Iu, and Id -> Mu, Md

(4) Return M

```
from skimage import io,filters,color,transform
import numpy as np
def global_thresh(img):
    mean_pre, end = img.mean(), 0
    mean_cur = (np.mean(img[img<mean_pre]) + np.mean(img[img>mean_pre])) / 2
    while abs(mean_pre - mean_cur) > end:
    mean_pre = mean_cur
    mean_cur = (np.mean(img[img < mean_pre]) + np.mean(img[img > mean_pre])) / 2
    return mean_cur
img = transform.rescale(color.rgb2gray(io.imread('lena.png')),[0.5,0.5],mode='constant')
thresh = global_thresh(img)
img_bin =(img > thresh)*1
```

skimage.filters.rank (http://scikit-image.org/docs/dev/api/skimage.filters.rank.html)
Some important filters examples

```
from skimage import io, color
from skimage.morphology import disk
import skimage.filters.rank as sfr
img = color.rgb2gray(io.imread('lena.png'))
localmaxima = sfr.maximum(img, disk(3))  # localmaxima
localminima = sfr.minimum(img, disk(3))  # local minima
meanf = sfr.mean(img, disk(3))  # mean filter
medianf = sfr.median(img, disk(3))  # median filter
ch = sfr.enhance_contrast(img, disk(3))
# contract enhancement, algo: replace local pixel with nearest local maxima/ minima
le = sfr.entropy(img, disk(5))  # local entropy
```

histogram

skimage.exposure.histogram(image, nbins=256)

```
from skimage import exposure
import numpy as np
hist1=np.histogram(img, bins=256)  # upper/lower bound for each bin
hist2=exposure.histogram(img, nbins=256)  # mid value for each bin
```

img1=exposure.equalize_hist(img)   -- histogram equalization

n, bins, patches = plt.hist(arr, bins=10, normed=0, facecolor='black', edgecolor='black',alpha=1, histtype='bar')

```
import matplotlib.pyplot as plt
plt.figure("hist")
arr=img.flatten()  # 2D to 1D
n, bins, patches = plt.hist(arr, bins=256, normed=1, edgecolor='None', facecolor='blue')
plt.show()
```

Image Morphology

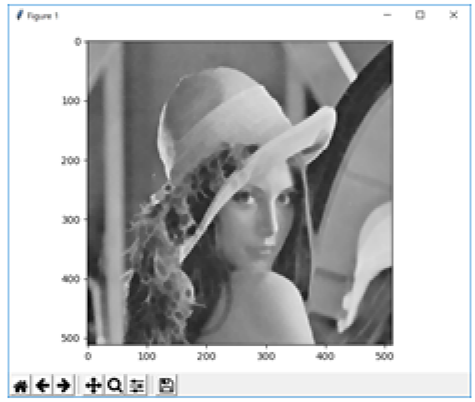
1. Dilation and Erosion, there are various structural element we can use

```
from skimage import io, color
img = color.rgb2gray(io.imread('lena.png'))
import skimage.morphology as morph
img_d = morph.dilation(img, morph.square(5))
img_e = morph.erosion(img, morph.disk(3))
```


2. Opening and Closing

Opening is an erosion followed by a dilation
Closing is a dilation followed by an erosion
There are predefined functions for them.

```
img_o = morph.opening(img,morph.diamond(3))
img_c = morph.closing(img,morph.star(3))
```

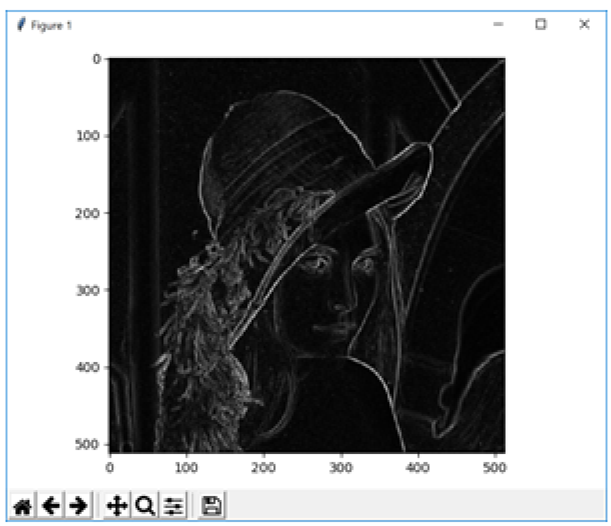


3. Morphological gradient

Dilation and erosion are often used in combination to produce a desired image processing effect. One simple combination is the morphological gradient.

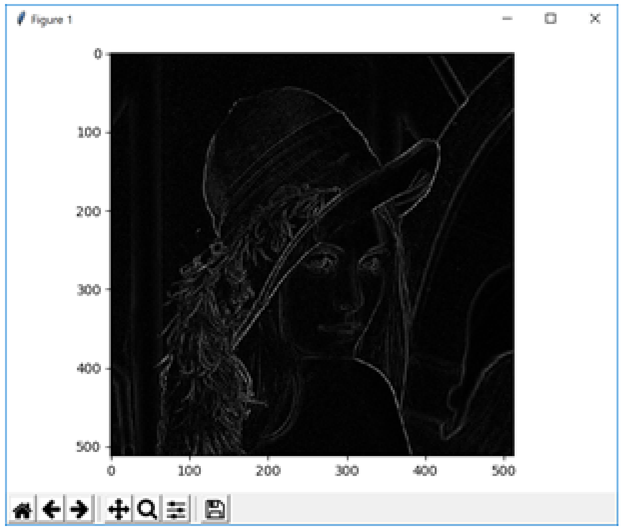
(1) Dilation - Erosion (Basic gradient)

```
img_gb = morph.dilation(img, morph.square(3)) - morph.erosion(img, morph.square(3))  # basic gradient
```



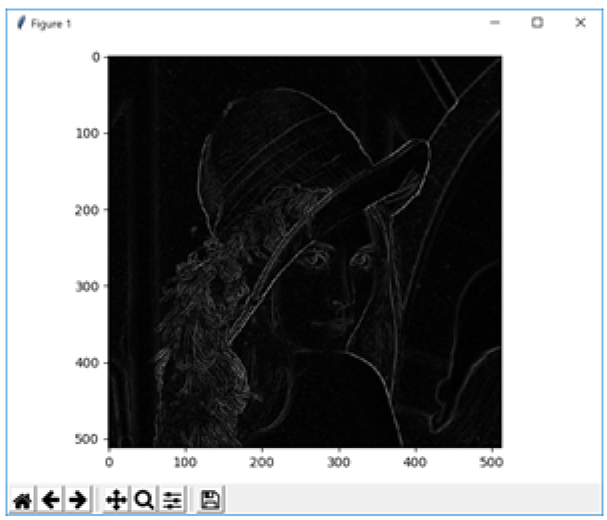
(2) Image – Erosion (internal gradient)

```
img_gi = img - morph.erosion(img, morph.square(3))   # internal gradient
```



(3) Dilation – Image (external gradient)

```
img_ge = morph.dilation(img, morph.square(3)) - img   # external gradient
```



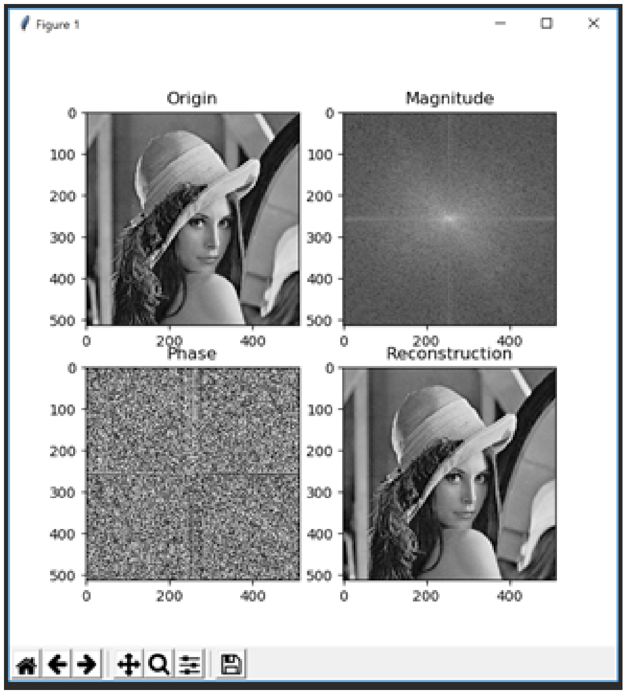
Frequency Domain

1. Fourier Transform

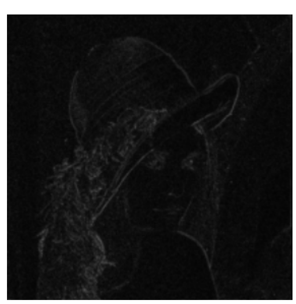
The Fourier transform is a mathematical formula that relates a signal sampled in time or space to the same signal sampled in frequency. In image processing, the Fourier transform can reveal important characteristics of a signal, namely, its frequency components. Fourier Transform depicts a signal (image) as sin and cos harmonic waves based on Euler's formula (that relates the sin and cos to complex numbers)

Discrete Fourier Transform implemented in fast Fourier algorithm using python numpy:

```
import numpy as np
from PIL import Image

img = Image.open("Lena.png")
img = img.convert('L')

f = np.fft.fft2(img)            # fast Fourier 2D
fs = np.fft.fftshift(f)         # shift low frequency to the middle
mag = np.log(np.abs(fs))        # log magnitude
phase = np.angle(fs)            # phase

from matplotlib import pyplot as plt
plt.figure(figsize=(6,6))
plt.subplot(2,2,1), plt.imshow(img,'gray'), plt.title('Origin')
plt.subplot(2,2,2), plt.imshow(mag,'gray'), plt.title('Magnitude')
plt.subplot(2,2,3), plt.imshow(phase,'gray'), plt.title('Phase')
#recon = np.abs(np.fft.ifft2(np.fft.ifftshift(fs)))
recon = np.abs(np.fft.ifft2( np.fft.ifftshift( np.exp(mag + 1j*phase) )));
plt.subplot(2,2,4), plt.imshow(recon,'gray'), plt.title('Reconstruction')
plt.show()

### low frequency is larger (larger energy)
plt.plot(mag.flatten())
plt.show()
```




```
### recon with only phase or mag
# ref: X. Hou and L. Zhang, "Saliency Detection: A Spectral Residual Approach," 2007 IEEE Conference on Computer Vision and Pattern Recognition, 2007, pp. 1-8, doi: 10.1109/CVPR.2007.383267.
#
recon_phase = np.abs(np.fft.ifft2( np.fft.ifftshift( np.exp(0 + 1j*phase) )))#**2 #could square according to the paper
#recon_mag = np.abs(np.fft.ifft2( np.fft.ifftshift( np.exp(mag + 1j*0) )))

plt.figure(figsize=(6,6))
plt.subplot(2,1,1), plt.imshow(recon_phase,'gray'), plt.title('recon_phase')
#plt.subplot(2,1,2), plt.imshow(recon_mag,'gray'), plt.title('recon_mag')
plt.show()
```



Fourier and Convolution

As the Convolutional Theorem states that convolutions on the spatial domain is equivalent as filtering on the frequency domain. We can perform fast convolution of large inputs by multiplying two Fourier transforms between the kernel and the image.

A toy example

```
import numpy as np
A = np.array([[8,1,6],[3,5,7],[4,9,2]])# The image, size M*N
B = np.ones([3,3])                     # The kernel, size P*Q
# pad A and B to at least M+P-1 * N+Q-1, usually the power of 2 to boost up fft
Ap = np.lib.pad(A, ((0,5),(0,5)), 'constant', constant_values=0)
Bp = np.lib.pad(B, ((0,5),(0,5)), 'constant', constant_values=0)

# convolution via FFT
C = np.fft.ifft2(np.multiply(np.fft.fft2(Ap),np.fft.fft2(Bp)))
C = np.real(C[1:4,1:4])
print(C)

# compare to spatial convolution
from scipy import ndimage
C2 = ndimage.convolve(A, B, mode='constant', cval=0.0)

print(C2)
```

both output [[17 30 19] [30 45 30] [21 30 23]] which is the convolution of A with kernel B.

Discrete Cosine Transform (DCT)

A DCT expresses a finite sequence of data points in terms of a sum of cosine functions oscillating at different frequencies. It is the cosine part of DFT and using only real numbers. Core of JPEG

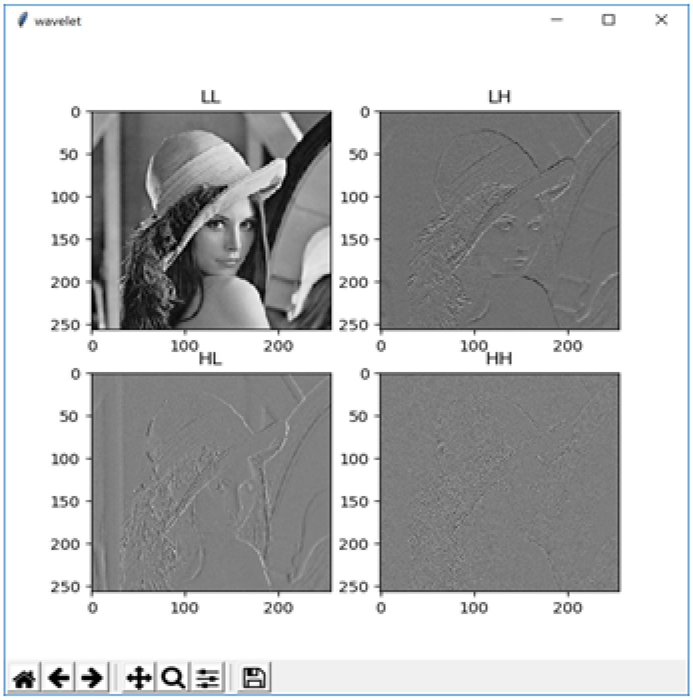
```
from skimage import io, color
img = color.rgb2gray(io.imread('lena.png'))
from scipy.fftpack import dct, idct
img_dct = dct(dct(img.T)).T   # 2D DCT, 1st col the row, or dct(dct(a, axis=0), axis=1)
recon = idct(idct(img_dct.T)).T
```

2. Wavelet

Discrete wavelet transforms (DWT) analyzes signals and images into progressively finer octave bands. As a time frequency information, wavelet multiresolution analysis enables the detection of patterns that are not visible in the raw data. This is the core of JPEG2000

PyWavelet package using:

```
import pywt
A, (H, V, D) = pywt.dwt2(img,'db1')
recon = pywt.idwt2((A, (H, V, D)),'db1')
```



For db1 or Haar transform, A is the down-sample, H is the horizontal, V vertical, D diagonal.
A simple 1D case can illustrate wavelet well:

image = [a,b,c,d] = > L (Low frequency) = [(a+b)/2, (c+d)/2] = [A1, A2]
H (High frequency) = [b-a, d-c] = [D1, D2]

e.g. [1, 2, 3, 4] => A: [1.5, 3.5] D: [1, 1]

reconstruction: [a, b, c, d] = [A1 – D1/2, A1 + D1/2, A2 – D2/2, A2 + D2/2]

a 2D Haar wavelet is to perform this to rows, then columns. And multi-level wavelet is to continue doing this in A …

if we want to do the implementation for better understanding of wavelet, (generalized) lifting scheme (https://en.wikipedia.org/wiki/Lifting_scheme) is preferred. Apart from the math, we can understand the z-transform in lifting scheme as the position, i.e. z**0 is current position, z**-1 is the previous neighbor, z**1 is the next neighbor, etc. Here I describe the process with less math and an example, a useful link:
https://www.mathworks.com/help/wavelet/ug/lifting-method-for-constructing-wavelets.html

for a signal X(n):

(1) Split: Partition X(n) into polyphase, a usual step is lazy wavelet: into odd components O(n) and even components E(n)

(2) Dual lifting: also called prediction, predict the odd polyphase component based on a linear combination of samples of the even polyphase component.
(3) Primal lifting: also called update, update the even polyphase component based on a linear combination of difference samples obtained from the predict step.
Instead of using z-transform and matrix format, I redo the [1, 2, 3, 4] example (lifting for Haar) here using only vectors:
(1) Split: O: [1, 3], E: [2, 4]
(2) Prediction: O_new = D = the prediction error = E – O = [1, 1]
(3) Update: E_new = E – ½ D = [1.5, 3.5]
The results are the same as the above, this is the magic behind the lifting schemes! In the reconstruction,
E = E_new + ½ O_new = [2, 4]
O = E – O_new = [1, 3]
Fully reconstructed. Understanding this example lead to direct implementations, often people multiply the result O_new and E_new by sqrt(2) for normalizations, anyway, I write the un-normalization version haar lifting here, they are quite simple and neat

```
def my_predict(O,E):
        return E - O

def my_update(O,E):
        return E - 1/2 * O

def my_iupdate(O,E):
        return E + 1/2 * O

def my_ipredict(O,E):
        return E - O
```

Test it

```
import numpy as np
O, E = np.array([1, 3]), np.array([2, 4])
O_new = my_predict(O,E)
E_new = my_update(O_new,E)
print(O_new,E_new)
E_r = my_iupdate(O_new,E_new)
O_r = my_ipredict(O_new,E_r)
print(O_r,E_r)
```

output:
[1 1] [1.5 3.5]

[1. 3.] [2. 4.]
That looks good.