

Python

Python 经典图像 处理

Scikit-image

K

2018-12-1

目录

一、	环境安装与配置	4
1、	需要的安装包	4
2、	下载并安装 anaconda	5
3、	简单测试	7
4、	skimage 包的子模块	8
二、	图像的读取、显示与保存	10
1、	从外部读取图片并显示	10
2、	程序自带图片	11
3、	保存图片	12
4、	图片信息	12
三、	图像像素的访问与裁剪	15
四、	图像数据类型及颜色空间转换	19
1、	图像数据类型及转换	19
2、	颜色空间及其转换	20
五、	图像的绘制	23
1、	用 figure 函数和 subplot 函数分别创建主窗口与子图	25
2、	用 subplots 来创建显示窗口与划分子图	27
3、	其它方法绘图并显示	29
六、	图像的批量处理	31
七、	图像的形变与缩放	35
1、	改变图片尺寸 resize	35
2、	按比例缩放 rescale	36
3、	旋转 rotate	36
4、	图像金字塔	37
八、	对比度与亮度调整	40
1、	gamma 调整	40
2、	log 对数调整	41
3、	判断图像对比度是否偏低	42
4、	调整强度	42
九、	直方图与均衡化	45
1、	计算直方图	45
2、	绘制直方图	45
3、	彩色图片三通道直方图	47
4、	直方图均衡化	47
十、	图像简单滤波	49
1、	sobel 算子	49
2、	roberts 算子	49
3、	scharr 算子	49
4、	prewitt 算子	50
5、	canny 算子	50
6、	gabor 滤波	51
7、	gaussian 滤波	52

8.median.....	53
9、水平、垂直边缘检测	54
10、交叉边缘检测.....	55
十一、 图像自动阈值分割.....	57
1、threshold_otsu.....	57
2、threshold_yen	58
3、threshold_li	58
4、threshold_isodata	59
5、threshold_adaptive	59
十二、 基本图形的绘制	61
1、画线条	61
2、画圆.....	62
3、多边形.....	63
4、椭圆.....	64
5、贝塞尔曲线.....	64
6、画空心圆	65
7、空心椭圆	66
十三、 基本形态学滤波	67
1、膨胀 (dilation).....	67
2、腐蚀 (erosion).....	68
3、开运算 (opening).....	69
4、闭运算 (closing)	70
5、白帽 (white-tophat).....	71
6、黑帽 (black-tophat).....	72
十四、 高级滤波	73
1、autolevel.....	73
2、bottomhat 与 tophat.....	74
3、enhance_contrast	75
4、entropy	75
5、equalize	76
6、gradient	77
7、其它滤波器.....	78
十五、 霍夫线变换	80
十六、 霍夫圆和椭圆变换.....	86
十七、 边缘与轮廓	91
十八、 高级形态学处理	96
1、凸包.....	96
2、连通区域标记	98
3、删除小块区域	101
十九、 骨架提取与分水岭算法	104
1、骨架提取	104
2、分水岭算法.....	108
二十、频率滤波.....	112
低通滤波器.....	112

高通滤波器.....	117
频率域高通滤波器.....	119
高斯高通滤波器.....	119
高斯低通滤波器.....	121
空间域的高斯滤波.....	122
巴特沃斯滤波器.....	123

一、 环境安装与配置

一提到数字图像处理编程，可能大多数人就会想到 **matlab**，但 **matlab** 也有自身的缺点：

- 1、不开源，价格贵
- 2、软件容量大。一般 3G 以上，高版本甚至达 5G 以上。
- 3、只能做研究，不易转化成软件。

因此，我们这里使用 **python** 这个脚本语言来进行数字图像处理。

要使用 **python**，必须先安装 **python**，一般是 2.7 版本以上，不管是在 **windows** 系统，还是 **linux** 系统，安装都是非常简单的。

要使用 **python** 进行各种开发和科学计算，还需要安装对应的包。这和 **matlab** 非常相似，只是 **matlab** 里面叫工具箱（**toolbox**），而 **python** 里面叫库或包。基于 **python** 脚本语言开发的数字图片处理包，其实很多，比如 **PIL**, **Pillow**, **opencv**, **scikit-image** 等。

对比这些包，**PIL** 和 **Pillow** 只提供最基础的数字图像处理，功能有限；**opencv** 实际上是一个 **c++** 库，只是提供了 **python** 接口，更新速度非常慢。到现在 **python** 都发展到了 3.5 版本，而 **opencv** 只支持到 **python 2.7** 版本；**scikit-image** 是基于 **scipy** 的一款图像处理包，它将图片作为 **numpy** 数组进行处理，正好与 **matlab** 一样，因此，我们最终选择 **scikit-image** 进行数字图像处理。

1/需要的安装包

因为 **scikit-image** 是基于 **scipy** 进行运算的，因此安装 **numpy** 和 **scipy** 是肯定的。要进行图片的显示，还需要安装 **matplotlib** 包，综合起来，需要的包有：



```
Python >= 2.6
```

```
Numpy >= 1.6.1
```

```
Cython >= 0.21
```

```
Six >=1.4
```

```
SciPy >=0.9
```

```
Matplotlib >= 1.1.0
```

```
NetworkX >= 1.8
```

```
Pillow >= 1.7.8
```

```
dask[array] >= 0.5.0
```



比较，安装起来非常费事，尤其是 **scipy**，在 **windows** 上基本安装不上。

但是不用怕，我们选择一款集成安装环境就行了，在此推荐 **Anaconda**，它把以上需要的包都集成在了一起，因此我们实际上从头到尾只需要安装 **Anaconda** 软件就行了，其它什么都不用装。

2、下载并安装 anaconda

先到 <https://www.continuum.io/downloads> 下载 anaconda，现在的版本有 **python2.7** 版本和 **python3.5** 版本，下载好对应版本、对应系统的 anaconda，它实际上是一个 **sh** 脚本文件，大约 **280M** 左右。

本系列以 **windows7+python3.5** 为例，因此我们下载如下图红框里的版本：

Anaconda for Windows

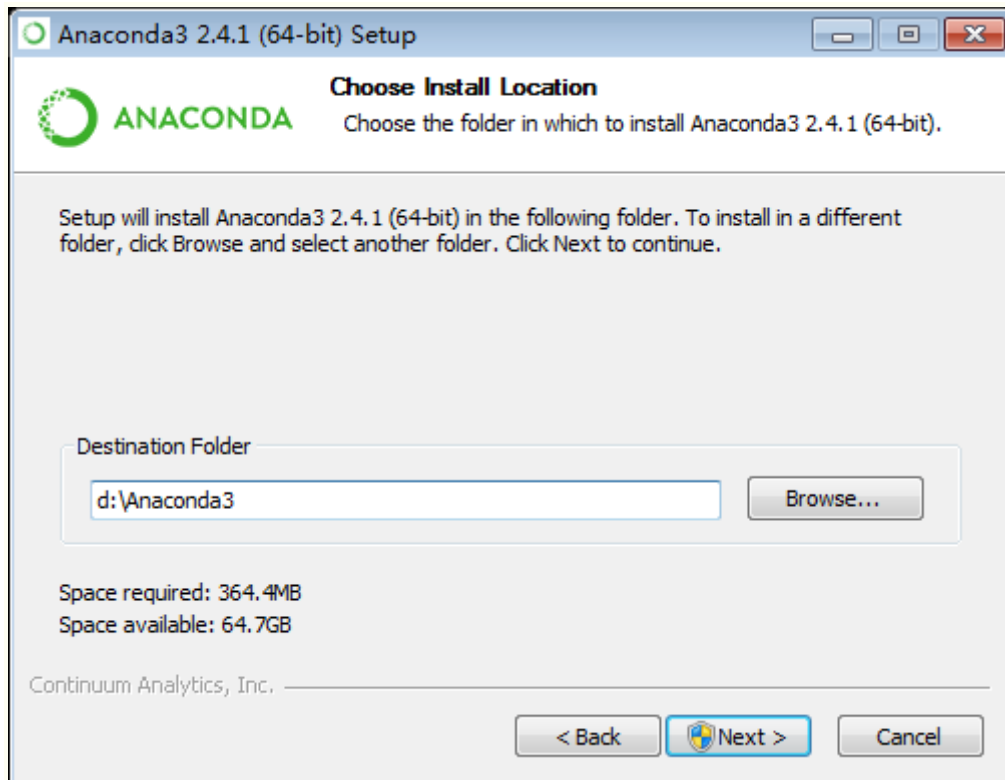
PYTHON 2.7	PYTHON 3.5
<div>Windows 64-bit Graphical Installer</div> <div>387M</div>	<div>Windows 64-bit Graphical Installer</div> <div>392M</div>
<div>Windows 32-bit Graphical Installer</div> <div>321M</div>	<div>Windows 32-bit Graphical Installer</div> <div>316M</div>

Behind a firewall? Use these [zipped Windows installers](#).

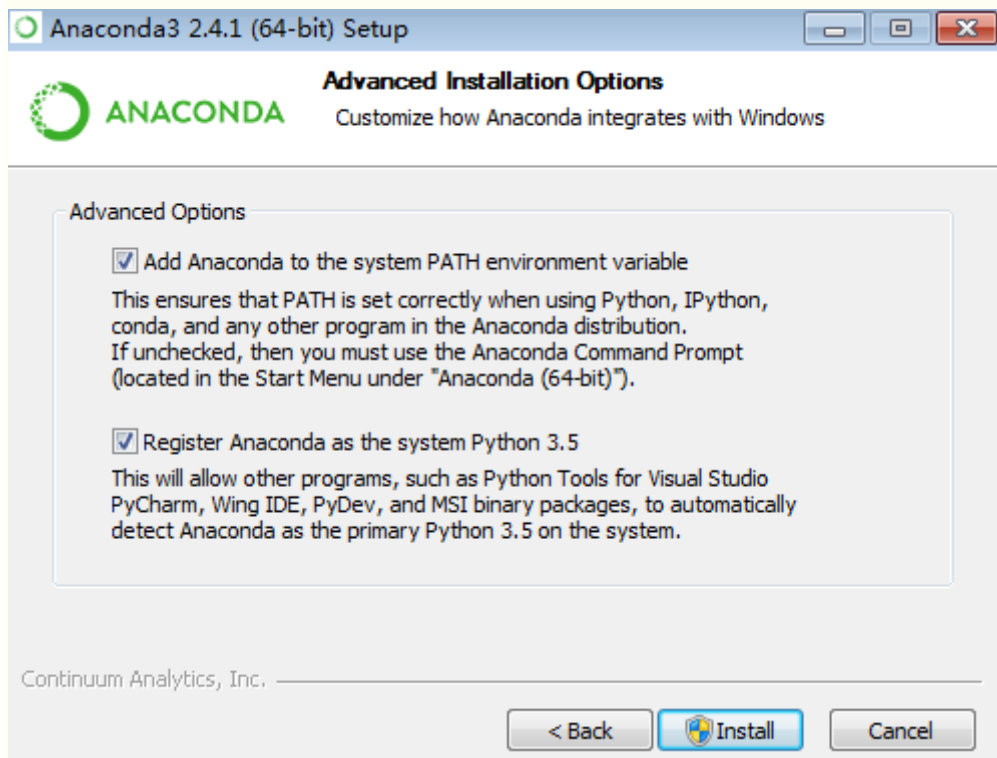
名称为： **Anaconda3-2.4.1-Windows-x86_64.exe**

是一个可执行的 **exe** 文件，下载完成好，直接双击就可以安装了。

在安装的时候，假设我们安装在 **D** 盘根目录，如：



并且将两个选项都选上，将安装路径写入环境变量。



然后等待安装完成就可以了。

安装完成后，打开 windows 的命令提示符：

输入 `conda list` 就可以查询现在安装了哪些库，常用的 `numpy`, `scipy` 名列其中。如果你还有什么包没有安装上，可以运行

`conda install ***` 来进行安装。（***为需要的包的名称）

如果某个包版本不是最新的，运行 `conda update ***` 就可以更新了。

3、简单测试

anaconda 自带了一款编辑器 `spyder`，我们以后就可以用这款编辑器来编写代码。

`spyder.exe` 放在安装目录下的 `Scripts` 里面，如我的是

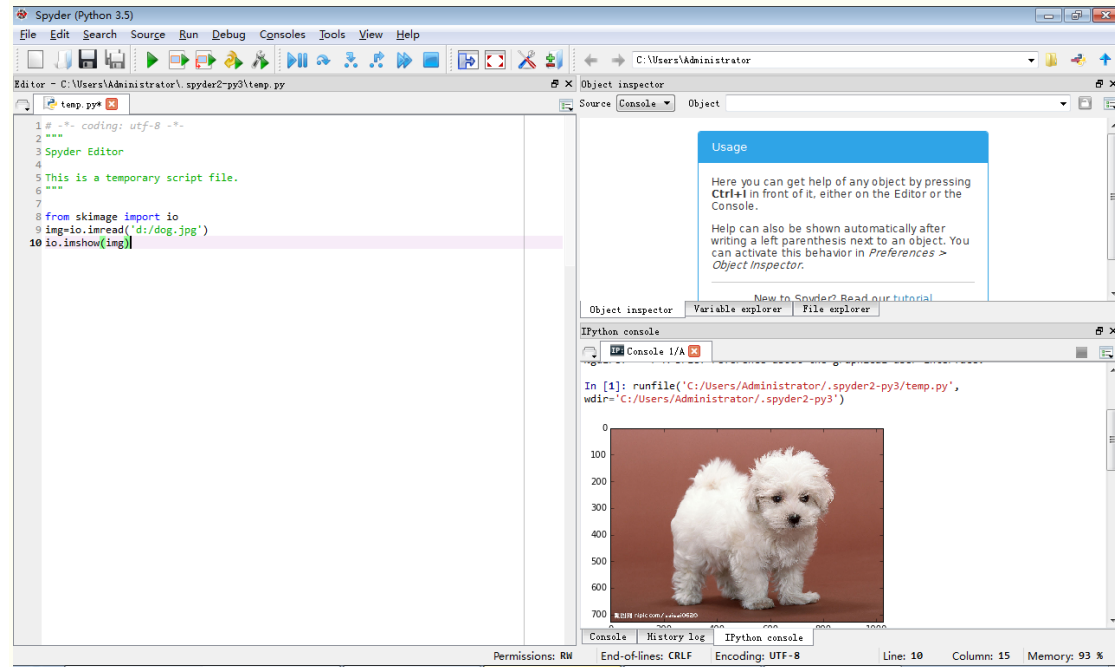
`D:/Anaconda3/Scripts/spyder.exe`，直接双击就能运行。我们可以右键发送到桌面快捷方式，以后运行就比较方便了。

我们简单编写一个程序来测试一下安装是否成功，该程序用来打开一张图片并显示。首先准备一张图片，然后打开 `spyder`，编写如下代码：

```
from skimage import io
img=io.imread('d:/dog.jpg')
io.imshow(img)
```

将其中的 `d:/dog.jpg` 改成你的图片位置

然后点击上面工具栏里的绿色三角进行运行，最终显示



如果右下角“`IPython console`”能显示出图片，说明我们的运行环境安装成功。

我们可以选择右上角的“`variable explorer`”来查看图片信息，如

restoration	图像恢复
util	通用函数

用到一些图片处理的操作函数时，需要导入对应的子模块，如果需要导入多个子模块，则用逗号隔开，如：

```
from skimage import io,data,color
```

二、 图像的读取、显示与保存

`skimage` 提供了 `io` 模块，顾名思义，这个模块是用来图片输入输出操作的。为了方便练习，也提供一个 `data` 模块，里面嵌套了一些示例图片，我们可以直接使用。

引入 `skimage` 模块可用：

1	<code>from skimage import io</code>
---	-------------------------------------

1、从外部读取图片并显示

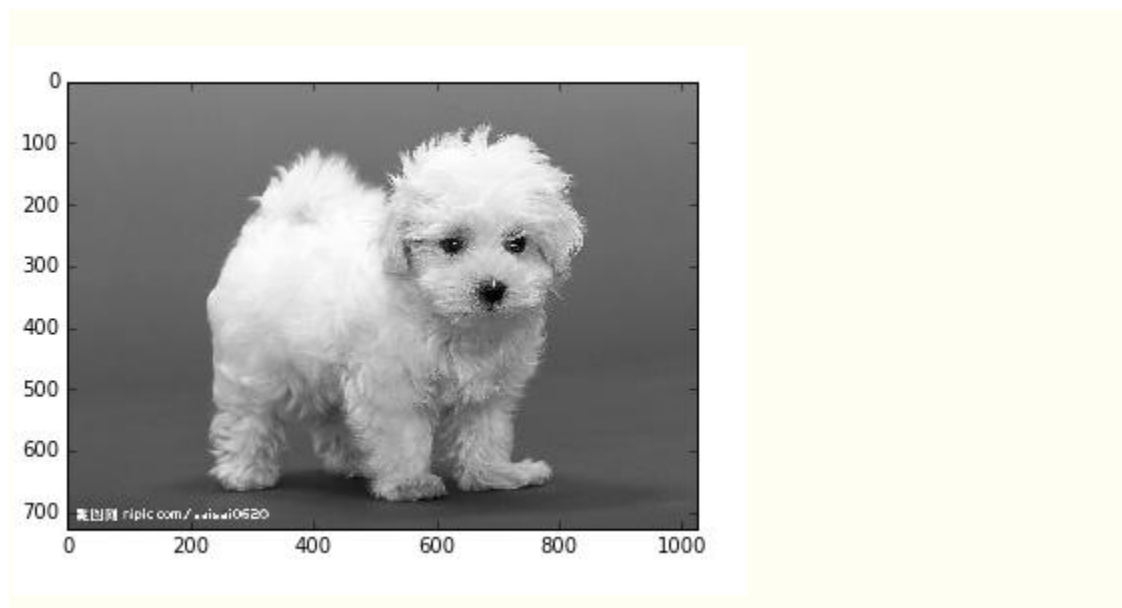
读取单张彩色 `rgb` 图片，使用 `skimage.io.imread (fname)` 函数，带一个参数，表示需要读取的文件路径。显示图片使用 `skimage.io.imshow (arr)` 函数，带一个参数，表示需要显示的 `arr` 数组（读取的图片以 `numpy` 数组形式计算）。

```
from skimage import io
img=io.imread('d:/dog.jpg')
io.imshow(img)
```



读取单张灰度图片，使用 `skimage.io.imread (fname, as_grey=True)` 函数，第一个参数为图片路径，第二个参数为 `as_grey`, `bool` 型值，默认为 `False`

```
from skimage import io
img=io.imread('d:/dog.jpg', as_grey=True)
io.imshow(img)
```



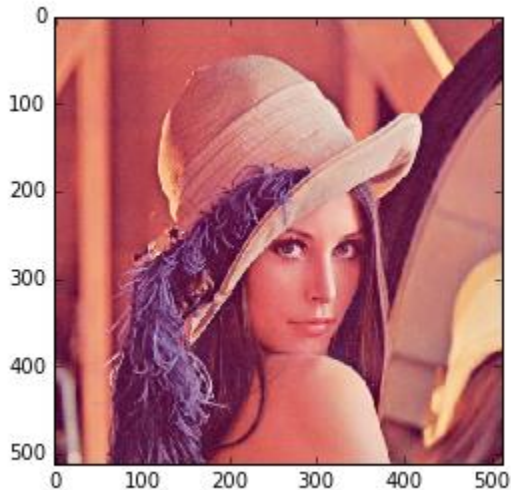
2、程序自带图片

`skimage` 程序自带了一些示例图片，如果我们不想从外部读取图片，就可以直接使用这些示例图片：

astronaut	宇航员 图片	coffee	一杯咖啡图 片	lena	lena 美女 图片
camera	拿相机 的人图 片	coins	硬币图片	moon	月亮 图片
checkerboard	棋盘图 片	horse	马图片	page	书页 图片
chelsea	小猫图 片	hubble_d eep_field	星空图片	text	文字 图片
clock	时钟图 片	immunohi stochemis try	结肠图片		

显示这些图片可用如下代码，不带任何参数

```
from skimage import io, data
img=data.lena()
io.imshow(img)
```



图片名对应的就是函数名，如 `camera` 图片对应的函数名为 `camera()`。这些示例图片存放在 `skimage` 的安装目录下面，路径名称为 `data_dir`，我们可以将这个路径打印出来看看：

```
from skimage import data_dir
print(data_dir)
```

显示为： `D:\Anaconda3\lib\site-packages\skimage\data`

也就是说，下面两行读取图片的代码效果是一样的：

```
from skimage import data_dir, data, io
img1=data.lena() #读取 lena 图片
img2=io.imread(data_dir+'lena.png') #读取 lena 图片
```

3、保存图片

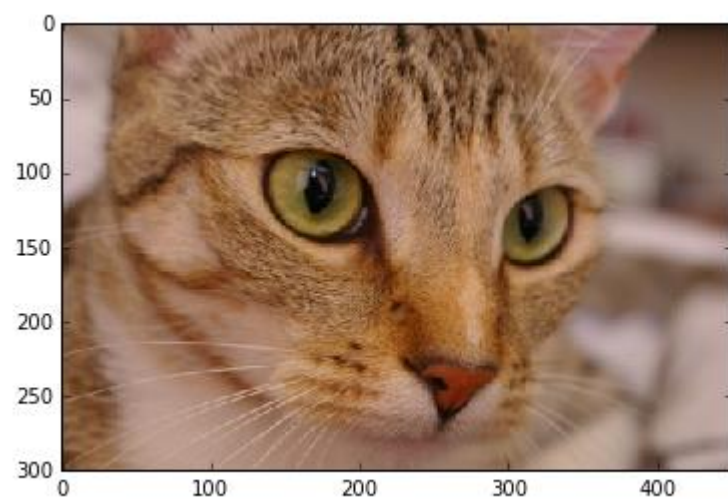
使用 `io` 模块的 `imsave (fname,arr)` 函数来实现。第一个参数表示保存的路径和名称，第二个参数表示需要保存的数组变量。

```
from skimage import io, data
img=data.chelsea()
io.imshow(img)
io.imsave('d:/cat.jpg',img)
```

保存图片的同时也起到了转换格式的作用。如果读取时图片格式为 `jpg` 图片，保存为 `png` 格式，则将图片从 `jpg` 图片转换为 `png` 图片并保存。

4、图片信息

如果我们想知道一些图片信息，可以在 `spyder` 编辑器的右上角显示：



三、 图像像素的访问与裁剪

图片读入程序中后，是以 `numpy` 数组存在的。因此对 `numpy` 数组的一切功能，对图片也适用。对数组元素的访问，实际上就是对图片像素点的访问。

彩色图片访问方式为：

`img[i,j,c]`

`i` 表示图片的行数，`j` 表示图片的列数，`c` 表示图片的通道数（RGB 三通道分别对应 0，1，2）。坐标是从左上角开始。

灰度图片访问方式为：

`gray[i,j]`

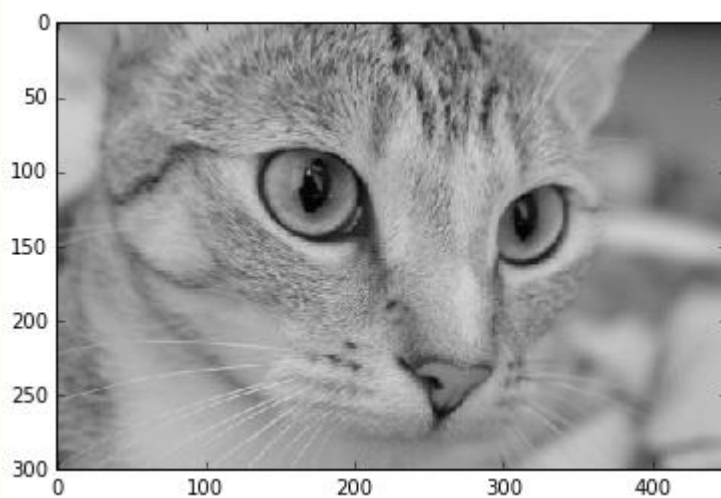
例 1：输出小猫图片的 G 通道中的第 20 行 30 列的像素值

```
from skimage import io, data
img=data.chelsea()
pixel=img[20, 30, 1]
print(pixel)
```

输出为 129

例 2：显示红色单通道图片

```
from skimage import io, data
img=data.chelsea()
R=img[:, :, 0]
io.imshow(R)
```



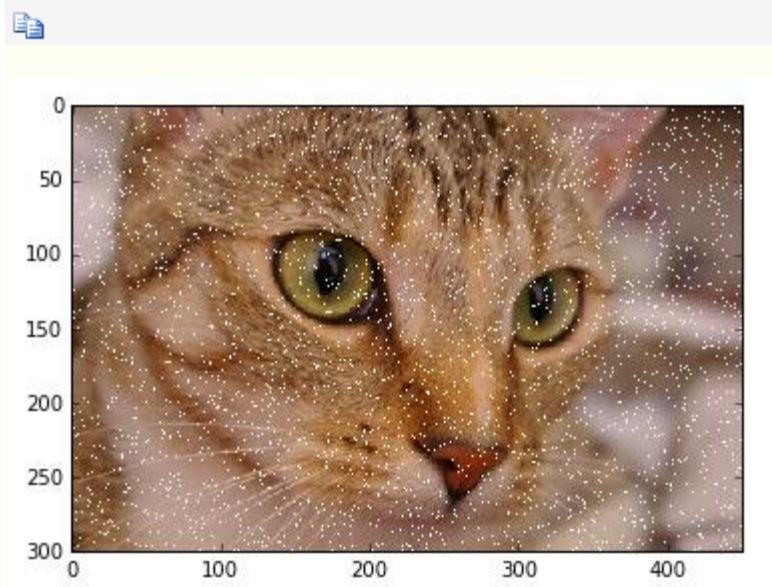
除了对像素进行读取，也可以修改像素值。

例 3：对小猫图片随机添加椒盐噪声


```
from skimage import io, data
import numpy as np
img=data.chelsea()

#随机生成 5000 个椒盐
rows, cols, dims=img. shape
for i in range(5000):
    x=np. random. randint(0, rows)
    y=np. random. randint(0, cols)
    img[x, y, :]=255

io.imshow(img)
```



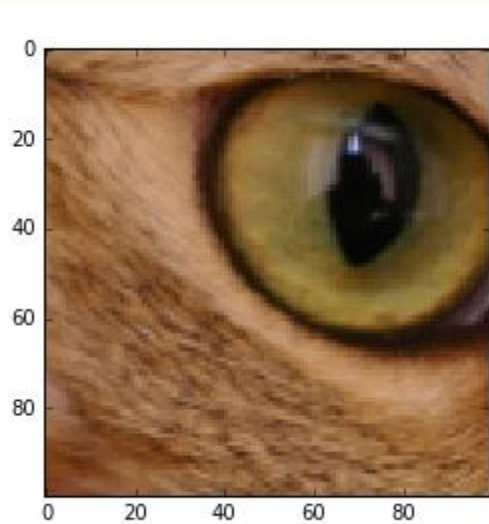
这里用到了 `numpy` 包里的 `random` 来生成随机数，`randint(0,cols)`表示随机生成一个整数，范围在 0 到 `cols` 之间。

用 `img[x,y,:]=255` 这句来对像素值进行修改，将原来的三通道像素值，变为 255

通过对数组的裁剪，就可以实现对图片的裁剪。

例 4：对小猫图片进行裁剪

```
from skimage import io, data
img=data.chelsea()
roi=img[80:180, 100:200, :]
io.imshow(roi)
```



对多个像素点进行操作，使用数组切片方式访问。切片方式返回的是以指定间隔下标访问 该数组的像素值。下面是有关灰度图像的一些例子：

```
img[i,:] = im[j,:] # 将第 j 行的数值赋值给第 i 行

img[:,i] = 100 # 将第 i 列的所有数值设为 100

img[:100,:50].sum() # 计算前 100 行、前 50 列所有数值的和

img[50:100,50:100] # 50~100 行，50~100 列（不包括第 100 行和第 100 列）

img[i].mean() # 第 i 行所有数值的平均值

img[:,-1] # 最后一列

img[-2,:] (or im[-2]) # 倒数第二行
```

最后我们再看两个对像素值进行访问和改变的例子：

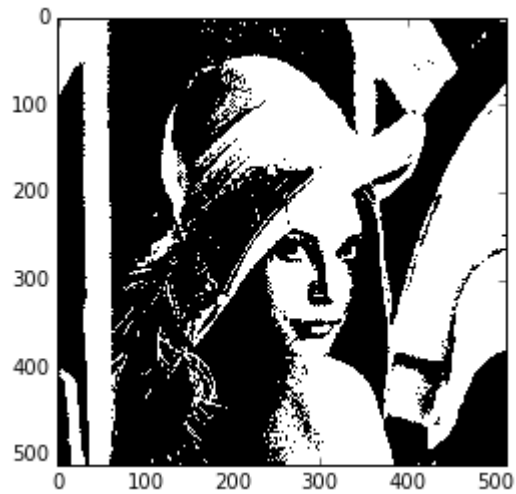
例 5：将 **lena** 图片进行二值化，像素值大于 128 的变为 1，否则变为 0

```
from skimage import io,data,color
img=data.lena()
img_gray=color.rgb2gray(img)
rows,cols=img_gray.shape
for i in range(rows):
    for j in range(cols):
        if (img_gray[i,j]<=0.5):
```

```

        img_gray[i,j]=0
    else:
        img_gray[i,j]=1
io.imshow(img_gray)

```



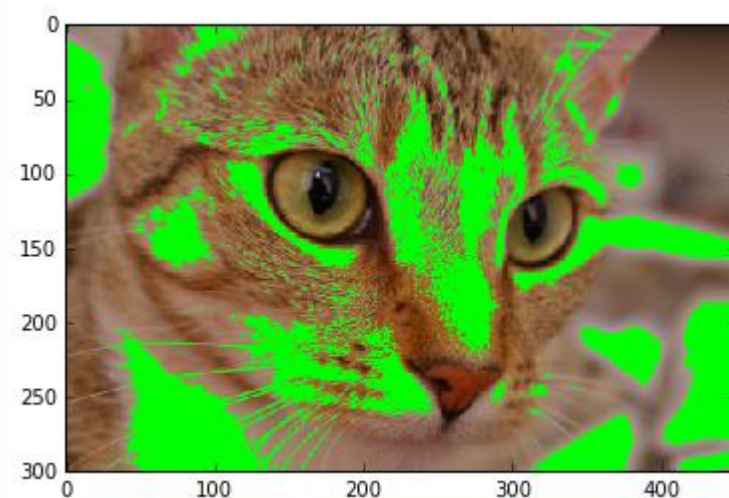
这个例子，使用了 `color` 模块的 `rgb2gray()` 函数，将彩色三通图转换成灰度图。转换结果为 `float64` 类型的数组，范围为 `[0,1]` 之间。

例 6:

```

from skimage import io,data
img=data.chelsea()
reddish = img[:, :, 0] > 170
img[reddish] = [0, 255, 0]
io.imshow(img)

```



这个例子先对 `R` 通道的所有像素值进行判断，如果大于 `170`，则将这个地方的像素值变为 `[0,255,0]`，即 `G` 通道值为 `255`，`R` 和 `B` 通道值为 `0`。

四、 图像数据类型及颜色空间转换

1、图像数据类型及转换

在 `skimage` 中，一张图片就是一个简单的 `numpy` 数组，数组的数据类型有很多种，相互之间也可以转换。这些数据类型及取值范围如下表所示：

Data type	Range
uint8	0 to 255
uint16	0 to 65535
uint32	0 to 232
float	-1 to 1 or 0 to 1
int8	-128 to 127
int16	-32768 to 32767
int32	-231 to 231 - 1

一张图片的像素值范围是`[0,255]`，因此默认类型是 `unit8`，可用如下代码查看数据类型：

```
from skimage import io,data
img=data.chelsea()
print(img.dtype.name)
```

在上面的表中，特别注意的是 `float` 类型，它的范围是`[-1,1]`或`[0,1]`之间。一张彩色图片转换为灰度图后，它的类型就由 `unit8` 变成了 `float`

1、unit8 转 float

```
from skimage import data,img_as_float
img=data.chelsea()
print(img.dtype.name)
dst=img_as_float(img)
print(dst.dtype.name)
```

输出：

```
uint8
float64
```

2、float 转 uint8

```
from skimage import img_as_ubyte
import numpy as np
img = np.array([0, 0.5, 1], dtype=float)
print(img.dtype.name)
dst=img_as_ubyte(img)
print(dst.dtype.name)
```

输出：

float64

uint8

float 转为 uint8，有可能会造成数据的损失，因此会有警告提醒。

除了这两种最常用的转换以外，其实有一些其它的类型转换，如下表：

Function name	Description
img_as_float	Convert to 64-bit floating point.
img_as_ubyte	Convert to 8-bit uint.
img_as_uint	Convert to 16-bit uint.
img_as_int	Convert to 16-bit int.

2、颜色空间及其转换

如前所述，除了直接转换可以改变数据类型外，还可以通过图像的颜色空间转换来改变数据类型。

常用的颜色空间有灰度空间、rgb 空间、hsv 空间和 cmyk 空间。颜色空间转换以后，图片类型都变成了 float 型。

所有的颜色空间转换函数，都放在 skimage 的 color 模块内。

例：rgb 转灰度图

```
from skimage import io,data,color
img=data.lena()
gray=color.rgb2gray(img)
io.imshow(gray)
```

其它的转换，用法都是一样的，列举常用的如下：

```
skimage.color.rgb2grey(rgb)
skimage.color.rgb2hsv(rgb)
```

```
skimage.color.rgb2lab(rgb)
skimage.color.gray2rgb(image)
skimage.color.hsv2rgb(hsv)
skimage.color.lab2rgb(lab)
```

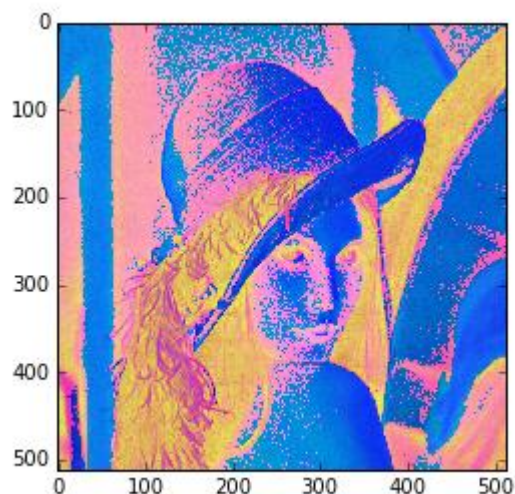
实际上，上面的所有转换函数，都可以用一个函数来代替

```
skimage.color.convert_colorspace(arr, fromspace, tospace)
```

表示将 **arr** 从 **fromspace** 颜色空间转换到 **tospace** 颜色空间。

例：rgb 转 hsv


```
from skimage import io, data, color
img=data.lena()
hsv=color.convert_colorspace(img, 'RGB', 'HSV')
io.imshow(hsv)
```



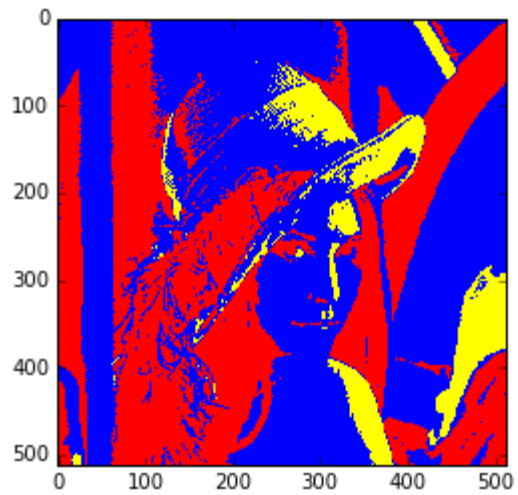
在 **color** 模块的颜色空间转换函数中，还有一个比较有用的函数是

skimage.color.label2rgb(arr)，可以根据标签值对图片进行着色。以后的图片分类后着色就可以用这个函数。

例：将 **lena** 图片分成三类，然后用默认颜色对三类进行着色

```

from skimage import io, data, color
import numpy as np
img=data.lena()
gray=color.rgb2gray(img)
rows,cols=gray.shape
labels=np.zeros([rows,cols])
for i in range(rows):
    for j in range(cols):
```

```
if(gray[i, j]<0.4):  
    labels[i, j]=0  
elif(gray[i, j]<0.75):  
    labels[i, j]=1  
else:  
    labels[i, j]=2  
dst=color.label2rgb(labels)  
io.imshow(dst)
```



五、 图像的绘制

实际上前面我们就已经用到了图像的绘制，如：

```
io.imshow(img)
```

这一行代码的实质是利用 **matplotlib** 包对图片进行绘制，绘制成功后，返回一个 **matplotlib** 类型的数据。因此，我们也可以这样写：

```
import matplotlib.pyplot as plt
plt.imshow(img)
```

imshow()函数格式为：

```
matplotlib.pyplot.imshow(X, cmap=None)
```

X：要绘制的图像或数组。

cmap：颜色图谱（**colormap**），默认绘制为 **RGB(A)**颜色空间。

其它可选的颜色图谱如下列表：

颜色图谱	描述
autumn	红-橙-黄
bone	黑-白，x 线
cool	青-洋红
copper	黑-铜
flag	红-白-蓝-黑
gray	黑-白
hot	黑-红-黄-白
hsv	hsv 颜色空间， 红-黄-绿-青-蓝-洋红-红
inferno	黑-红-黄
jet	蓝-青-黄-红
magma	黑-红-白
pink	黑-粉-白
plasma	绿-红-黄

颜色图谱	描述
prism	红-黄-绿-蓝-紫-...-绿模式
spring	洋红-黄
summer	绿-黄
viridis	蓝-绿-黄
winter	蓝-绿

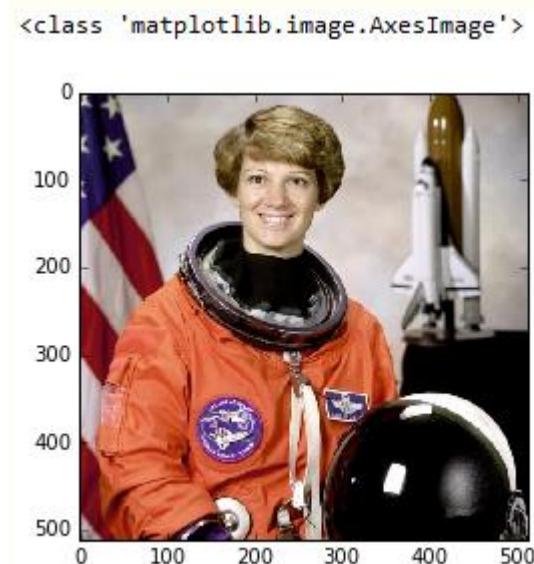
用的比较多的有 `gray,jet` 等，如：

```
plt.imshow(image,plt.cm.gray)
plt.imshow(img,cmap=plt.cm.jet)
```

在窗口上绘制完图片后，返回一个 **AxesImage** 对象。要在窗口上显示这个对象，我们可以调用 `show()` 函数来进行显示，但进行练习的时候（`ipython` 环境中），一般我们可以省略 `show()` 函数，也能自动显示出来。

```
from skimage import io,data
img=data.astronaut()
dst=io.imshow(img)
print(type(dst))
io.show()
```

显示为：



可以看到，类型是 `'matplotlib.image.AxesImage'`。显示一张图片，我们通常更愿意这样写：

```
import matplotlib.pyplot as plt
from skimage import io, data
img=data.astronaut()
plt.imshow(img)
plt.show()
```

matplotlib 是一个专业绘图的库，相当于 matlab 中的 plot, 可以设置多个 figure 窗口, 设置 figure 的标题，隐藏坐标尺，甚至可以使用 subplot 在一个 figure 中显示多张图片。一般我们可以这样导入 matplotlib 库：

```
import matplotlib.pyplot as plt
```

也就是说，我们绘图实际上用的是 matplotlib 包的 pyplot 模块。

1、用 figure 函数和 subplot 函数分别创建主窗口与子图

例: 分开并同时显示宇航员图片的三个通道

```
from skimage import data
import matplotlib.pyplot as plt
img=data.astronaut()
plt.figure(num='astronaut', figsize=(8,8)) #创建一个名为 astronaut 的窗口, 并设置大小

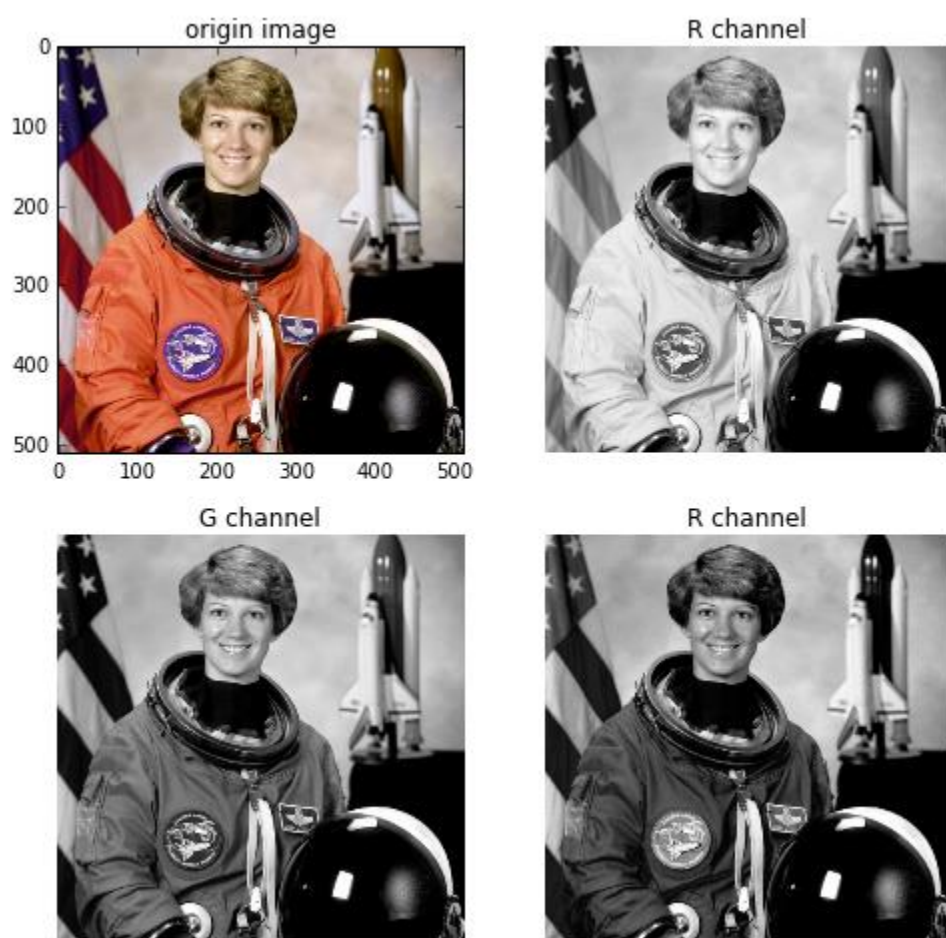
plt.subplot(2,2,1) #将窗口分为两行两列四个子图, 则可显示四幅图片
plt.title('origin image') #第一幅图片标题
plt.imshow(img) #绘制第一幅图片

plt.subplot(2,2,2) #第二个子图
plt.title('R channel') #第二幅图片标题
plt.imshow(img[:, :, 0], plt.cm.gray) #绘制第二幅图片, 且为灰度图
plt.axis('off') #不显示坐标尺寸

plt.subplot(2,2,3) #第三个子图
plt.title('G channel') #第三幅图片标题
plt.imshow(img[:, :, 1], plt.cm.gray) #绘制第三幅图片, 且为灰度图
plt.axis('off') #不显示坐标尺寸

plt.subplot(2,2,4) #第四个子图
plt.title('B channel') #第四幅图片标题
plt.imshow(img[:, :, 2], plt.cm.gray) #绘制第四幅图片, 且为灰度图
plt.axis('off') #不显示坐标尺寸
```

```
plt.show() #显示窗口
```



在图片绘制过程中，我们用 `matplotlib.pyplot` 模块下的 `figure()` 函数来创建显示窗口，该函数的格式为：

```
matplotlib.pyplot.figure(num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None)
```

所有参数都是可选的，都有默认值，因此调用该函数时可以不带任何参数，其中：

num: 整型或字符型都可以。如果设置为整型，则该整型数字表示窗口的序号。如果设置为字符型，则该字符串表示窗口的名称。用该参数来命名窗口，如果两个窗口序号或名相同，则后一个窗口会覆盖前一个窗口。

figsize: 设置窗口大小。是一个 `tuple` 型的整数，如 `figsize=(8, 8)`

dpi: 整形数字，表示窗口的分辨率。

facecolor: 窗口的背景颜色。

edgecolor: 窗口的边框颜色。

用 `figure()` 函数创建的窗口，只能显示一幅图片，如果想要显示多幅图片，则需要将这个窗口再划分为几个子图，在每个子图中显示不同的图片。我们可以使用 `subplot()` 函数来划分子图，函数格式为：

```
matplotlib.pyplot.subplot(nrows, ncols, plot_number)
```

nrows: 子图的行数。

ncols: 子图的列数。

plot_number: 当前子图的编号。

如：

```
plt.subplot(2, 2, 1)
```

则表示将 `figure` 窗口划分成了 2 行 2 列共 4 个子图，当前为第 1 个子图。我们有时也可以用这种写法：


```
plt.subplot(221)
```

两种写法效果是一样的。每个子图的标题可用 `title()` 函数来设置，是否使用坐标尺可用 `axis()` 函数来设置，如：

```
plt.subplot(221)
plt.title("first subwindow")
plt.axis('off')
```

2、用 subplots 来创建显示窗口与划分子图

除了上面那种方法创建显示窗口和划分子图，还有另外一种编写方法也可以，如下例：

```

import matplotlib.pyplot as plt
from skimage import data, color

img = data.immunohistochemistry()
hsv = color.rgb2hsv(img)

fig, axes = plt.subplots(2, 2, figsize=(7, 6))
ax0, ax1, ax2, ax3 = axes.ravel()

ax0.imshow(img)
ax0.set_title("Original image")

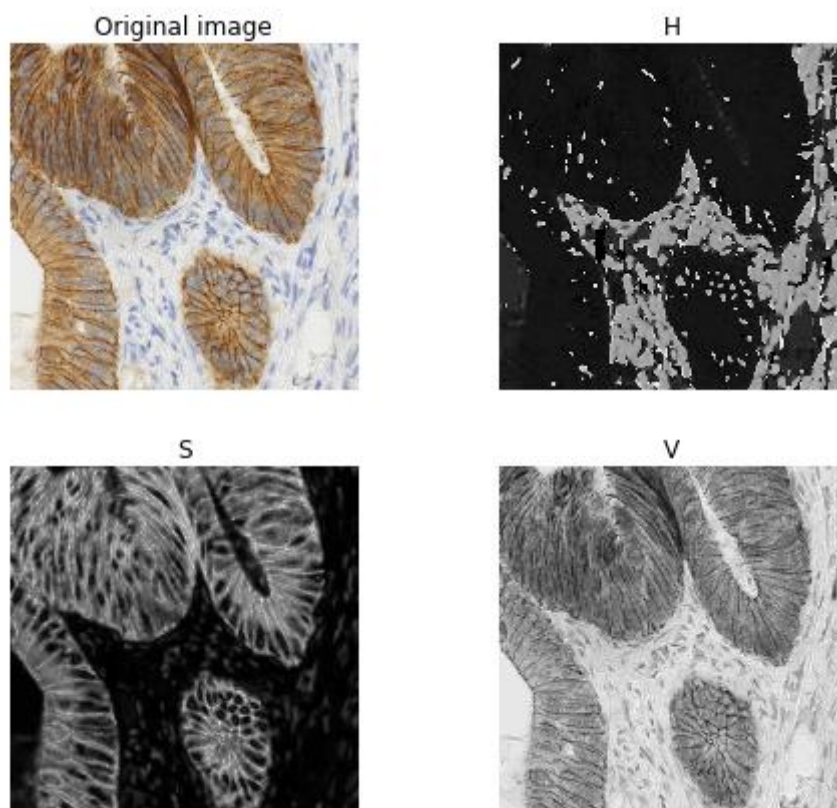
ax1.imshow(hsv[:, :, 0], cmap=plt.cm.gray)
ax1.set_title("H")
```

```
ax2.imshow(hsv[:, :, 1], cmap=plt.cm.gray)
ax2.set_title("S")

ax3.imshow(hsv[:, :, 2], cmap=plt.cm.gray)
ax3.set_title("V")

for ax in axes.ravel():
    ax.axis('off')

fig.tight_layout() #自动调整 subplot 间的参数
```



直接用 `subplots()` 函数来创建并划分窗口。注意，比前面的 `subplot()` 函数多了一个 `s`，该函数格式为：

```
matplotlib.pyplot.subplots(nrows=1, ncols=1)
```

nrows: 所有子图行数，默认为 1。

ncols: 所有子图列数，默认为 1。

返回一个窗口 **figure**，和一个 **tuple** 型的 **ax** 对象，该对象包含所有的子图，可结合 `ravel()` 函数列出所有子图，如：

```
fig, axes = plt.subplots(2, 2, figsize=(7, 6))
ax0, ax1, ax2, ax3 = axes.ravel()
```

创建了 2 行 2 列 4 个子图，分别取名为 ax0,ax1,ax2 和 ax3，每个子图的标题用 set_title() 函数来设置，如：

```
ax0.imshow(img)
ax0.set_title("Original image")
```

如果有多个子图，我们还可以使用 tight_layout()函数来调整显示的布局，该函数格式为：

```
matplotlib.pyplot.tight_layout(pad=1.08, h_pad=None, w_pad=None, rect=None)
```

所有的参数都是可选的，调用该函数时可省略所有的参数。

pad: 主窗口边缘和子图边缘间的间距，默认为 1.08

h_pad, w_pad: 子图边缘之间的间距，默认为 pad_inches

rect: 一个矩形区域，如果设置这个值，则将所有子图调整到这个矩形区域内。

一般调用为：

```
plt.tight_layout() #自动调整 subplot 间的参数
```

3、其它方法绘图并显示

除了使用 matplotlib 库来绘制图片，skimage 还有另一个子模块 viewer，也提供一个函数来显示图片。不同的是，它利用 Qt 工具来创建一块画布，从而在画布上绘制图像。

例：

```
from skimage import data
from skimage.viewer import ImageViewer

img = data.coins()
viewer = ImageViewer(img)
viewer.show()
```

最后总结一下，绘制和显示图片常用到的函数有：

函数名	功能	调用格式
figure	创建一个显示窗口	plt.figure(num=1,figsize=(8,8))
imshow	绘制图片	plt.imshow(image)
show	显示窗口	plt.show()

subplot	划分子图	plt.subplot(2,2,1)
title	设置子图标题(与 subplot 结合使用)	plt.title('origin image')
axis	是否显示坐标尺	plt.axis('off')
subplots	创建带有多个子图的 窗口	fig,axes=plt.subplots(2,2,figsize=(8,8))
ravel	为每个子图设置变量	ax0,ax1,ax2,ax3=axes.ravel()
set_title	设置子图标题 (与 axes 结合使用)	ax0.set_title('first window')
tight_layout	自动调整子图显示布 局	plt.tight_layout()

六、 图像的批量处理

有些时候，我们不仅要对一张图片进行处理，可能还会对一批图片处理。这时候，我们可以通过循环来执行处理，也可以调用程序自带的图片集合来处理。

图片集合函数为：

skimage.io.ImageCollection(load_pattern,load_func=None)

这个函数是放在 `io` 模块内的，带两个参数，第一个参数 `load_pattern`，表示图片组的路径，可以是一个 `str` 字符串。第二个参数 `load_func` 是一个回调函数，我们对图片进行批量处理就可以通过这个回调函数实现。回调函数默认为 `imread()`，即默认这个函数是批量读取图片。

先看一个例子：

```
import skimage.io as io
from skimage import data_dir
str=data_dir + '/*.png'
coll = io.ImageCollection(str)
print(len(coll))
```

显示结果为 **25**，说明系统自带了 **25** 张 **png** 的示例图片，这些图片都读取了出来，放在图片集合 `coll` 里。如果我们想显示其中一张图片，则可以在后加上一行代码：

```
io.imshow(coll[10])
```

显示为：



如果一个文件夹里，我们既存放了一些 **jpg** 格式的图片，又存放了一些 **png** 格式的图片，现在想把它们全部读取出来，该怎么做呢？

```
import skimage.io as io
from skimage import data_dir
str='d:/pic/*.jpg:d:/pic/*.png'
coll = io.ImageCollection(str)
```



```
print(len(coll))
```

注意这个地方'd:/pic/*.jpg:d:/pic/*.png'，是两个字符串合在一起的，第一个是'd:/pic/*.jpg'，第二个是'd:/pic/*.png'，合在一起后，中间用冒号来隔开，这样就可以把d:/pic/文件夹下的jpg和png格式的图片都读取出来。如果还想读取存放在其它地方的图片，也可以一并加进去，只是中间同样用冒号来隔开。

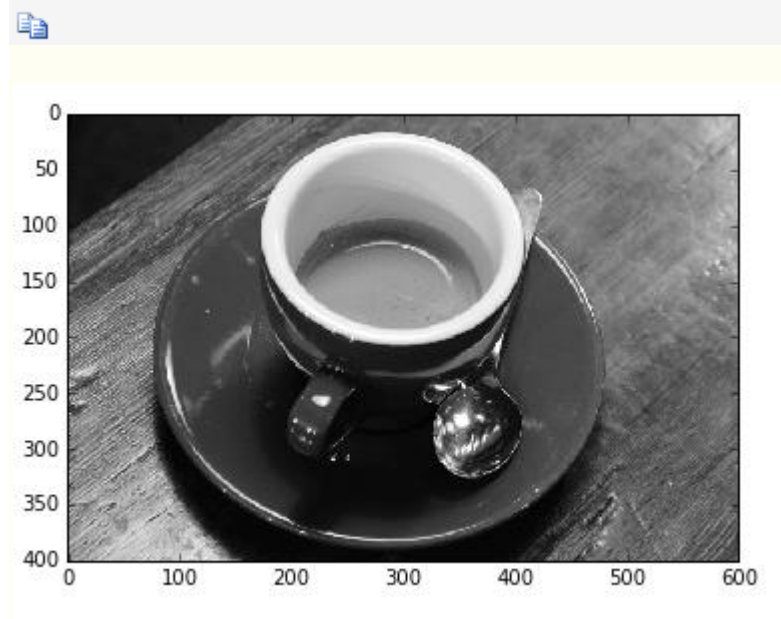
io.ImageCollection()这个函数省略第二个参数，就是批量读取。如果我们不是想批量读取，而是其它批量操作，如批量转换为灰度图，那又该怎么做呢？

那就需要先定义一个函数，然后将这个函数作为第二个参数，如：

```
from skimage import data_dir, io, color

def convert_gray(f):
    rgb=io.imread(f)
    return color.rgb2gray(rgb)

str=data_dir+'/*.png'
coll = io.ImageCollection(str, load_func=convert_gray)
io.imshow(coll[10])
```



这种批量操作对视频处理是极其有用的，因为视频就是一系列的图片组合

```
from skimage import data_dir, io, color

class AVILoader:
    video_file = 'myvideo.avi'

    def __call__(self, frame):
```

```

        return video_read(self.video_file, frame)

avi_load = AVILoader()

frames = range(0, 1000, 10) # 0, 10, 20, ...
ic = io.ImageCollection(frames, load_func=avi_load)

```

这段代码的意思，就是将 **myvideo.avi** 这个视频中每隔 **10** 帧的图片读取出来，放在图片集合中。

得到图片集合以后，我们还可以将这些图片连接起来，构成一个维度更高的数组，连接图片的函数为：

```
skimage.io.concatenate_images(ic)
```

带一个参数，就是以上的图片集合，如：

```

from skimage import data_dir, io, color

coll = io.ImageCollection('d:/pic/*.jpg')
mat=io.concatenate_images(coll)

```

使用 `concatenate_images(ic)` 函数的前提是读取的这些图片尺寸必须一致，否则会出错。我们看看图片连接前后的维度变化：

```

from skimage import data_dir, io, color

coll = io.ImageCollection('d:/pic/*.jpg')
print(len(coll))      #连接的图片数量
print(coll[0].shape)  #连接前的图片尺寸，所有的都一样
mat=io.concatenate_images(coll)
print(mat.shape)      #连接后的数组尺寸

```

显示结果：

```

2
(870, 580, 3)
(2, 870, 580, 3)

```

可以看到，将 **2** 个 **3** 维数组，连接成了一个 **4** 维数组

如果我们对图片进行批量操作后，想把操作后的结果保存起来，也是可以办到的。

例：把系统自带的所有 **png** 示例图片，全部转换成 **256*256** 的 **jpg** 格式灰度图，保存在 **d:/data/** 文件夹下

改变图片的大小，我们可以使用 `tranform` 模块的 `resize()` 函数，后续会讲到这个模块。

```

from skimage import data_dir, io, transform, color
import numpy as np

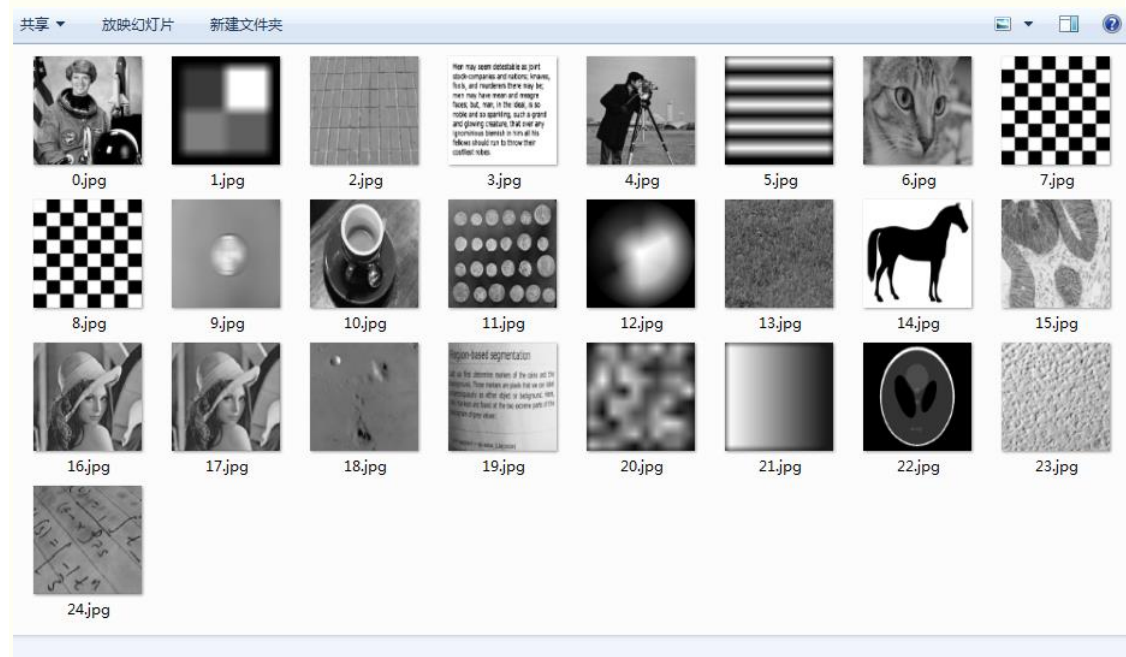
def convert_gray(f):
    rgb=io.imread(f)      #依次读取 rgb 图片
    gray=color.rgb2gray(rgb)  #将 rgb 图片转换成灰度图
    dst=transform.resize(gray, (256, 256))  #将灰度图片大小转换为
256*256
    return dst

str=data_dir+'/*.png'
coll = io.ImageCollection(str, load_func=convert_gray)
for i in range(len(coll)):
    io.imsave('d:/data/'+np.str(i)+' .jpg', coll[i])  #循环保存图片

```



结果:



七、 图像的形变与缩放

图像的形变与缩放，使用的是 `skimage` 的 `transform` 模块，函数比较多，功能齐全。


1、改变图片尺寸 `resize`

函数格式为：

`skimage.transform.resize(image, output_shape)`

image: 需要改变尺寸的图片


output_shape: 新的图片尺寸

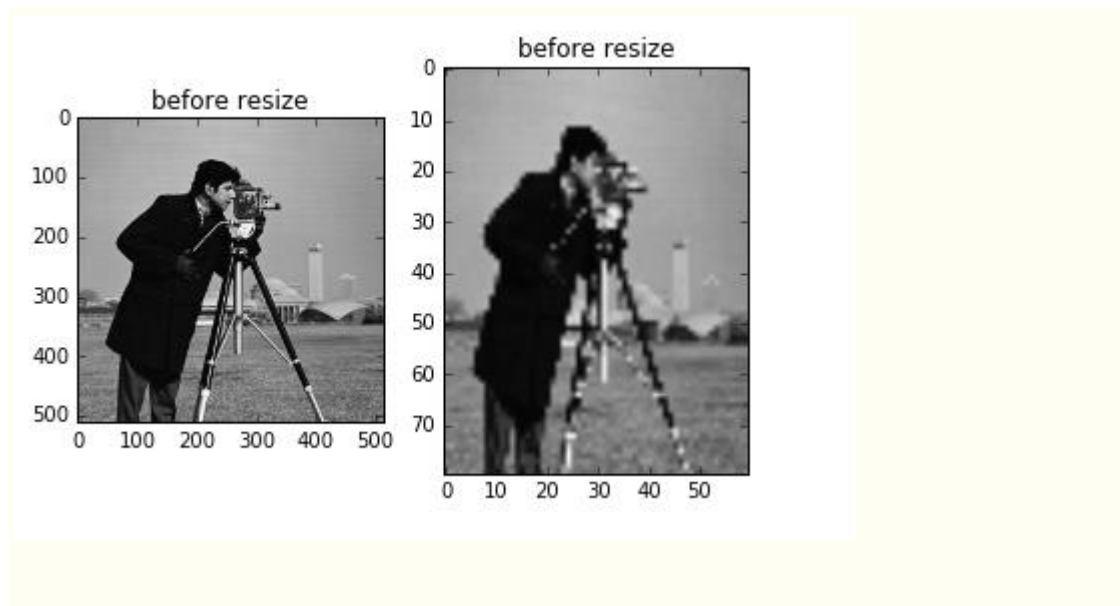
```
from skimage import transform, data
import matplotlib.pyplot as plt
img = data.camera()
dst=transform.resize(img, (80, 60))
plt.figure('resize')

plt.subplot(121)
plt.title('before resize')
plt.imshow(img, plt.cm.gray)

plt.subplot(122)
plt.title('before resize')
plt.imshow(dst, plt.cm.gray)

plt.show()
```

将 `camera` 图片由原来的 `512*512` 大小，变成了 `80*60` 大小。从下图中的坐标尺，我们能够看出来：



2、按比例缩放 rescale

函数格式为:

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

scale 参数可以是单个 **float** 数, 表示缩放的倍数, 也可以是一个 **float** 型的 **tuple**, 如 **[0.2,0.5]**,表示将行列数分开进行缩放

```
from skimage import transform, data
img = data.camera()
print(img.shape) #图片原始大小
print(transform.rescale(img, 0.1).shape) #缩小为原来图片大小的 0.1 倍
print(transform.rescale(img, [0.5, 0.25]).shape) #缩小为原来图片行数
一半, 列数四分之一
print(transform.rescale(img, 2).shape) #放大为原来图片大小的 2 倍
```

结果为:

```
(512, 512)
(51, 51)
(256, 128)
(1024, 1024)
```

3、旋转 rotate

```
skimage.transform.rotate(image, angle[, ...],resize=False)
```

angle 参数是个 **float** 类型数, 表示旋转的度数

resize 用于控制在旋转时, 是否改变大小, 默认为 **False**

```

from skimage import transform, data
import matplotlib.pyplot as plt
img = data.camera()
print(img.shape) #图片原始大小
img1=transform.rotate(img, 60) #旋转 90 度，不改变大小
print(img1.shape)
img2=transform.rotate(img, 30,resize=True) #旋转 30 度，同时改变大小
print(img2.shape)

plt.figure('resize')

plt.subplot(121)
plt.title('rotate 60')
plt.imshow(img1,plt.cm.gray)

plt.subplot(122)
plt.title('rotate 30')
plt.imshow(img2,plt.cm.gray)

plt.show()

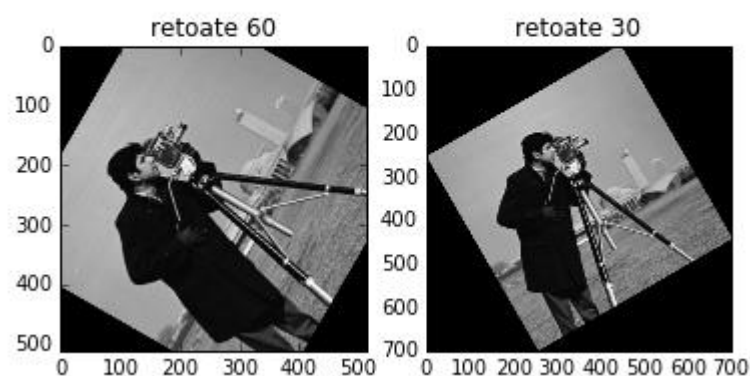
```

显示结果:

```

(512, 512)
(512, 512)
(700, 700)

```

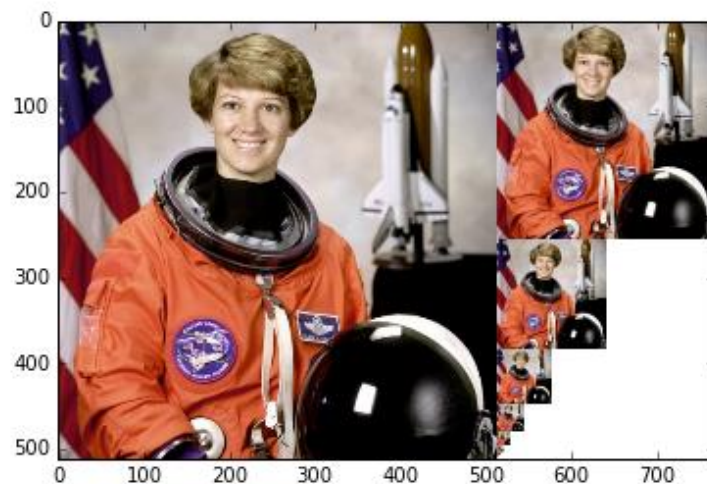


4、图像金字塔

以多分辨率来解释图像的一种有效但概念简单的结构就是图像金字塔。图像金字塔最初用于机器视觉和图像压缩，一幅图像的金字塔是一系列以金字塔形状排列的分辨率逐步降低的图像集合。金字塔的底部是待处理图像的高分辨率表示，而顶部是低分辨率的近似。当向金字塔的上层移动时，尺寸和分辨率就降低。

在此，我们举一个高斯金字塔的应用实例，函数原型为：

```
skimage.transform.pyramid_gaussian(image, downscale=2)
downscale 控制着金字塔的缩放比例
import numpy as np
import matplotlib.pyplot as plt
from skimage import data, transform
image = data.astronaut() #载入宇航员图片
rows, cols, dim = image.shape #获取图片的行数，列数和通道数
pyramid = tuple(transform.pyramid_gaussian(image, downscale=2)) #产生高斯金字塔图像
#共生成了  $\log(512)=9$  幅金字塔图像，加上原始图像共 10 幅，pyramid[0]-pyramid[1]
composite_image = np.ones((rows, cols + cols / 2, 3),
dtype=np.double) #生成背景
composite_image[:rows, :cols, :] = pyramid[0] #融合原始图像
i_row = 0
for p in pyramid[1:]:
    n_rows, n_cols = p.shape[:2]
    composite_image[i_row:i_row + n_rows, cols:cols + n_cols] = p #
    i_row += n_rows
plt.imshow(composite_image)
plt.show()
```

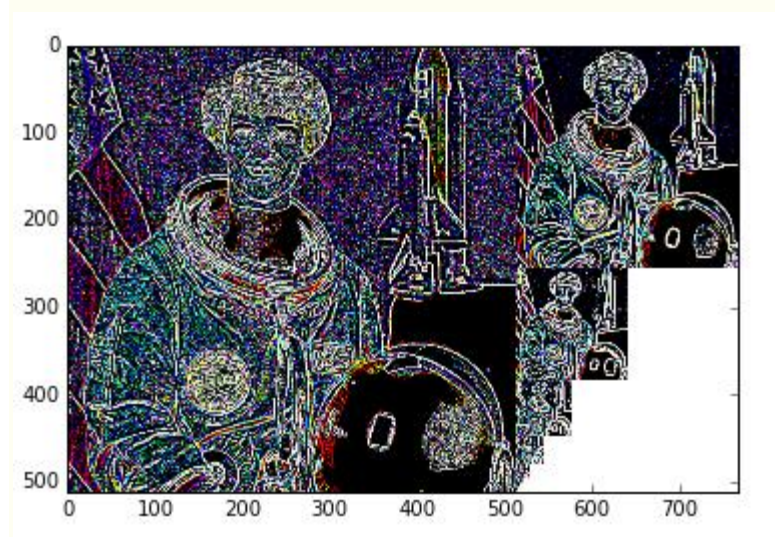


Index	Type	Size	Value
0	float64	(512, 512, 3)	array([[[0.60392157, 0.57647059, 0... [0.42745098, 0.403921 ...
1	float64	(256, 256, 3)	array([[[5.70355398e-01, 5.493192... [3.765 ...
2	float64	(128, 128, 3)	array([[[0.63013534, 0.61247746, 0... [0.62751387, 0.610783 ...
3	float64	(64, 64, 3)	array([[[0.69085518, 0.66881871, 0... [0.27221277, 0.244216 ...
4	float64	(32, 32, 3)	array([[[0.45417524, 0.42708602, 0... [0.16566446, 0.136332 ...
5	float64	(16, 16, 3)	array([[[0.26640978, 0.23532107, 0... [0.49727513, 0.469722 ...
6	float64	(8, 8, 3)	array([[[0.35630078, 0.32297746, 0... [0.67742879, 0.646883 ...
7	float64	(4, 4, 3)	array([[[0.51493432, 0.45885025, 0... [0.65852397, 0.593295 ...
8	float64	(2, 2, 3)	array([[[0.59223425, 0.4730815 , 0... [0.61556371, 0.559794 ...
9	float64	(1, 1, 3)	array([[[0.55514703, 0.41474292, 0.37833363]]])

上右图，就是 10 张金字塔图像，下标为 0 的表示原始图像，后面每层的图像行和列变为上一层的一半，直至变为 1

除了高斯金字塔外，还有其它的金字塔，如：

`skimage.transform.pyramid_laplacian(image, downscale=2):`



八、 对比度与亮度调整

图像亮度与对比度的调整，是放在 `skimage` 包的 `exposure` 模块里面

1、gamma 调整

原理： $I = I^g$

对原图像的像素，进行幂运算，得到新的像素值。公式中的 g 就是 `gamma` 值。

如果 `gamma > 1`，新图像比原图像暗

如果 `gamma < 1`，新图像比原图像亮

函数格式为：`skimage.exposure.adjust_gamma(image, gamma=1)`

`gamma` 参数默认为 1，原像不发生变化。

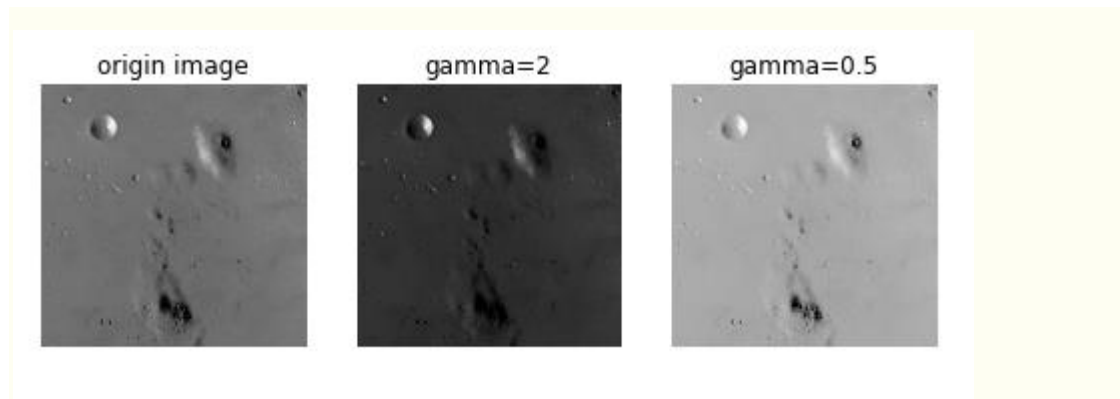
```
from skimage import data, exposure, img_as_float
import matplotlib.pyplot as plt
image = img_as_float(data.moon())
gam1 = exposure.adjust_gamma(image, 2) #调暗
gam2 = exposure.adjust_gamma(image, 0.5) #调亮
plt.figure('adjust_gamma', figsize=(8, 8))

plt.subplot(131)
plt.title('origin image')
plt.imshow(image, plt.cm.gray)
plt.axis('off')

plt.subplot(132)
plt.title('gamma=2')
plt.imshow(gam1, plt.cm.gray)
plt.axis('off')

plt.subplot(133)
plt.title('gamma=0.5')
plt.imshow(gam2, plt.cm.gray)
plt.axis('off')

plt.show()
```



2、log 对数调整

这个刚好和 `gamma` 相反

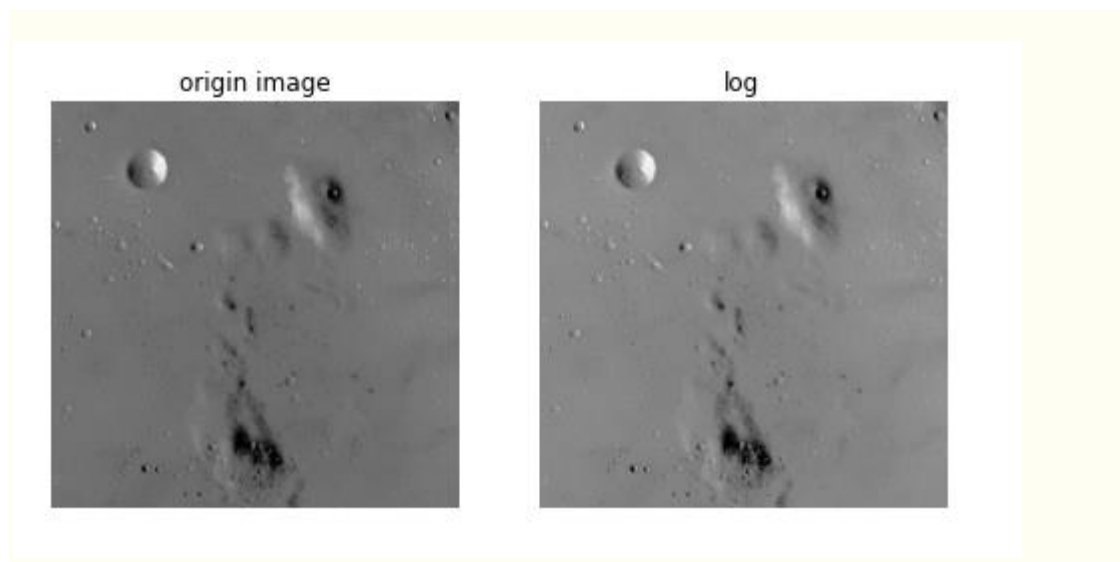
原理: $I = \log(I)$

```
from skimage import data, exposure, img_as_float
import matplotlib.pyplot as plt
image = img_as_float(data.moon())
gam1 = exposure.adjust_log(image) #对数调整
plt.figure('adjust_gamma', figsize=(8, 8))

plt.subplot(121)
plt.title('origin image')
plt.imshow(image, plt.cm.gray)
plt.axis('off')

plt.subplot(122)
plt.title('log')
plt.imshow(gam1, plt.cm.gray)
plt.axis('off')

plt.show()
```



3、判断图像对比度是否偏低

函数: `is_low_contrast(img)`

返回一个 `bool` 型值

```
from skimage import data, exposure
image = data.moon()
result = exposure.is_low_contrast(image)
print(result)
```

输出为 `False`

4、调整强度

函数:

`skimage.exposure.rescale_intensity(image, in_range='image', out_range='dtype')`

`in_range` 表示输入图片的强度范围，默认为 `'image'`，表示用图像的最大/最小像素值作为范围

`out_range` 表示输出图片的强度范围，默认为 `'dtype'`，表示用图像的类型的最小/最大值作为范围

默认情况下，输入图片的 `[min,max]` 范围被拉伸到 `[dtype.min, dtype.max]`，如果 `dtype=uint8`，那么 `dtype.min=0`，`dtype.max=255`

```
import numpy as np
from skimage import exposure
image = np.array([51, 102, 153], dtype=np.uint8)
```

```
mat=exposure.rescale_intensity(image)
print(mat)
```

输出为[0 127 255]

即像素最小值由 51 变为 0，最大值由 153 变为 255，整体进行了拉伸，但是数据类型没有变，还是 uint8

前面我们讲过，可以通过 `img_as_float()` 函数将 `uint8` 类型转换为 `float` 型，实际上还有更简单的方法，就是乘以 1.0

```
import numpy as np
image = np.array([51, 102, 153], dtype=np.uint8)
print(image*1.0)
```

即由[51,102,153]变成了[51. 102. 153.]

而 `float` 类型的范围是[0,1]，因此对 `float` 进行 `rescale_intensity` 调整后，范围变为[0,1]，而不是[0,255]

```
import numpy as np
from skimage import exposure
image = np.array([51, 102, 153], dtype=np.uint8)
tmp=image*1.0
mat=exposure.rescale_intensity(tmp)
print(mat)
```

结果为[0. 0.5 1.]

如果原始像素值不想被拉伸，只是等比例缩小，就使用 `in_range` 参数，如：

```
import numpy as np
from skimage import exposure
image = np.array([51, 102, 153], dtype=np.uint8)
tmp=image*1.0
mat=exposure.rescale_intensity(tmp, in_range=(0, 255))
print(mat)
```

输出为：[0.2 0.4 0.6]，即原像素值除以 255

如果参数 `in_range` 的[main,max]范围要比原始像素值的范围[min,max] 大或者小，那就进行裁剪，如：

```
mat=exposure.rescale_intensity(tmp, in_range=(0, 102))
print(mat)
```

输出[0.5 1. 1.]，即原像素值除以 102，超出 1 的变为 1

如果一个数组里面有负数，现在想调整到正数，就使用 `out_range` 参数。如：

```
import numpy as np
from skimage import exposure
image = np.array([-10, 0, 10], dtype=np.int8)
```

```
mat=exposure.rescale_intensity(image, out_range=(0, 127))  
print(mat)
```

输出[0 63 127]

九、直方图与均衡化

在图像处理中，直方图是非常重要的，也是非常有用的一个处理要素。

在 `skimage` 库中对直方图的处理，是放在 `exposure` 这个模块中。

1、计算直方图

函数: `skimage.exposure.histogram(image, nbins=256)`

在 `numpy` 包中，也提供了一个计算直方图的函数 `histogram()`，两者大同小义。

返回一个 `tuple (hist, bins_center)`，前一个数组是直方图的统计量，后一个数组是每个 `bin` 的中间值

```
import numpy as np
from skimage import exposure, data
image = data.camera()*1.0
hist1=np.histogram(image, bins=2)    #用 numpy 包计算直方图
hist2=exposure.histogram(image, nbins=2)  #用 skimage 计算直方图
print(hist1)
print(hist2)
```

输出:

```
(array([107432, 154712], dtype=int64), array([ 0. , 127.5, 255. ]))
(array([107432, 154712], dtype=int64), array([ 63.75, 191.25]))
```

分成两个 `bin`，每个 `bin` 的统计量是一样的，但 `numpy` 返回的是每个 `bin` 的两端的范围值，而 `skimage` 返回的是每个 `bin` 的中间值

2、绘制直方图

绘图都可以调用 `matplotlib.pyplot` 库来进行，其中的 `hist` 函数可以直接绘制直方图。

调用方式:

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

`hist` 的参数非常多，但常用的就这六个，只有第一个是必须的，后面四个可选

arr: 需要计算直方图的一维数组

bins: 直方图的柱数，可选项，默认为 10

normed: 是否将得到的直方图向量归一化。默认为 0

facecolor: 直方图颜色

edgecolor: 直方图边框颜色

alpha: 透明度

histtype: 直方图类型, 'bar', 'barstacked', 'step', 'stepfilled'

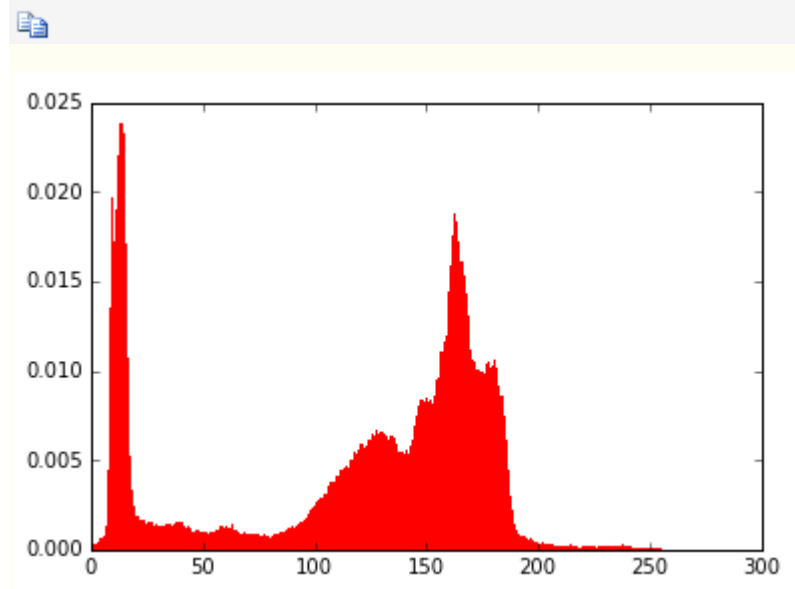
返回值 :

n: 直方图向量, 是否归一化由参数 **normed** 设定

bins: 返回各个 bin 的区间范围

patches: 返回每个 bin 里面包含的数据, 是一个 list

```
from skimage import data
import matplotlib.pyplot as plt
img=data.camera()
plt.figure("hist")
arr=img.flatten()
n, bins, patches = plt.hist(arr, bins=256,
normed=1, edgecolor='None', facecolor='red')
plt.show()
```



其中的 **flatten()**函数是 **numpy** 包里面的, 用于将二维数组序列化成一维数组。

是按行序列, 如

```
mat=[[1 2 3
      4 5 6]]
```

经过 **mat.flatten()**后, 就变成了

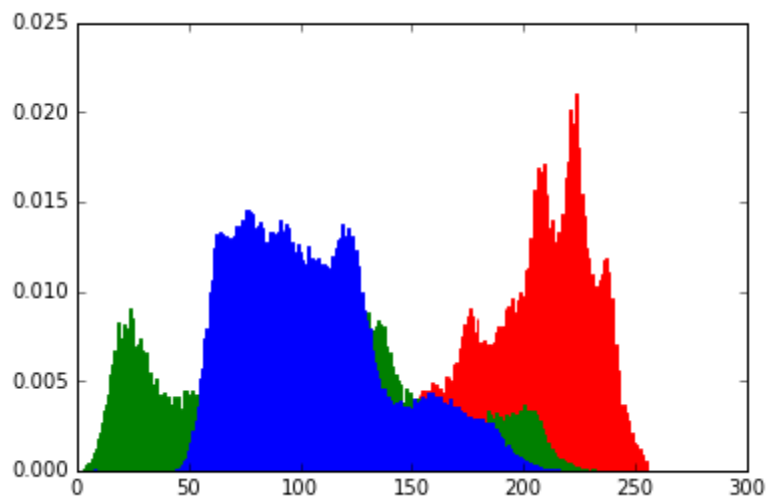
```
mat=[1 2 3 4 5 6]
```

3、彩色图片三通道直方图

一般来说直方图都是针对灰度图的，如果要画 **rgb** 图像的三通道直方图，实际上就是三个直方图的叠加。

```
from skimage import data
import matplotlib.pyplot as plt
img=data.lena()
ar=img[:, :, 0].flatten()
plt.hist(ar, bins=256, normed=1, facecolor='r', edgecolor='r', hold=1)
ag=img[:, :, 1].flatten()
plt.hist(ag, bins=256, normed=1, facecolor='g', edgecolor='g', hold=1)
ab=img[:, :, 2].flatten()
plt.hist(ab, bins=256, normed=1, facecolor='b', edgecolor='b')
plt.show()
```

其中，加一个参数 **hold=1**,表示可以叠加



4、直方图均衡化

如果一副图像的像素占有很多的灰度级而且分布均匀，那么这样的图像往往有高对比度和多变的灰度色调。直方图均衡化就是一种能仅靠输入图像直方图信息自动达到这种效果的变换函数。它的基本思想是对图像中像素个数多的灰度级进行展宽，而对图像中像素个数少的灰度级进行压缩，从而扩展取值的动态范围，提高了对比度和灰度色调的变化，使图像更加清晰。

```
from skimage import data, exposure
import matplotlib.pyplot as plt
```



```

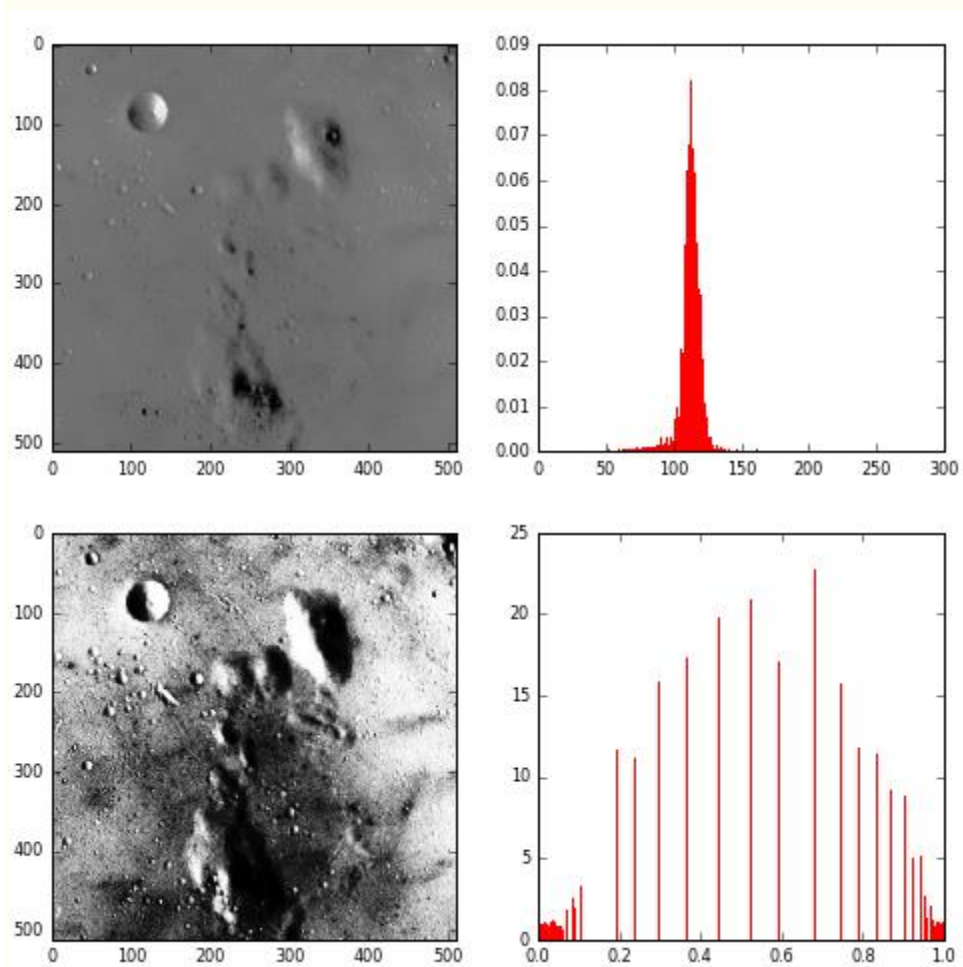
img=data.moon()
plt.figure("hist",figsize=(8,8))

arr=img.flatten()
plt.subplot(221)
plt.imshow(img,plt.cm.gray) #原始图像
plt.subplot(222)
plt.hist(arr, bins=256, normed=1,edgecolor='None',facecolor='red') #
原始图像直方图

img1=exposure.equalize_hist(img)
arr1=img1.flatten()
plt.subplot(223)
plt.imshow(img1,plt.cm.gray) #均衡化图像
plt.subplot(224)
plt.hist(arr1, bins=256, normed=1,edgecolor='None',facecolor='red') #
均衡化直方图

plt.show()

```



十、 图像简单滤波

对图像进行滤波，可以有两种效果：一种是平滑滤波，用来抑制噪声；另一种是微分算子，可以用来检测边缘和特征提取。

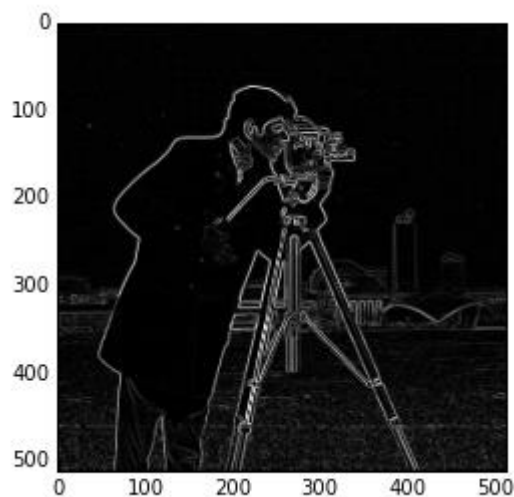
`skimage` 库中通过 `filters` 模块进行滤波操作。

1、sobel 算子

`sobel` 算子可用来检测边缘

函数格式为: `skimage.filters.sobel(image, mask=None)`

```
from skimage import data, filters
import matplotlib.pyplot as plt
img = data.camera()
edges = filters.sobel(img)
plt.imshow(edges, plt.cm.gray)
```



2、roberts 算子

`roberts` 算子和 `sobel` 算子一样，用于检测边缘

调用格式也是一样的：

```
edges = filters.roberts(img)
```

3、scharr 算子

功能同 `sobel`，调用格式：

```
edges = filters.scharr(img)
```

4、prewitt 算子

功能同 **sobel**，调用格式：

```
edges = filters.prewitt(img)
```

5、canny 算子

canny 算子也是用于提取边缘特征，但它不是放在 **filters** 模块，而是放在 **feature** 模块

函数格式：skimage.feature.canny(image, sigma=1.0)

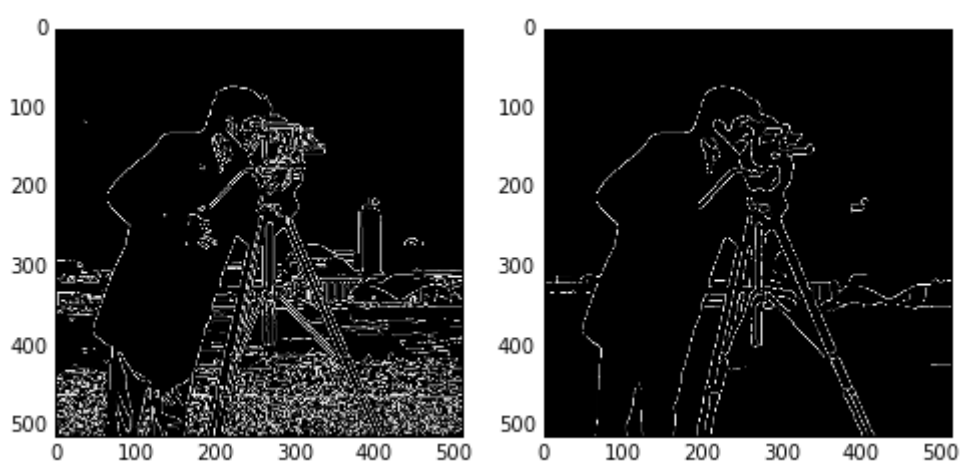
可以修改 **sigma** 的值来调整效果

```
from skimage import data, filters, feature
import matplotlib.pyplot as plt
img = data.camera()
edges1 = feature.canny(img)    #sigma=1
edges2 = feature.canny(img, sigma=3)    #sigma=3

plt.figure('canny', figsize=(8, 8))
plt.subplot(121)
plt.imshow(edges1, plt.cm.gray)

plt.subplot(122)
plt.imshow(edges2, plt.cm.gray)

plt.show()
```



从结果可以看出， σ 越小，边缘线条越细小。

6、gabor 滤波

gabor 滤波可用来进行边缘检测和纹理特征提取。

函数调用格式: `skimage.filters.gabor_filter(image, frequency)`

通过修改 **frequency** 值来调整滤波效果，返回一对边缘结果，一个是用真实滤波核的滤波结果，一个是想象的滤波核的滤波结果。

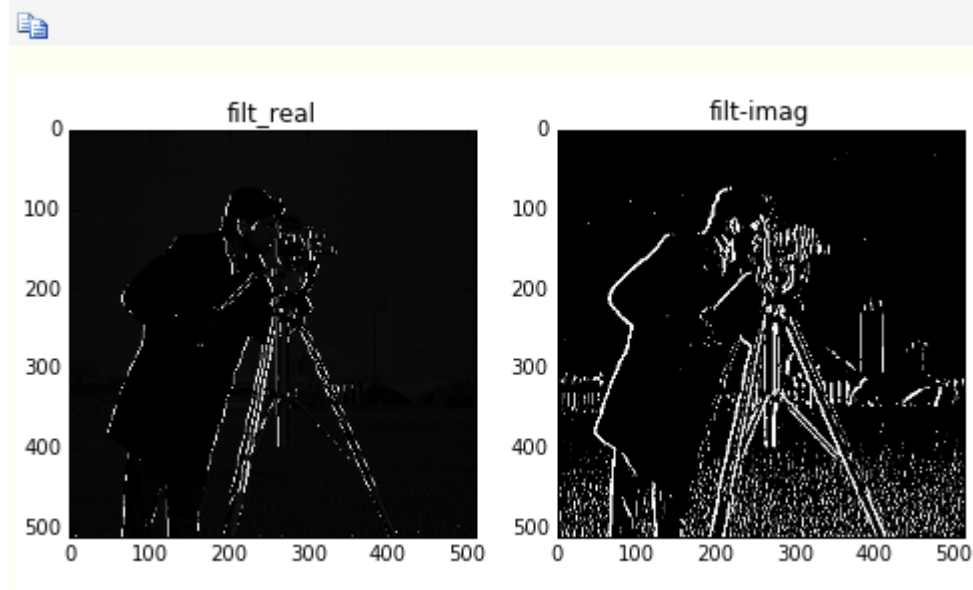
```
from skimage import data, filters
import matplotlib.pyplot as plt
img = data.camera()
filt_real, filt_imag = filters.gabor_filter(img, frequency=0.6)

plt.figure('gabor', figsize=(8, 8))

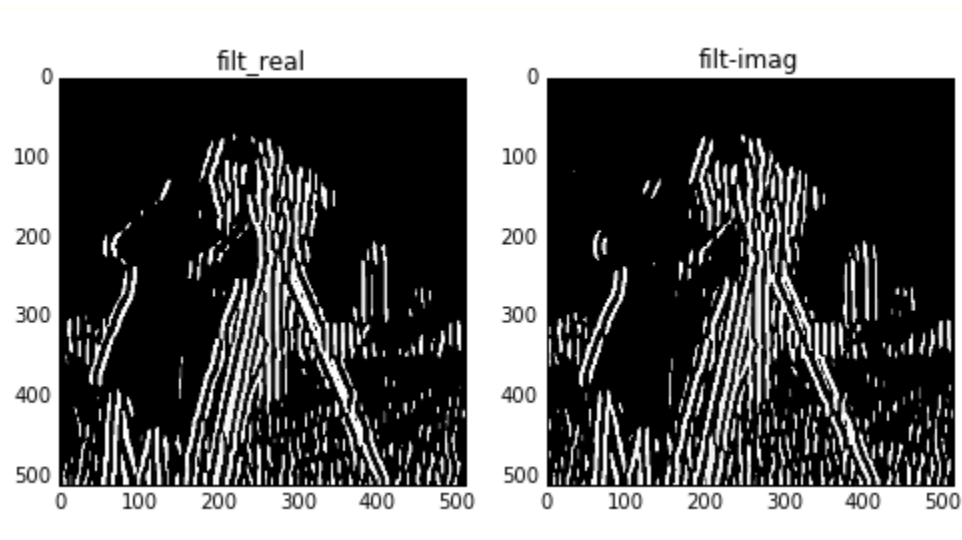
plt.subplot(121)
plt.title('filt_real')
plt.imshow(filt_real, plt.cm.gray)

plt.subplot(122)
plt.title('filt-imag')
plt.imshow(filt_imag, plt.cm.gray)

plt.show()
```



以上为 $\text{frequency}=0.6$ 的结果图。



以上为 $\text{frequency}=0.1$ 的结果图

7、gaussian 滤波

多维的滤波器，是一种平滑滤波，可以消除高斯噪声。

调用函数为: `skimage.filters.gaussian_filter(image, sigma)`

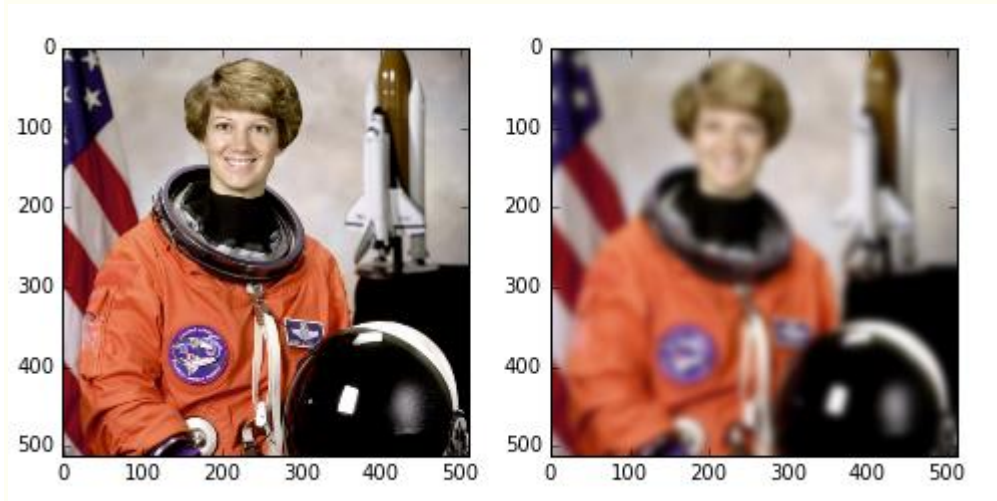
通过调节 **sigma** 的值来调整滤波效果

```
from skimage import data, filters
import matplotlib.pyplot as plt
img = data.astronaut()
edges1 = filters.gaussian_filter(img, sigma=0.4)  #sigma=0.4
edges2 = filters.gaussian_filter(img, sigma=5)   #sigma=5

plt.figure('gaussian', figsize=(8, 8))
plt.subplot(121)
plt.imshow(edges1, plt.cm.gray)

plt.subplot(122)
plt.imshow(edges2, plt.cm.gray)

plt.show()
```



可见 `sigma` 越大，过滤后的图像越模糊

8.median

中值滤波，一种平滑滤波，可以消除噪声。

需要用 `skimage.morphology` 模块来设置滤波器的形状。



```
from skimage import data, filters
import matplotlib.pyplot as plt
from skimage.morphology import disk
img = data.camera()
edges1 = filters.median(img, disk(5))
edges2 = filters.median(img, disk(9))
```

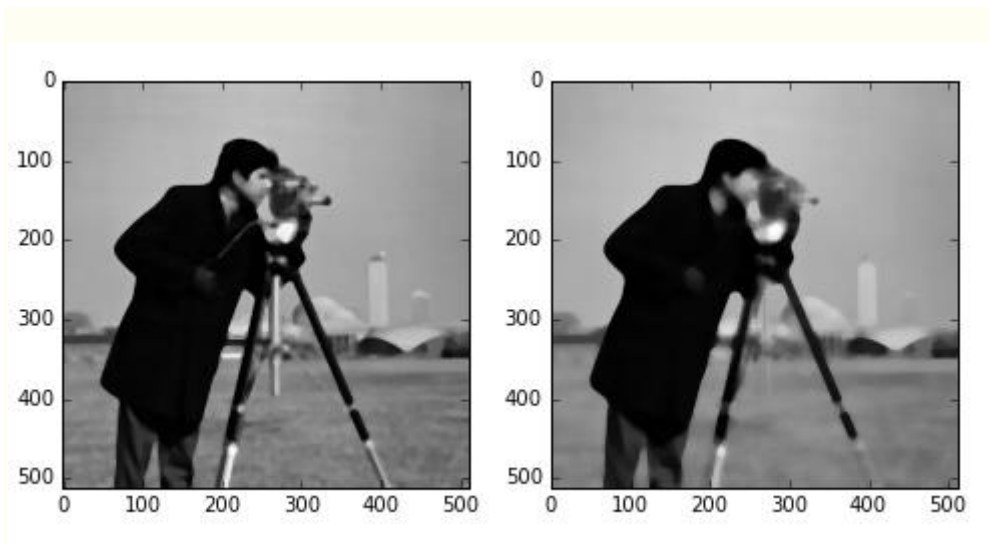
```
plt.figure('median', figsize=(8, 8))
```

```
plt.subplot(121)
plt.imshow(edges1, plt.cm.gray)
```

```
plt.subplot(122)
plt.imshow(edges2, plt.cm.gray)
```

```
plt.show()
```





从结果可以看出，滤波器越大，图像越模糊。

9、水平、垂直边缘检测

上边所举的例子都是进行全部边缘检测，有些时候我们只需要检测水平边缘，或垂直边缘，就可用下面的方法。

水平边缘检测: `sobel_h`, `prewitt_h`, `scharr_h`

垂直边缘检测: `sobel_v`, `prewitt_v`, `scharr_v`

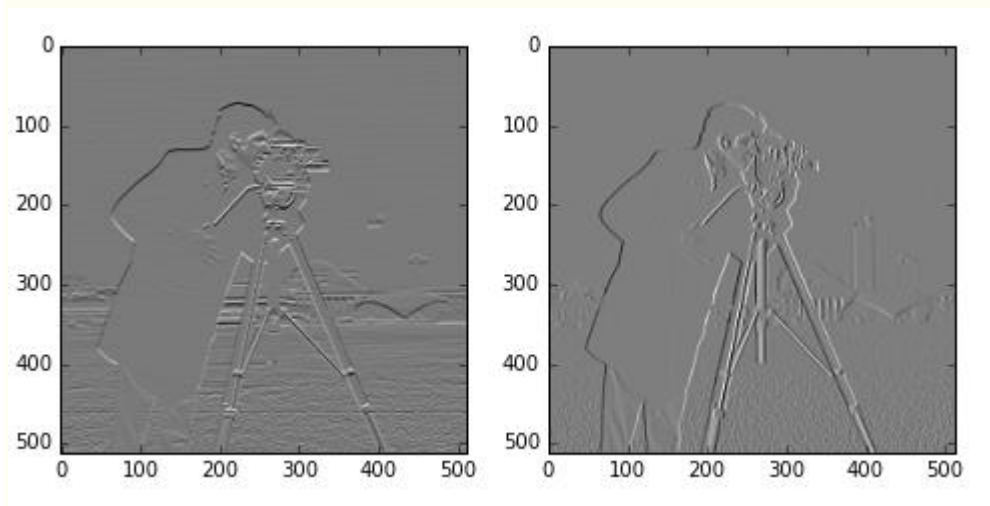
```
from skimage import data, filters
import matplotlib.pyplot as plt
img = data.camera()
edges1 = filters.sobel_h(img)
edges2 = filters.sobel_v(img)

plt.figure('sobel_v_h', figsize=(8, 8))

plt.subplot(121)
plt.imshow(edges1, plt.cm.gray)

plt.subplot(122)
plt.imshow(edges2, plt.cm.gray)

plt.show()
```



上边左图为检测出的水平边缘，右图为检测出的垂直边缘。

10、交叉边缘检测

可使用 **Roberts** 的十字交叉核来进行过滤，以达到检测交叉边缘的目的。这些交叉边缘实际上是梯度在某个方向上的一个分量。

其中一个核：

```
0  1
-1 0
```

对应的函数：

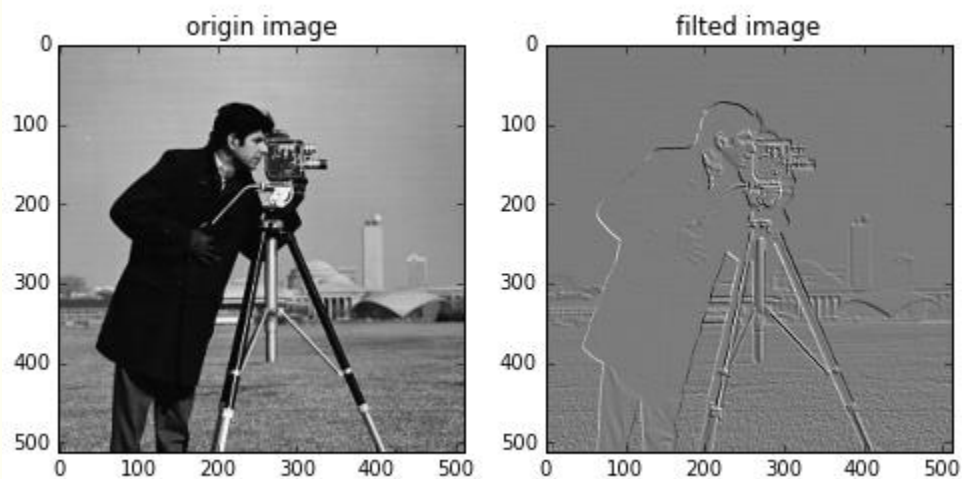
`roberts_neg_diag(image)`

例：

```
from skimage import data, filters
import matplotlib.pyplot as plt
img = data.camera()
dst = filters.roberts_neg_diag(img)

plt.figure('filters', figsize=(8, 8))
plt.subplot(121)
plt.title('origin image')
plt.imshow(img, plt.cm.gray)

plt.subplot(122)
plt.title('filted image')
plt.imshow(dst, plt.cm.gray)
```

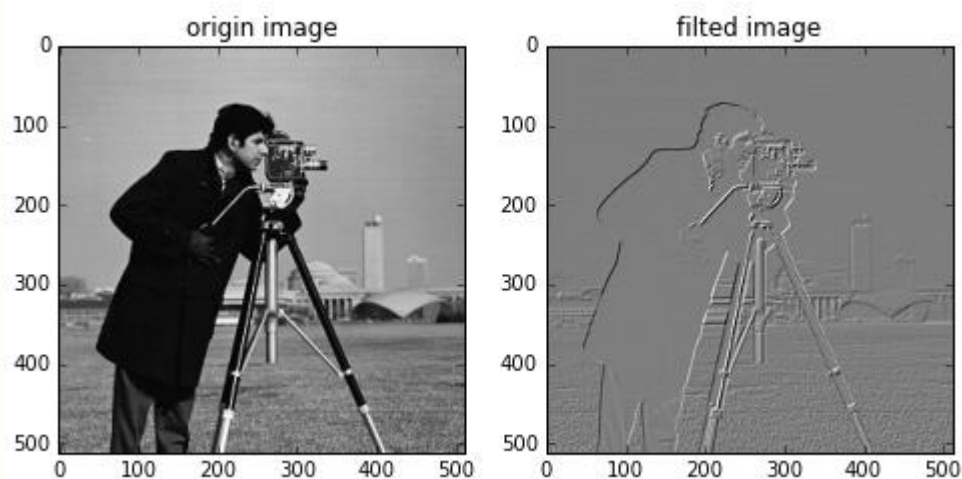



另外一个核:

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

对应函数为:

```
roberts_pos_diag(image)
from skimage import data, filters
import matplotlib.pyplot as plt
img = data.camera()
dst = filters.roberts_pos_diag(img)
plt.figure('filters', figsize=(8,8))
plt.subplot(121)
plt.title('origin image')
plt.imshow(img, plt.cm.gray)
plt.subplot(122)
plt.title('filted image')
plt.imshow(dst, plt.cm.gray)
```



十一、 图像自动阈值分割

图像阈值分割是一种广泛应用的分割技术，利用图像中要提取的目标区域与其背景在灰度特性上的差异，把图像看作具有不同灰度级的两类区域(目标区域和背景区域)的组合，选取一个比较合理的阈值，以确定图像中每个像素点应该属于目标区域还是背景区域，从而产生相应的二值图像。

在 **skimage** 库中，阈值分割的功能是放在 **filters** 模块中。

我们可以手动指定一个阈值，从而来实现分割。也可以让系统自动生成一个阈值，下面几种方法就是用来自动生成阈值。

1、threshold_otsu

基于 **Otsu** 的阈值分割方法，函数调用格式：

```
skimage.filters.threshold_otsu(image, nbins=256)
```

参数 **image** 是指灰度图像，返回一个阈值。

```

from skimage import data, filters
import matplotlib.pyplot as plt
image = data.camera()
thresh = filters.threshold_otsu(image)    #返回一个阈值
dst =(image <= thresh)*1.0    #根据阈值进行分割

plt.figure('thresh', figsize=(8, 8))

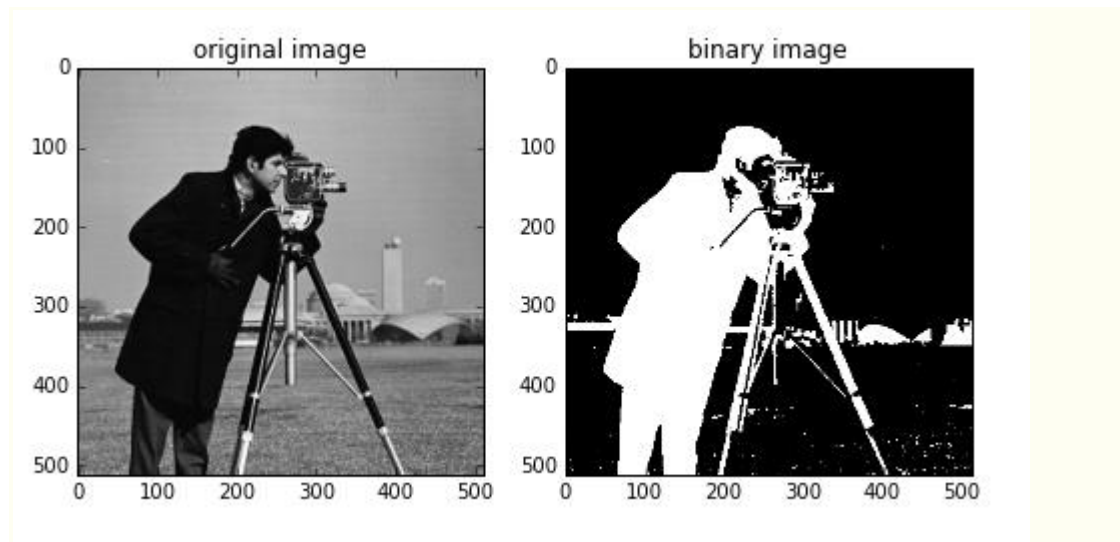
plt.subplot(121)
plt.title('original image')
plt.imshow(image, plt.cm.gray)

plt.subplot(122)
plt.title('binary image')
plt.imshow(dst, plt.cm.gray)

plt.show()
```



返回阈值为 **87**，根据 **87** 进行分割得下图：

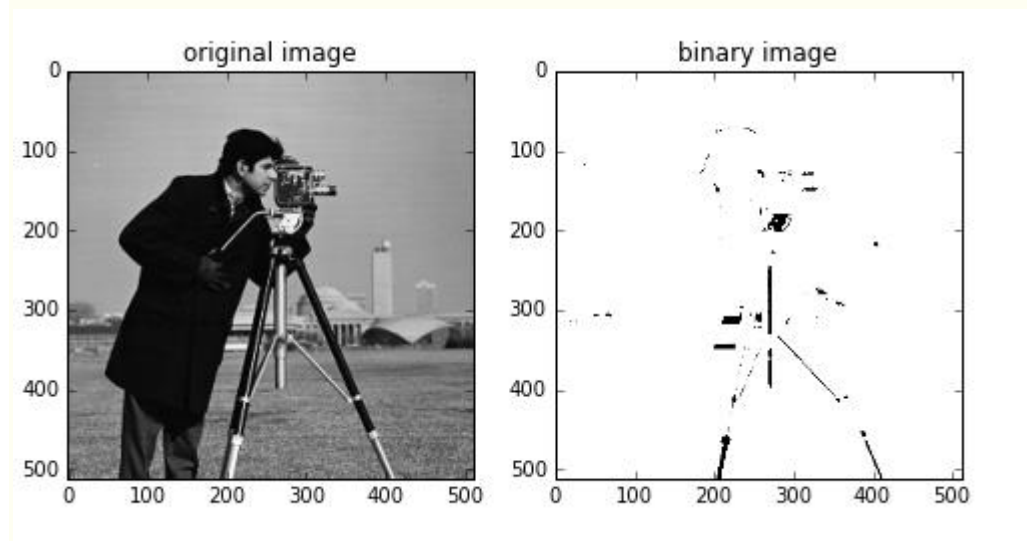


2、threshold_yen

使用方法同上：

```
thresh = filters.threshold_yen(image)
```

返回阈值为 **198**，分割如下图：

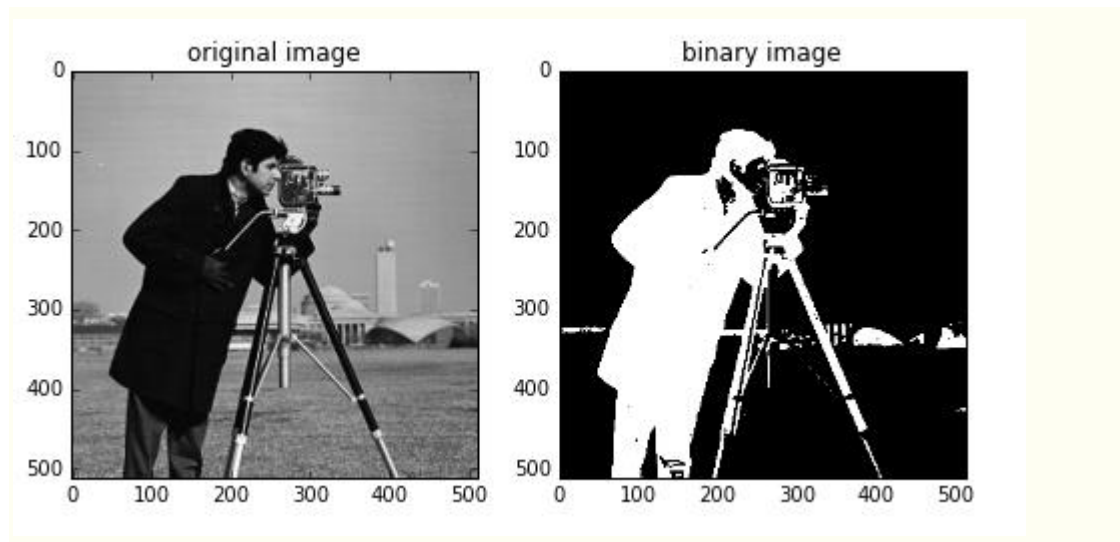


3、threshold_li

使用方法同上：

```
thresh = filters.threshold_li(image)
```

返回阈值 **64.5**，分割如下图：



4、threshold_isodata

阈值计算方法：

$$\text{threshold} = (\text{image}[\text{image} \leq \text{threshold}].\text{mean}() + \text{image}[\text{image} > \text{threshold}].\text{mean}()) / 2.0$$

使用方法同上：

```
thresh = filters.threshold_isodata(image)
```

返回阈值为 87，因此分割效果和 threshold_otsu 一样。

5、threshold_adaptive

调用函数为：

```
skimage.filters.threshold_adaptive(image, block_size, method='gaussian')
```

block_size: 块大小，指当前像素的相邻区域大小，一般是奇数（如 3，5，7。。。）

method: 用来确定自适应阈值的方法，有 'mean', 'generic', 'gaussian' 和 'median'。省略时默认为 gaussian

该函数直接访问一个阈值后的图像，而不是阈值。



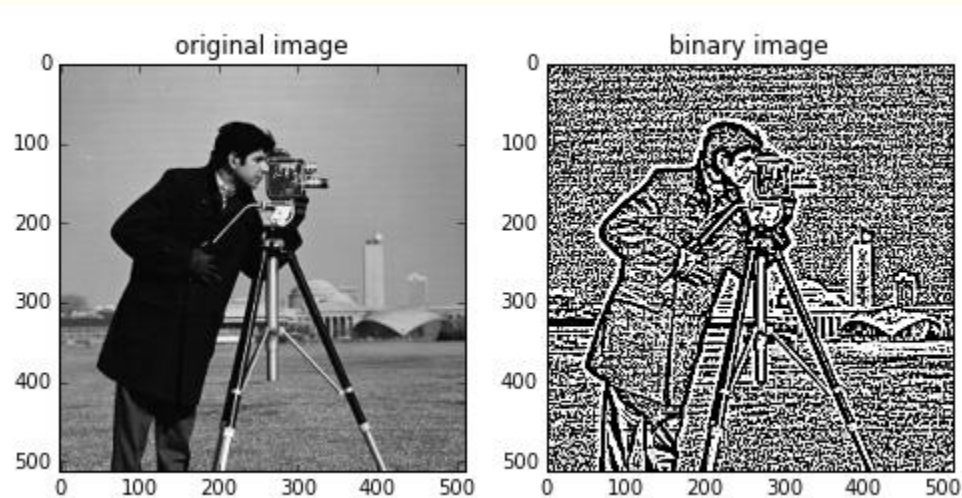
```
from skimage import data, filters
import matplotlib.pyplot as plt
image = data.camera()
dst = filters.threshold_adaptive(image, 15) #返回一个阈值图像

plt.figure('thresh', figsize=(8, 8))
```

```
plt.subplot(121)
plt.title('original image')
plt.imshow(image, plt.cm.gray)
```

```
plt.subplot(122)
plt.title('binary image')
plt.imshow(dst, plt.cm.gray)
```

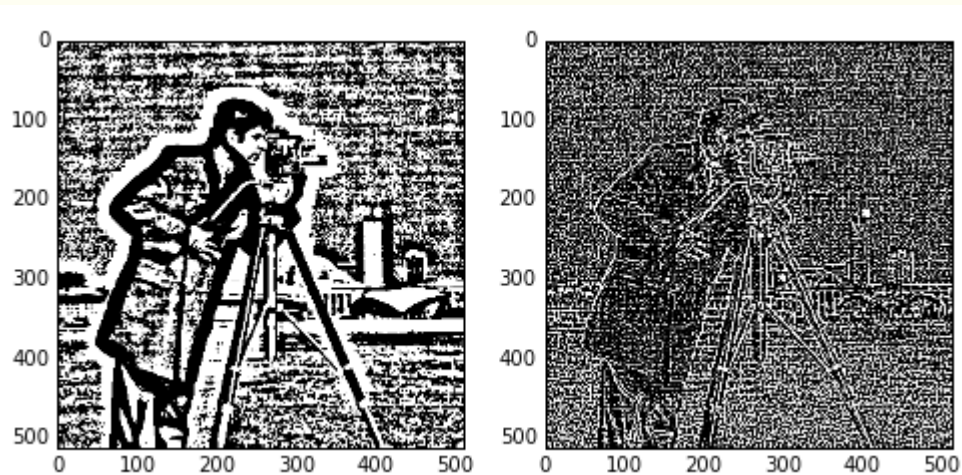
```
plt.show()
```



大家可以修改 `block_size` 的大小和 `method` 值来查看更多的效果。如：

```
dst1 =filters.threshold_adaptive(image,31,'mean')
dst2 =filters.threshold_adaptive(image,5,'median')
```

两种效果如下：



十二、 基本图形的绘制

图形包括线条、圆形、椭圆形、多边形等。

在 `skimage` 包中，绘制图形用的是 `draw` 模块，不要和绘制图像搞混了。

1、画线条

函数调用格式为：

```
skimage.draw.line(r1,c1,r2,c2)
```

r1,r2: 开始点的行数和结束点的行数

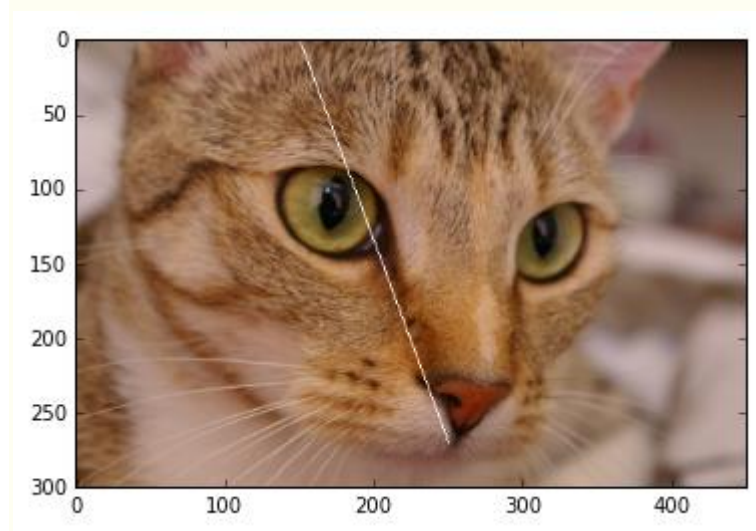
c1,c2: 开始点的列数和结束点的列数

返回当前绘制图形上所有点的坐标，如：

```
rr, cc =draw.line(1, 5, 8, 2)
```

表示从（1，5）到（8，2）连一条线，返回线上所有的像素点坐标[rr,cc]

```
from skimage import draw,data
import matplotlib.pyplot as plt
img=data.chelsea()
rr, cc =draw.line(1, 150, 470, 450)
img[rr, cc] =255
plt.imshow(img,plt.cm.gray)
```



如果想画其它颜色的线条，则可以使用 `set_color()` 函数，格式为：

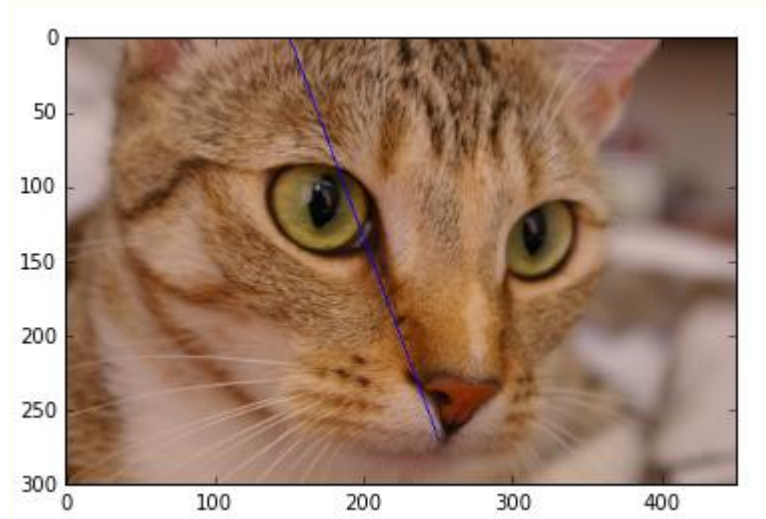
```
skimage.draw.set_color(img, coords, color)
```

例：


```
draw.set_color(img, [rr, cc], [255, 0, 0])
```

则绘制红色线条。

```
from skimage import draw, data
import matplotlib.pyplot as plt
img=data.chelsea()
rr, cc =draw.line(1, 150, 270, 250)
draw.set_color(img, [rr, cc], [0, 0, 255])
plt.imshow(img, plt.cm.gray)
```

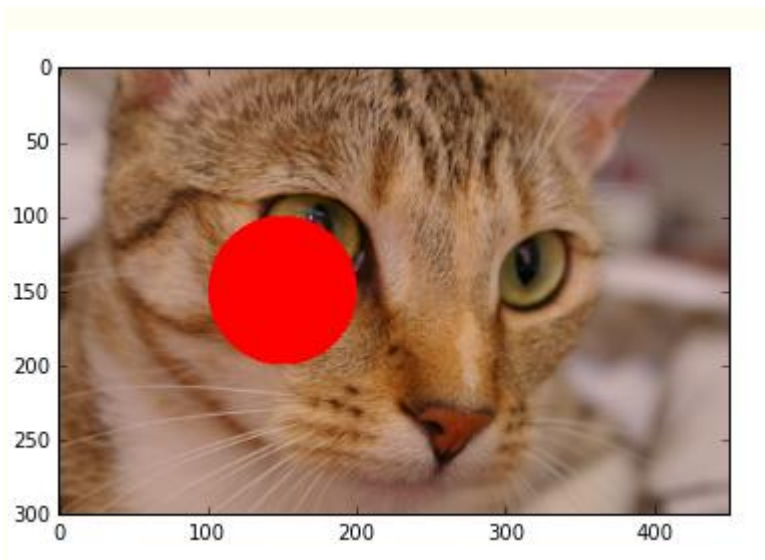


2、画圆

函数格式: `skimage.draw.circle(cy, cx, radius)`

cy 和 *cx* 表示圆心点, *radius* 表示半径

```
from skimage import draw, data
import matplotlib.pyplot as plt
img=data.chelsea()
rr, cc=draw.circle(150,150,50)
draw.set_color(img, [rr, cc], [255, 0, 0])
plt.imshow(img, plt.cm.gray)
```



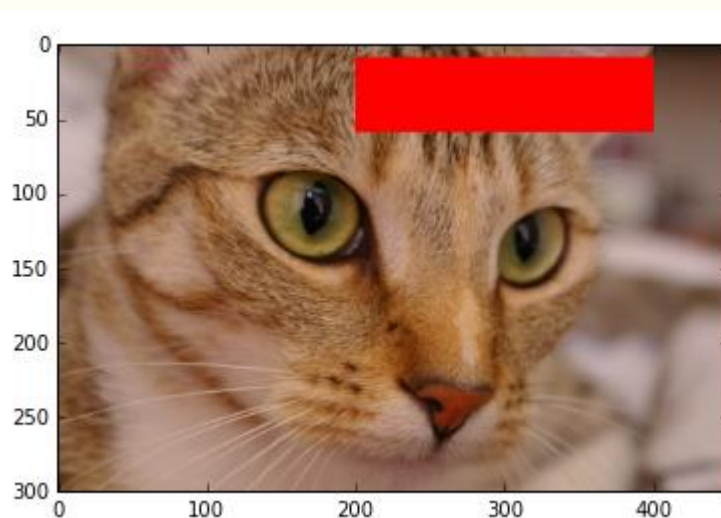
3、多边形

函数格式: `skimage.draw.polygon(Y,X)`

Y 为多边形顶点的行集合, X 为各顶点的列值集合。



```
from skimage import draw, data
import matplotlib.pyplot as plt
import numpy as np
img=data.chelsea()
Y=np.array([10,10,60,60])
X=np.array([200,400,400,200])
rr, cc=draw.polygon(Y,X)
draw.set_color(img, [rr,cc], [255,0,0])
plt.imshow(img,plt.cm.gray)
```



我在此处只设置了四个顶点，因此是个四边形。

4、椭圆

格式: `skimage.draw.ellipse(cy, cx, yradius, xradius)`

cy 和 *cx* 为中心点坐标, *yradius* 和 *xradius* 代表长短轴。

```
from skimage import draw, data
import matplotlib.pyplot as plt
img=data.chelsea()
rr, cc=draw.ellipse(150, 150, 30, 80)
draw.set_color(img, [rr,cc], [255,0,0])
plt.imshow(img,plt.cm.gray)
```



5、贝塞尔曲线

格式: `skimage.draw.bezier_curve(y1,x1,y2,x2,y3,x3,weight)`

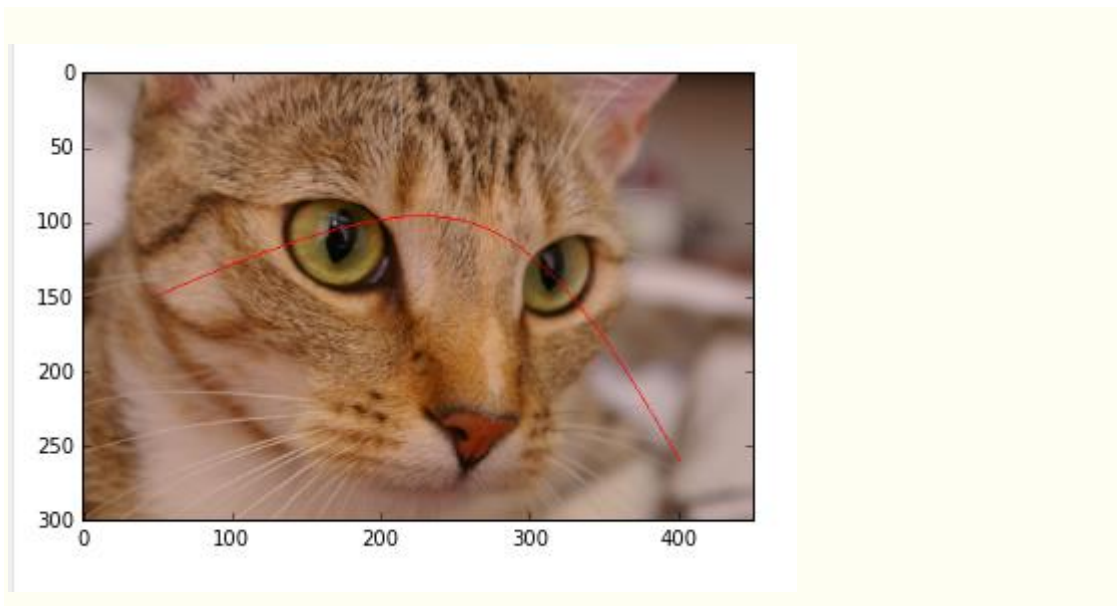
y1,x1 表示第一个控制点坐标

y2,x2 表示第二个控制点坐标

y3,x3 表示第三个控制点坐标

weight 表示中间控制点的权重, 用于控制曲线的弯曲度。

```
from skimage import draw, data
import matplotlib.pyplot as plt
img=data.chelsea()
rr, cc=draw.bezier_curve(150, 50, 50, 280, 260, 400, 2)
draw.set_color(img, [rr,cc], [255,0,0])
plt.imshow(img,plt.cm.gray)
```



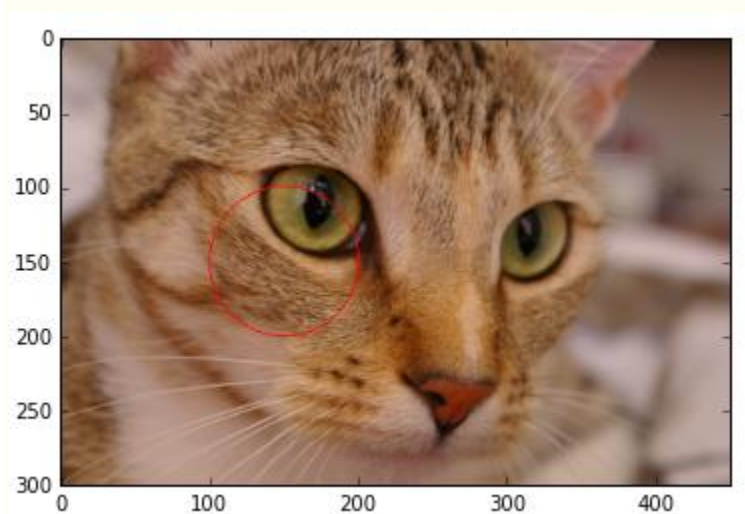
6、画空心圆

和前面的画圆是一样的，只是前面是实心圆，而此处画空心圆，只有边框线。

格式: `skimage.draw.circle_perimeter(yx,yc,radius)`

`yx,yc` 是圆心坐标，`radius` 是半径

```
from skimage import draw,data
import matplotlib.pyplot as plt
img=data.chelsea()
rr, cc=draw.circle_perimeter(150,150,50)
draw.set_color(img, [rr,cc], [255,0,0])
plt.imshow(img,plt.cm.gray)
```



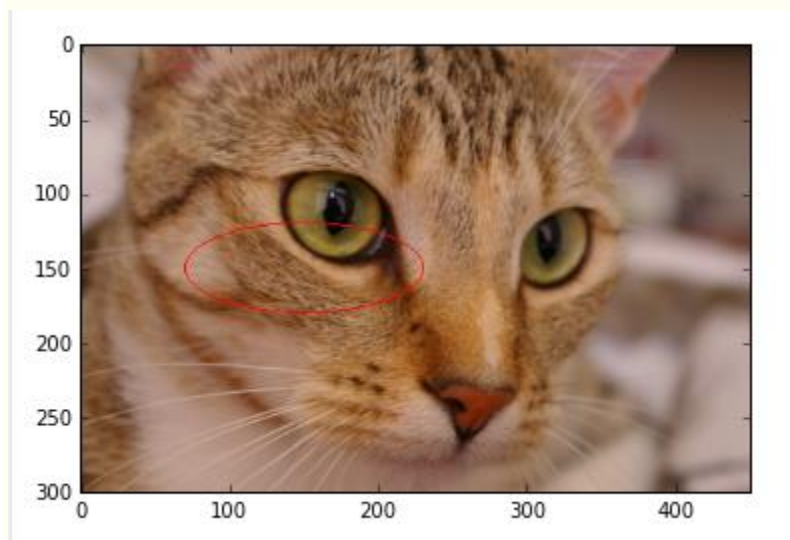
7、空心椭圆

格式: `skimage.draw.ellipse_perimeter(cy, cx, yradius, xradius)`

cy,cx 表示圆心

yradius,xradius 表示长短轴

```
from skimage import draw, data
import matplotlib.pyplot as plt
img=data.chelsea()
rr, cc=draw.ellipse_perimeter(150, 150, 30, 80)
draw.set_color(img, [rr,cc], [255,0,0])
plt.imshow(img,plt.cm.gray)
```



十三、 基本形态学滤波

对图像进行形态学变换。变换对象一般为灰度图或二值图，功能函数放在 `morphology` 子模块内。

1、膨胀 (dilation)

原理：一般对二值图像进行操作。找到像素值为 1 的点，将它的邻近像素点都设置成这个值。1 值表示白，0 值表示黑，因此膨胀操作可以扩大白色值范围，压缩黑色值范围。一般用来扩充边缘或填充小的孔洞。

功能函数：`skimage.morphology.dilation(image, selem=None)`

`selem` 表示结构元素，用于设定局部区域的形状和大小。

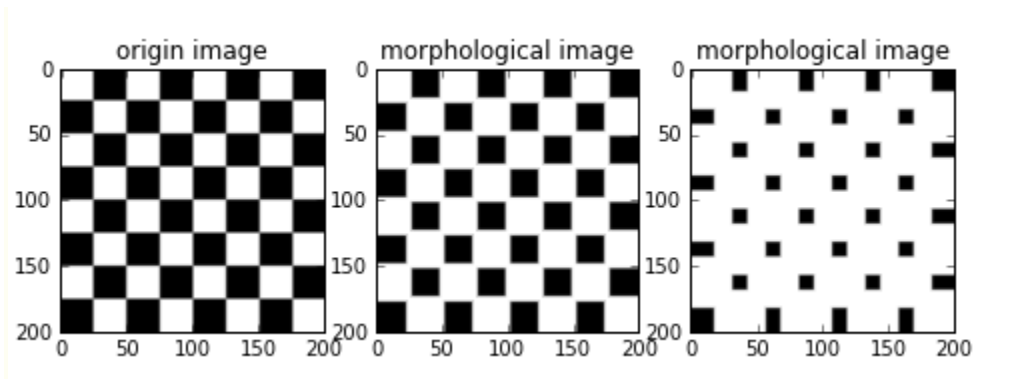
```
from skimage import data
import skimage.morphology as sm
import matplotlib.pyplot as plt
img=data.checkerboard()
dst1=sm.dilation(img, sm.square(5))  #用边长为 5 的正方形滤波器进行膨胀滤波
dst2=sm.dilation(img, sm.square(15)) #用边长为 15 的正方形滤波器进行膨胀滤波

plt.figure('morphology',figsize=(8,8))
plt.subplot(131)
plt.title('origin image')
plt.imshow(img,plt.cm.gray)

plt.subplot(132)
plt.title('morphological image')
plt.imshow(dst1,plt.cm.gray)

plt.subplot(133)
plt.title('morphological image')
plt.imshow(dst2,plt.cm.gray)
```

分别用边长为 5 或 15 的正方形滤波器对棋盘图片进行膨胀操作，结果如下：



可见滤波器的大小，对操作结果的影响非常大。一般设置为奇数。

除了正方形的滤波器外，滤波器的形状还有一些，现列举如下：

`morphology.square`: 正方形

`morphology.disk`: 平面圆形

`morphology.ball`: 球形

`morphology.cube`: 立方体形

`morphology.diamond`: 钻石形

`morphology.rectangle`: 矩形

`morphology.star`: 星形

`morphology.octagon`: 八角形

`morphology.octahedron`: 八面体

注意，如果处理图像为二值图像（只有 0 和 1 两个值），则可以调用：

`skimage.morphology.binary_dilation(image, selem=None)`

用此函数比处理灰度图像要快。

2、腐蚀 (erosion)

函数: `skimage.morphology.erosion(image, selem=None)`

`selem` 表示结构元素，用于设定局部区域的形状和大小。

和膨胀相反的操作，将 0 值扩充到邻近像素。扩大黑色部分，减小白色部分。可用来提取骨干信息，去掉毛刺，去掉孤立的像素。

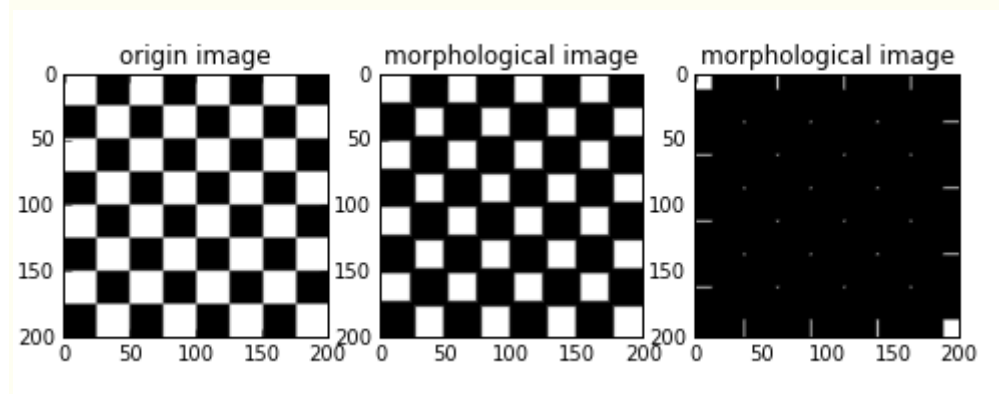
```
from skimage import data
import skimage.morphology as sm
import matplotlib.pyplot as plt
img=data.checkerboard()
```

```
dst1=sm.erosion(img, sm.square(5)) #用边长为 5 的正方形滤波器进行膨胀滤波
dst2=sm.erosion(img, sm.square(25)) #用边长为 25 的正方形滤波器进行膨胀滤波
```

```
plt.figure('morphology', figsize=(8, 8))
plt.subplot(131)
plt.title('origin image')
plt.imshow(img, plt.cm.gray)
```

```
plt.subplot(132)
plt.title('morphological image')
plt.imshow(dst1, plt.cm.gray)
```

```
plt.subplot(133)
plt.title('morphological image')
plt.imshow(dst2, plt.cm.gray)
```



注意，如果处理图像为二值图像（只有 0 和 1 两个值），则可以调用：

```
skimage.morphology.binary_erosion(image, selem=None)
```

用此函数比处理灰度图像要快。

3、开运算 (opening)

函数: `skimage.morphology.opening(image, selem=None)`

`selem` 表示结构元素，用于设定局部区域的形状和大小。

先腐蚀再膨胀，可以消除小物体或小斑块。

```
from skimage import io,color
import skimage.morphology as sm
import matplotlib.pyplot as plt
img=color.rgb2gray(io.imread('d:/pic/mor.png'))
```

```
dst=sm.opening(img, sm.disk(9)) #用边长为 9 的圆形滤波器进行膨胀滤波

plt.figure('morphology',figsize=(8,8))
plt.subplot(121)
plt.title('origin image')
plt.imshow(img,plt.cm.gray)
plt.axis('off')

plt.subplot(122)
plt.title('morphological image')
plt.imshow(dst,plt.cm.gray)
plt.axis('off')
```



注意，如果处理图像为二值图像（只有 0 和 1 两个值），则可以调用：

```
skimage.morphology.binary_opening(image, selem=None)
```

用此函数比处理灰度图像要快。

4、闭运算（closing）

函数: `skimage.morphology.closing(image, selem=None)`

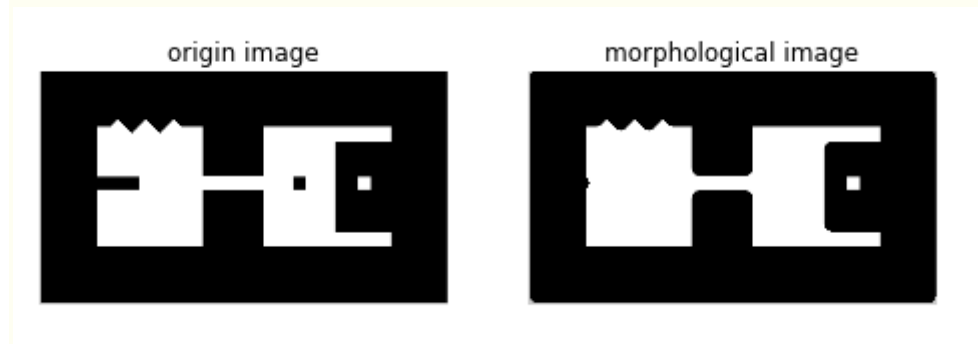
selem 表示结构元素，用于设定局部区域的形状和大小。

先膨胀再腐蚀，可用来填充孔洞。

```
from skimage import io,color
import skimage.morphology as sm
import matplotlib.pyplot as plt
img=color.rgb2gray(io.imread('d:/pic/mor.png'))
dst=sm.closing(img, sm.disk(9)) #用边长为 5 的圆形滤波器进行膨胀滤波

plt.figure('morphology',figsize=(8,8))
plt.subplot(121)
plt.title('origin image')
plt.imshow(img,plt.cm.gray)
plt.axis('off')
```

```
plt.subplot(122)
plt.title('morphological image')
plt.imshow(dst, plt.cm.gray)
plt.axis('off')
```



注意，如果处理图像为二值图像（只有 0 和 1 两个值），则可以调用：

```
skimage.morphology.binary_closing(image, selem=None)
```

用此函数比处理灰度图像要快。

5、白帽 (white-tophat)

函数: `skimage.morphology.white_tophat(image, selem=None)`

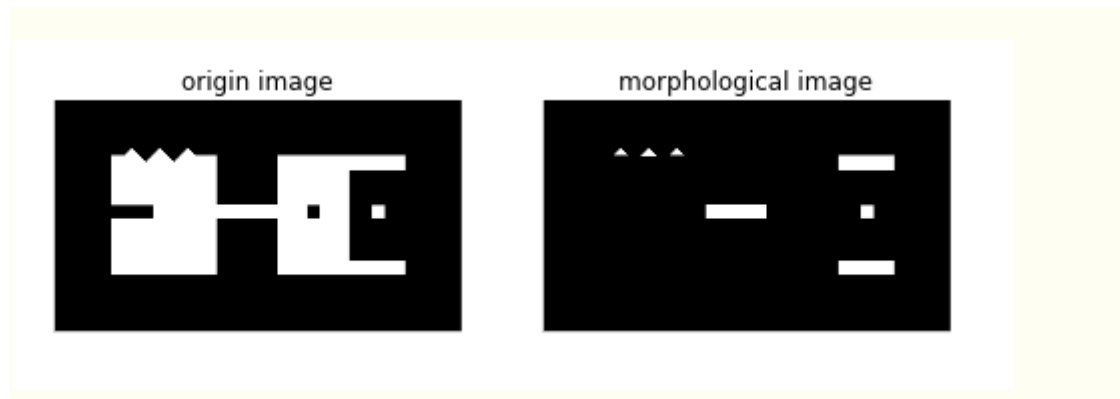
selem 表示结构元素，用于设定局部区域的形状和大小。

将原图像减去它的开运算值，返回比结构化元素小的白点

```
from skimage import io,color
import skimage.morphology as sm
import matplotlib.pyplot as plt
img=color.rgb2gray(io.imread('d:/pic/mor.png'))
dst=sm.white_tophat(img, sm.square(21))
```

```
plt.figure('morphology',figsize=(8,8))
plt.subplot(121)
plt.title('origin image')
plt.imshow(img, plt.cm.gray)
plt.axis('off')
```

```
plt.subplot(122)
plt.title('morphological image')
plt.imshow(dst, plt.cm.gray)
plt.axis('off')
```

6、黑帽 (black-tophat)

函数: `skimage.morphology.black_tophat(image, selem=None)`

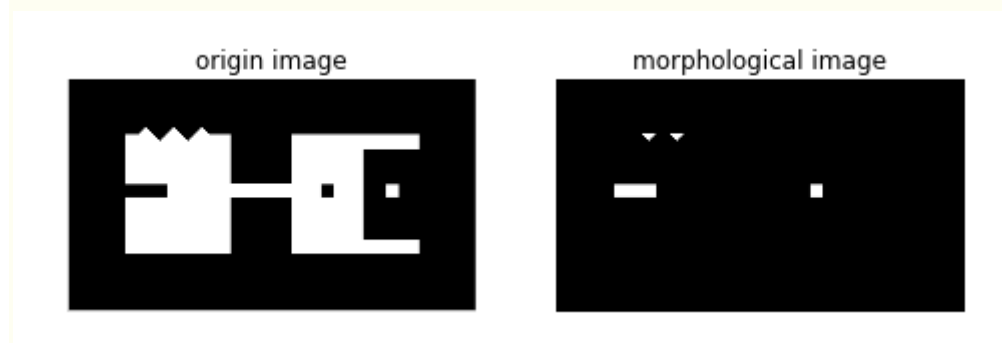
selem 表示结构元素，用于设定局部区域的形状和大小。

将原图像减去它的闭运算值，返回比结构化元素小的黑点，且将这些黑点反色。

```
from skimage import io,color
import skimage.morphology as sm
import matplotlib.pyplot as plt
img=color.rgb2gray(io.imread('d:/pic/mor.png'))
dst=sm.black_tophat(img, sm.square(21))

plt.figure('morphology',figsize=(8,8))
plt.subplot(121)
plt.title('origin image')
plt.imshow(img,plt.cm.gray)
plt.axis('off')

plt.subplot(122)
plt.title('morphological image')
plt.imshow(dst,plt.cm.gray)
plt.axis('off')
```



十四、 高级滤波

本文提供更多更强大的滤波方法，这些方法放在 `filters.rank` 子模块内。

这些方法需要用户自己设定滤波器的形状和大小，因此需要导入 `morphology` 模块来设定。

1、autolevel

这个词在 `photoshop` 里面翻译成自动色阶，用局部直方图来对图片进行滤波分级。

该滤波器局部地拉伸灰度像素值的直方图，以覆盖整个像素值范围。

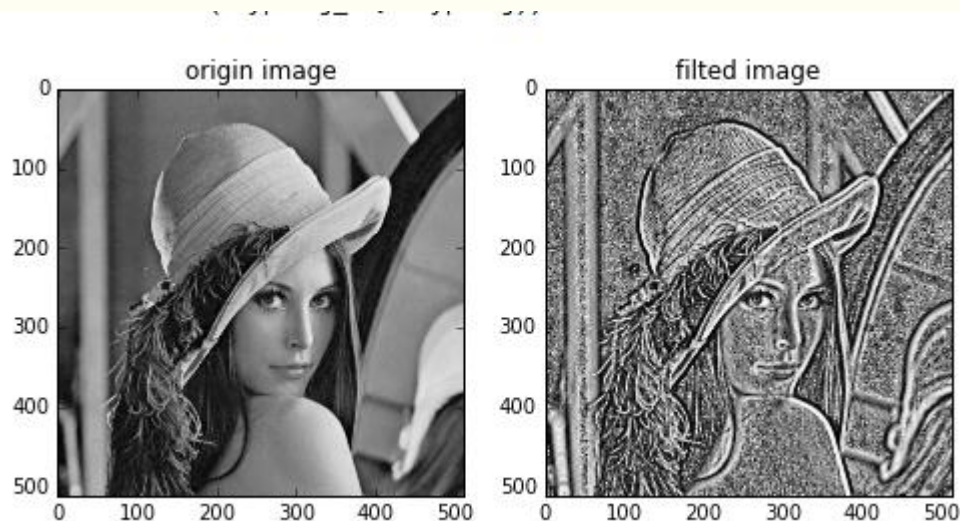
格式: `skimage.filters.rank.autolevel(image, selem)`

`selem` 表示结构化元素，用于设定滤波器。

```
from skimage import data,color
import matplotlib.pyplot as plt
from skimage.morphology import disk
import skimage.filters.rank as sfr
img =color.rgb2gray(data.lena())
auto =sfr.autolevel(img, disk(5))  #半径为 5 的圆形滤波器
```

```
plt.figure('filters',figsize=(8,8))
plt.subplot(121)
plt.title('origin image')
plt.imshow(img,plt.cm.gray)
```

```
plt.subplot(122)
plt.title('filted image')
plt.imshow(auto,plt.cm.gray)
```



2、bottomhat 与 tophat

bottomhat: 此滤波器先计算图像的形态学闭运算，然后用原图像减去运算的结果值，有点像黑帽操作。

bottomhat: 此滤波器先计算图像的形态学开运算，然后用原图像减去运算的结果值，有点像白帽操作。

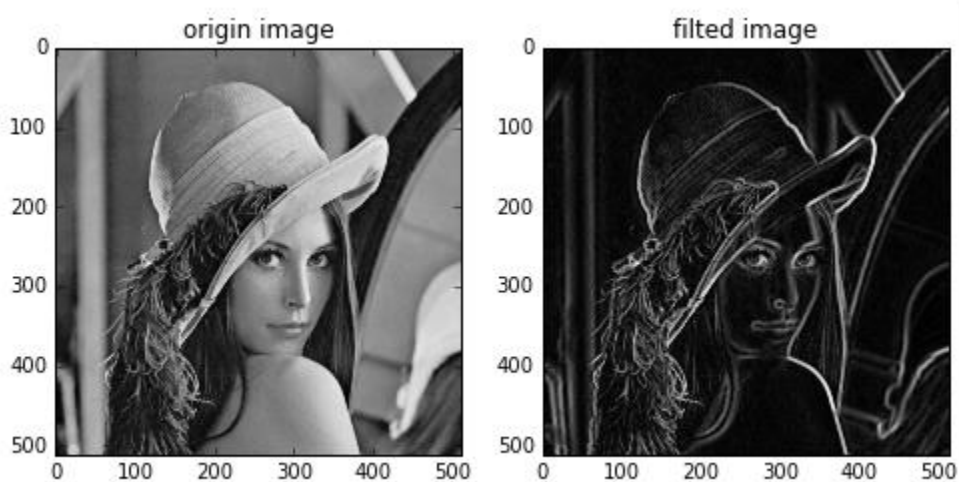
格式：

```
skimage.filters.rank.bottomhat(image, selem)  
skimage.filters.rank.tophat(image, selem)
```

selem 表示结构化元素，用于设定滤波器。

下面是 **bottomhat** 滤波的例子：

```
from skimage import data,color  
import matplotlib.pyplot as plt  
from skimage.morphology import disk  
import skimage.filters.rank as sfr  
img =color.rgb2gray(data.lena())  
auto =sfr.bottomhat(img, disk(5))  #半径为 5 的圆形滤波器  
  
plt.figure('filters',figsize=(8,8))  
plt.subplot(121)  
plt.title('origin image')  
plt.imshow(img,plt.cm.gray)  
  
plt.subplot(122)  
plt.title('filted image')  
plt.imshow(auto,plt.cm.gray)
```



3、 enhance_contrast

对比度增强。求出局部区域的最大值和最小值，然后看当前点像素值最接近最大值还是最小值，然后替换为最大值或最小值。

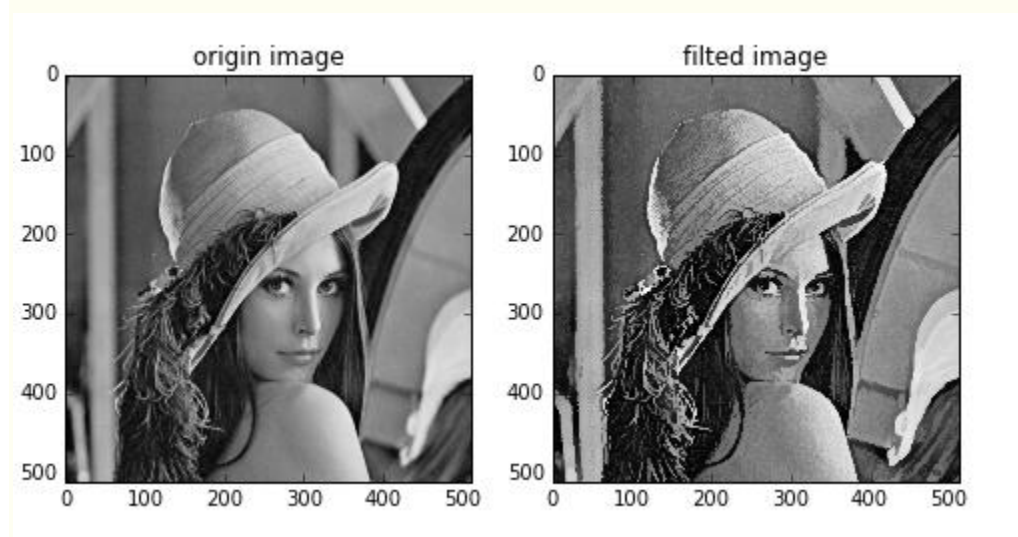
函数： `enhance_contrast(image, selem)`

`selem` 表示结构化元素，用于设定滤波器。

```
from skimage import data,color
import matplotlib.pyplot as plt
from skimage.morphology import disk
import skimage.filters.rank as sfr
img =color.rgb2gray(data.lena())
auto =sfr.enhance_contrast(img, disk(5)) #半径为 5 的圆形滤波器

plt.figure('filters',figsize=(8,8))
plt.subplot(121)
plt.title('origin image')
plt.imshow(img,plt.cm.gray)

plt.subplot(122)
plt.title('filted image')
plt.imshow(auto,plt.cm.gray)
```



4、 entropy

求局部熵，熵是使用基为 2 的对数运算出来的。该函数将局部区域的灰度值分布进行二进制编码，返回编码的最小值。

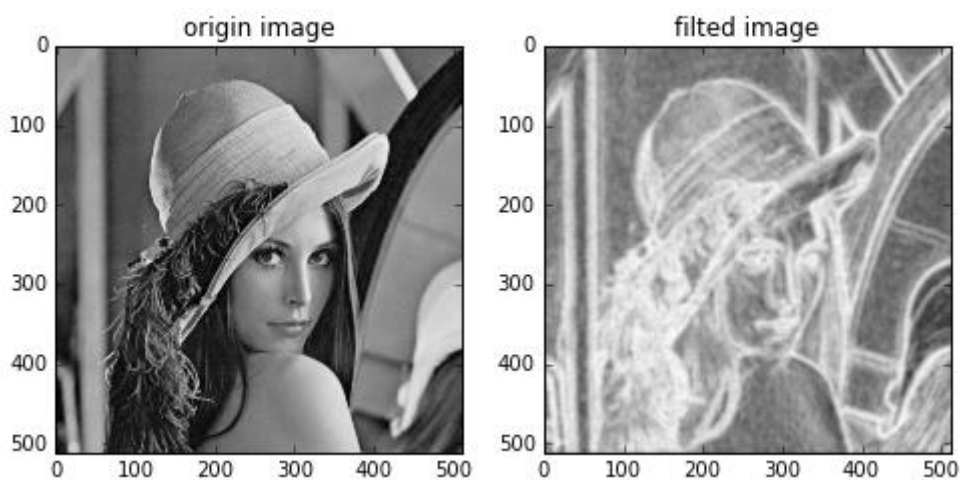
函数格式: `entropy(image, selem)`

`selem` 表示结构化元素, 用于设定滤波器。

```
from skimage import data, color
import matplotlib.pyplot as plt
from skimage.morphology import disk
import skimage.filters.rank as sfr
img = color.rgb2gray(data.lena())
dst = sfr.entropy(img, disk(5)) #半径为 5 的圆形滤波器

plt.figure('filters', figsize=(8, 8))
plt.subplot(121)
plt.title('origin image')
plt.imshow(img, plt.cm.gray)

plt.subplot(122)
plt.title('filted image')
plt.imshow(dst, plt.cm.gray)
```



5、equalize

均衡化滤波。利用局部直方图对图像进行均衡化滤波。

函数格式: `equalize(image, selem)`

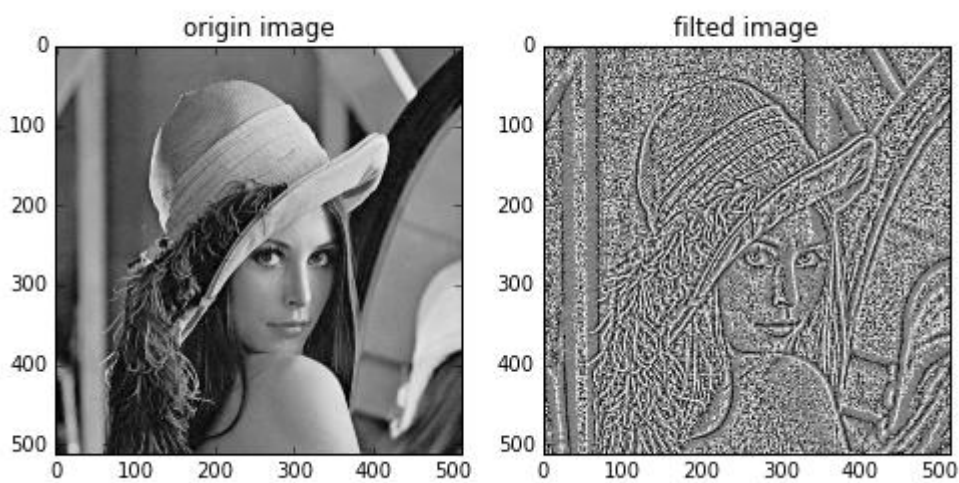
`selem` 表示结构化元素, 用于设定滤波器。

```
from skimage import data, color
import matplotlib.pyplot as plt
from skimage.morphology import disk
import skimage.filters.rank as sfr
```

```
img =color.rgb2gray(data.lena())
dst =sfr.equalize(img, disk(5)) #半径为 5 的圆形滤波器
```

```
plt.figure('filters',figsize=(8,8))
plt.subplot(121)
plt.title('origin image')
plt.imshow(img,plt.cm.gray)
```

```
plt.subplot(122)
plt.title('filted image')
plt.imshow(dst,plt.cm.gray)
```



6、gradient

返回图像的局部梯度值（如：最大值-最小值），用此梯度值代替区域内所有像素值。

函数格式：gradient(image, selem)

selem 表示结构化元素，用于设定滤波器。

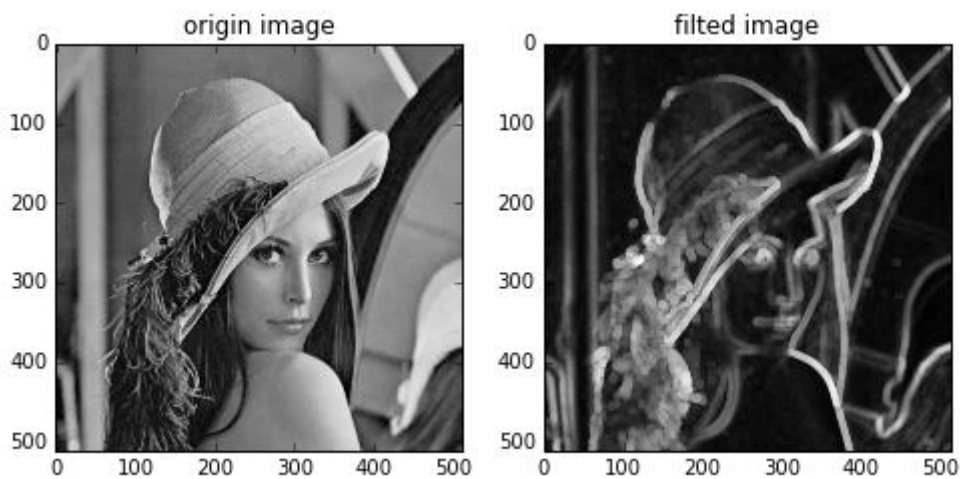


```
from skimage import data,color
import matplotlib.pyplot as plt
from skimage.morphology import disk
import skimage.filters.rank as sfr
img =color.rgb2gray(data.lena())
dst =sfr.gradient(img, disk(5)) #半径为 5 的圆形滤波器
```

```
plt.figure('filters',figsize=(8,8))
plt.subplot(121)
plt.title('origin image')
plt.imshow(img,plt.cm.gray)
```



```
plt.subplot(122)
plt.title('filted image')
plt.imshow(dst, plt.cm.gray)
```



7、其它滤波器

滤波方式很多，下面不再一一详细讲解，仅给出核心代码，所有的函数调用方式都是一样的。

最大值滤波器 (maximum): 返回图像局部区域的最大值，用此最大值代替该区域内所有像素值。

```
dst =sfr.maximum(img, disk(5))
```

最小值滤波器 (minimum): 返回图像局部区域内的最小值，用此最小值取代该区域内所有像素值。

```
dst =sfr.minimum(img, disk(5))
```

均值滤波器 (mean): 返回图像局部区域内的均值，用此均值取代该区域内所有像素值。

```
dst =sfr.mean(img, disk(5))
```

中值滤波器 (median): 返回图像局部区域内的中值，用此中值取代该区域内所有像素值。

```
dst =sfr.median(img, disk(5))
```

莫代尔滤波器 (modal): 返回图像局部区域内的 **modal** 值，用此值取代该区域内所有像素值。

```
dst =sfr.modal(img, disk(5))
```

otsu 阈值滤波 (otsu): 返回图像局部区域内的 **otsu** 阈值，用此值取代该区域内所有像素值。

```
dst =sfr.otsu(img, disk(5))
```

阈值滤波 (**threshold**): 将图像局部区域中的每个像素值与均值比较, 大于则赋值为 **1**, 小于赋值为 **0**, 得到一个二值图像。

```
dst =sfr.threshold(img, disk(5))
```

减均值滤波 (**subtract_mean**): 将局部区域中的每一个像素, 减去该区域中的均值。

```
dst =sfr.subtract_mean(img, disk(5))
```

求和滤波 (**sum**) :求局部区域的像素总和, 用此值取代该区域内所有像素值。

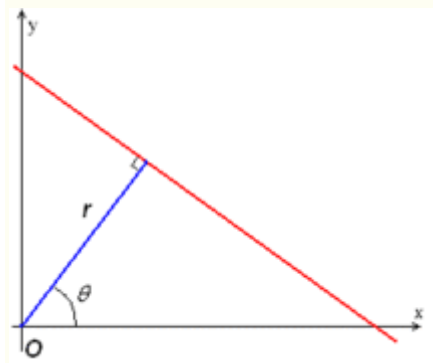
```
dst =sfr.sum(img, disk(5))
```


十五、霍夫线变换

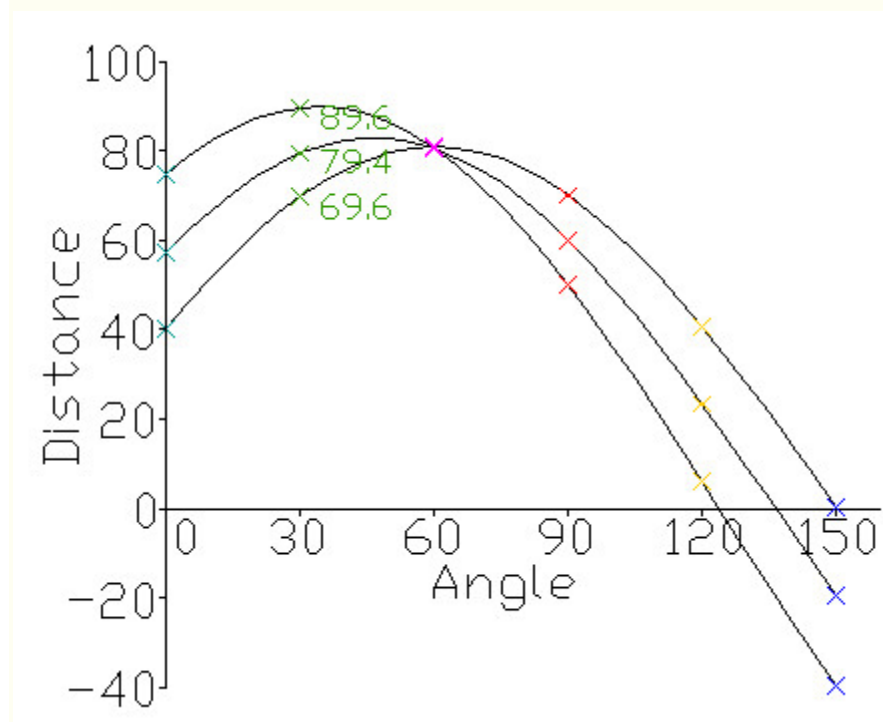
在图片处理中，霍夫变换主要是用来检测图片中的几何形状，包括直线、圆、椭圆等。

在 `skimage` 中，霍夫变换是放在 `tranform` 模块内，本篇主要讲解霍夫线变换。

对于平面中的一条直线，在笛卡尔坐标系中，可用 $y=mx+b$ 来表示，其中 m 为斜率， b 为截距。但是如果直线是一条垂直线，则 m 为无穷大，所有通常我们在另一坐标系中表示直线，即极坐标系下的 $r=x\cos(\theta)+y\sin(\theta)$ 。即可用 (r,θ) 来表示一条直线。其中 r 为该直线到原点的距离， θ 为该直线的垂线与 x 轴的夹角。如下图所示。



对于一个给定的点 (x_0,y_0) ，我们在极坐标下绘出所有通过它的直线 (r,θ) ，将得到一条正弦曲线。如果将图片中的所有非 0 点的正弦曲线都绘制出来，则会存在一些交点。所有经过这个交点的正弦曲线，说明都拥有同样的 (r,θ) ，意味着这些点在一条直线上。



发上图所示，三个点(对应图中的三条正弦曲线)在一条直线上，因为这三个曲线交于一点，具有相同的 (r,θ) 。霍夫线变换就是利用这种方法来寻找图中的直线。

函数: `skimage.transform.hough_line(img)`

返回三个值:

h: 霍夫变换累积器

theta: 点与 x 轴的夹角集合, 一般为 0-179 度

distance: 点到原点的距离, 即上面的所说的 r.

例:



```
import skimage.transform as st
import numpy as np
import matplotlib.pyplot as plt

# 构建测试图片
image = np.zeros((100, 100)) #背景图
idx = np.arange(25, 75)      #25-74 序列
image[idx[::-1], idx] = 255  # 线条\
image[idx, idx] = 255        # 线条/

# hough 线变换
h, theta, d = st.hough_line(image)

#生成一个一行两列的窗口（可显示两张图片）.
fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(8, 6))
plt.tight_layout()

#显示原始图片
ax0.imshow(image, plt.cm.gray)
ax0.set_title('Input image')
ax0.set_axis_off()

#显示 hough 变换所得数据
ax1.imshow(np.log(1 + h))
ax1.set_title('Hough transform')
ax1.set_xlabel('Angles (degrees)')
ax1.set_ylabel('Distance (pixels)')
ax1.axis('image')
```



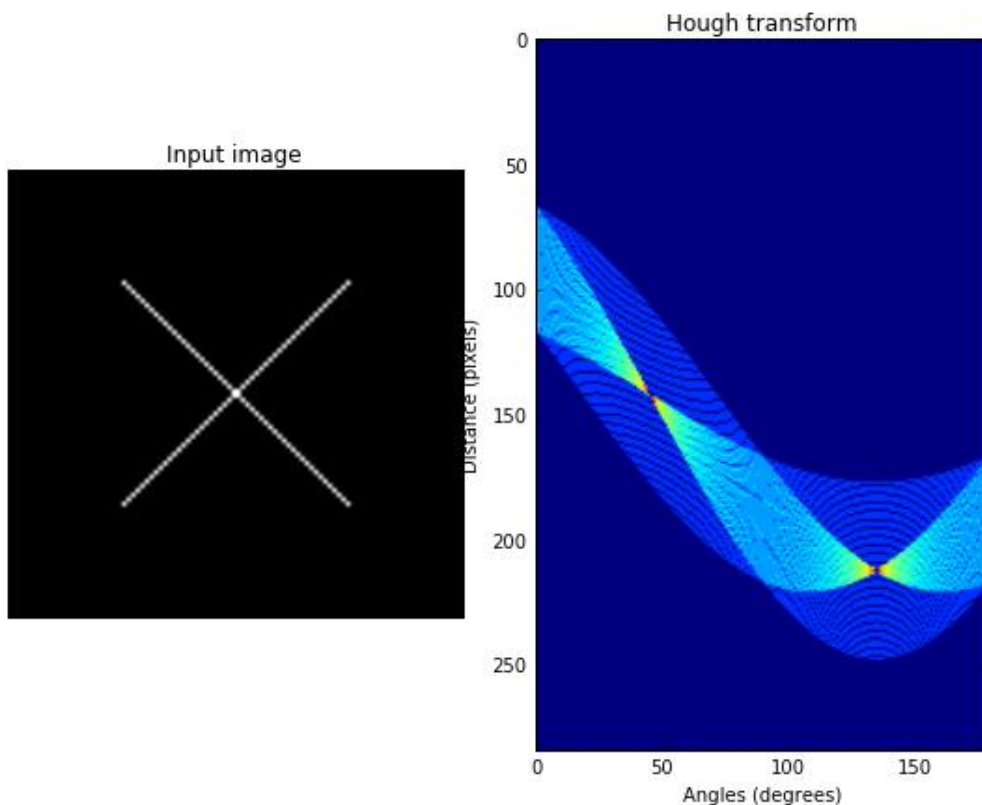


图 10.1

从右边那张图可以看出，有两个交点，说明原图像中有两条直线。

如果我们要把图中的两条直线绘制出来，则需要用到另外一个函数：

`skimage.transform.hough_line_peaks(hspace, angles, dists)`

用这个函数可以取出峰值点，即交点，也即原图中的直线。

返回的参数与输入的参数一样。我们修改一下上边的程序，在原图中将两直线绘制出来。

```
import skimage.transform as st
import numpy as np
import matplotlib.pyplot as plt

# 构建测试图片
image = np.zeros((100, 100)) #背景图
idx = np.arange(25, 75)      #25-74 序列
image[idx[::-1], idx] = 255  # 线条\
image[idx, idx] = 255        # 线条/

# hough 线变换
h, theta, d = st.hough_line(image)
```

```

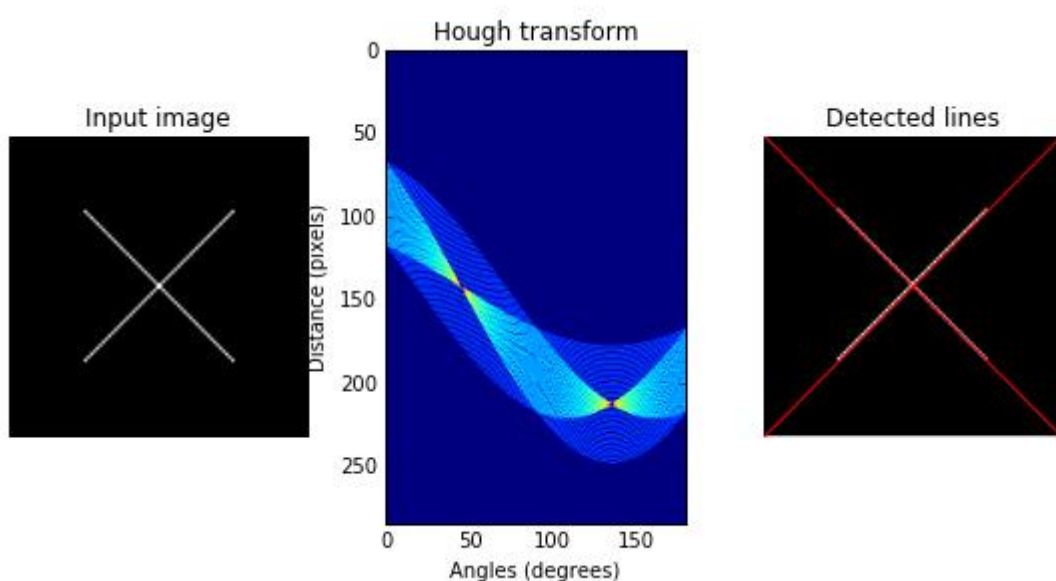
#生成一个一行三列的窗口（可显示三张图片）。
fig, (ax0, ax1, ax2) = plt.subplots(1, 3, figsize=(8, 6))
plt.tight_layout()

#显示原始图片
ax0.imshow(image, plt.cm.gray)
ax0.set_title('Input image')
ax0.set_axis_off()

#显示 hough 变换所得数据
ax1.imshow(np.log(1 + h))
ax1.set_title('Hough transform')
ax1.set_xlabel('Angles (degrees)')
ax1.set_ylabel('Distance (pixels)')
ax1.axis('image')

#显示检测出的线条
ax2.imshow(image, plt.cm.gray)
row1, col1 = image.shape
for _, angle, dist in zip(*st.hough_line_peaks(h, theta, d)):
    y0 = (dist - 0 * np.cos(angle)) / np.sin(angle)
    y1 = (dist - col1 * np.cos(angle)) / np.sin(angle)
    ax2.plot((0, col1), (y0, y1), '-r')
ax2.axis((0, col1, row1, 0))
ax2.set_title('Detected lines')
ax2.set_axis_off()

```



注意，绘制线条的时候，要从极坐标转换为笛卡尔坐标，公式为：

$$y = -\frac{\cos(\theta)}{\sin(\theta)}x + \frac{r}{\sin(\theta)}$$

`skimage` 还提供了另外一个检测直线的霍夫变换函数，概率霍夫线变换：

```
skimage.transform.probabilistic_hough_line(img, threshold=10,  
line_length=5, line_gap=3)
```

参数：

img：待检测的图像。

threshold： 阈值，可选项，默认为 10

line_length：检测的最短线条长度，默认为 50

line_gap：线条间的最大间隙。增大这个值可以合并破碎的线条。默认为 10

返回：

lines：线条列表，格式如`((x0, y0), (x1, y0))`，标明开始点和结束点。

下面，我们用 `canny` 算子提取边缘，然后检测哪些边缘是直线？



```
import skimage.transform as st
import matplotlib.pyplot as plt
from skimage import data, feature

#使用 Probabilistic Hough Transform.
image = data.camera()
edges = feature.canny(image, sigma=2, low_threshold=1,
high_threshold=25)
lines = st.probabilistic_hough_line(edges, threshold=10,
line_length=5, line_gap=3)

# 创建显示窗口.
fig, (ax0, ax1, ax2) = plt.subplots(1, 3, figsize=(16, 6))
plt.tight_layout()

#显示原图像
ax0.imshow(image, plt.cm.gray)
ax0.set_title('Input image')
ax0.set_axis_off()

#显示 canny 边缘
ax1.imshow(edges, plt.cm.gray)
```

```
ax1.set_title('Canny edges')
ax1.set_axis_off()

#用 plot 绘制出所有的直线
ax2.imshow(edges * 0)
for line in lines:
    p0, p1 = line
    ax2.plot((p0[0], p1[0]), (p0[1], p1[1]))
row2, col2 = image.shape
ax2.axis((0, col2, row2, 0))
ax2.set_title('Probabilistic Hough')
ax2.set_axis_off()
plt.show()
```



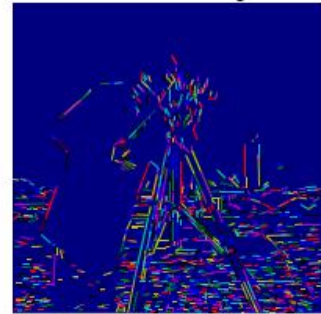
Input image



Canny edges



Probabilistic Hough



十六、霍夫圆和椭圆变换

在极坐标中，圆的表示方式为：

$$x=x_0+r\cos\theta$$

$$y=y_0+r\sin\theta$$

圆心为 (x_0, y_0) , r 为半径, θ 为旋转度数, 值范围为 0-359

如果给定圆心点和半径, 则其它点是否在圆上, 我们就能检测出来了。在图像中, 我们将每个非 0 像素点作为圆心点, 以一定的半径进行检测, 如果有一个点在圆上, 我们就对这个圆心累加一次。如果检测到一个圆, 那么这个圆心点就累加到最大, 成为峰值。因此, 在检测结果中, 一个峰值点, 就对应一个圆心点。

霍夫圆检测的函数:

`skimage.transform.hough_circle(image, radius)`

radius 是一个数组, 表示半径的集合, 如[3, 4, 5, 6]

返回一个 3 维的数组 (**radius index, M, N**), 第一维表示半径的索引, 后面两维表示图像的尺寸。

例 1: 绘制两个圆形, 用霍夫圆变换将它们检测出来。



```
import numpy as np
import matplotlib.pyplot as plt
from skimage import draw, transform, feature

img = np.zeros((250, 250, 3), dtype=np.uint8)
rr, cc = draw.circle_perimeter(60, 60, 50) #以半径 50 画一个圆
rr1, cc1 = draw.circle_perimeter(150, 150, 60) #以半径 60 画一个圆
img[cc, rr, :] = 255
img[cc1, rr1, :] = 255

fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(8, 5))

ax0.imshow(img) #显示原图
ax0.set_title('origin image')

hough_radii = np.arange(50, 80, 5) #半径范围
hough_res = transform.hough_circle(img[:, :, 0], hough_radii) #圆变换

centers = [] #保存所有圆心点坐标
accums = [] #累积值
radii = [] #半径
```

```

for radius, h in zip(hough_radii, hough_res):
    #每一个半径值，取出其中两个圆
    num_peaks = 2
    peaks = feature.peak_local_max(h, num_peaks=num_peaks) #取出峰值
    centers.extend(peaks)
    accums.extend(h[peaks[:, 0], peaks[:, 1]])
    radii.extend([radius] * num_peaks)

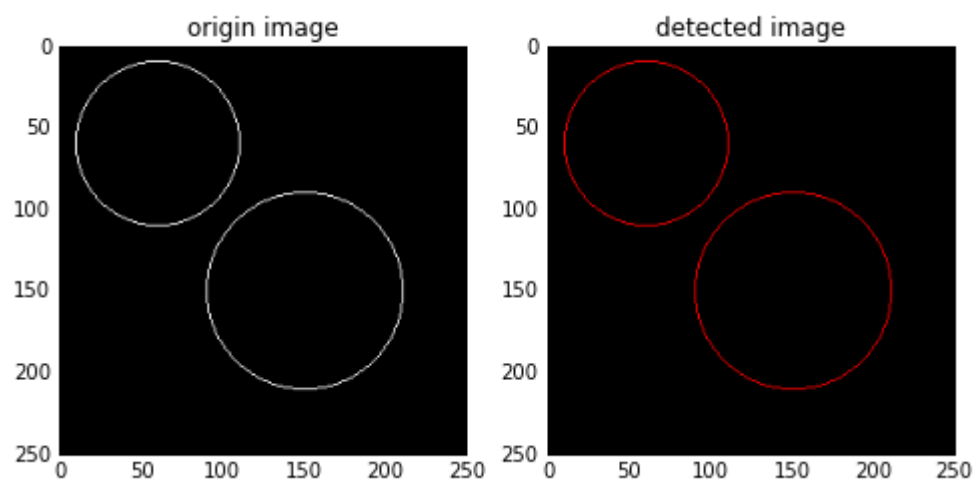
#画出最接近的圆
image = np.copy(img)
for idx in np.argsort(accums)[::-1][:2]:
    center_x, center_y = centers[idx]
    radius = radii[idx]
    cx, cy = draw.circle_perimeter(center_y, center_x, radius)
    image[cy, cx] = (255, 0, 0)

ax1.imshow(image)
ax1.set_title('detected image')

```



结果图如下：原图中的圆用白色绘制，检测出的圆用红色绘制。



例 2，检测出下图中存在的硬币。



```

import numpy as np
import matplotlib.pyplot as plt
from skimage import data, color, draw, transform, feature, util

```



```

image = util.img_as_ubyte(data.coins()[0:95, 70:370]) #裁剪原图片
edges = feature.canny(image, sigma=3, low_threshold=10,
high_threshold=50) #检测 canny 边缘

fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(8, 5))

ax0.imshow(edges, cmap=plt.cm.gray) #显示 canny 边缘
ax0.set_title('original iamge')

hough_radii = np.arange(15, 30, 2) #半径范围
hough_res = transform.hough_circle(edges, hough_radii) #圆变换

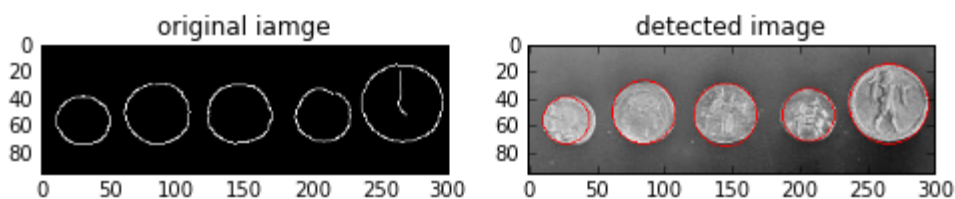
centers = [] #保存中心点坐标
accums = [] #累积值
radii = [] #半径

for radius, h in zip(hough_radii, hough_res):
    #每一个半径值，取出其中两个圆
    num_peaks = 2
    peaks = feature.peak_local_max(h, num_peaks=num_peaks) #取出峰值
    centers.extend(peaks)
    accums.extend(h[peaks[:, 0], peaks[:, 1]])
    radii.extend([radius] * num_peaks)

#画出最接近的 5 个圆
image = color.gray2rgb(image)
for idx in np.argsort(accums)[-5:]:
    center_x, center_y = centers[idx]
    radius = radii[idx]
    cx, cy = draw.circle_perimeter(center_y, center_x, radius)
    image[cy, cx] = (255, 0, 0)

ax1.imshow(image)
ax1.set_title('detected image')

```



椭圆变换是类似的，使用函数为：

```
skimage.transform.hough_ellipse(img, accuracy, threshold, min_size, max_size)
```

输入参数：

img: 待检测图像。

accuracy: 使用在累加器上的短轴二进制尺寸，是一个 **double** 型的值，默认为 1

thresh: 累加器阈值，默认为 4

min_size: 长轴最小长度，默认为 4

max_size: 短轴最大长度，默认为 **None**,表示图片最短边的一半。

返回一个 `[(accumulator, y0, x0, a, b, orientation)]` 数组，**accumulator** 表示累加器，**(y0,x0)**表示椭圆中心点，**(a,b)**分别表示长短轴，**orientation** 表示椭圆方向

例：检测出咖啡图片中的椭圆杯口

```
import matplotlib.pyplot as plt
from skimage import data, draw, color, transform, feature

#加载图片，转换成灰度图并检测边缘
image_rgb = data.coffee()[0:220, 160:420] #裁剪原图像，不然速度非常慢
image_gray = color.rgb2gray(image_rgb)
edges = feature.canny(image_gray, sigma=2.0, low_threshold=0.55,
high_threshold=0.8)

#执行椭圆变换
result =transform.hough_ellipse(edges, accuracy=20,
threshold=250,min_size=100, max_size=120)
result.sort(order='accumulator') #根据累加器排序

#估计椭圆参数
best = list(result[-1]) #排完序后取最后一个
yc, xc, a, b = [int(round(x)) for x in best[1:5]]
orientation = best[5]

#在原图上画出椭圆
cy, cx =draw.ellipse_perimeter(yc, xc, a, b, orientation)
image_rgb[cy, cx] = (0, 0, 255) #在原图中用蓝色表示检测出的椭圆

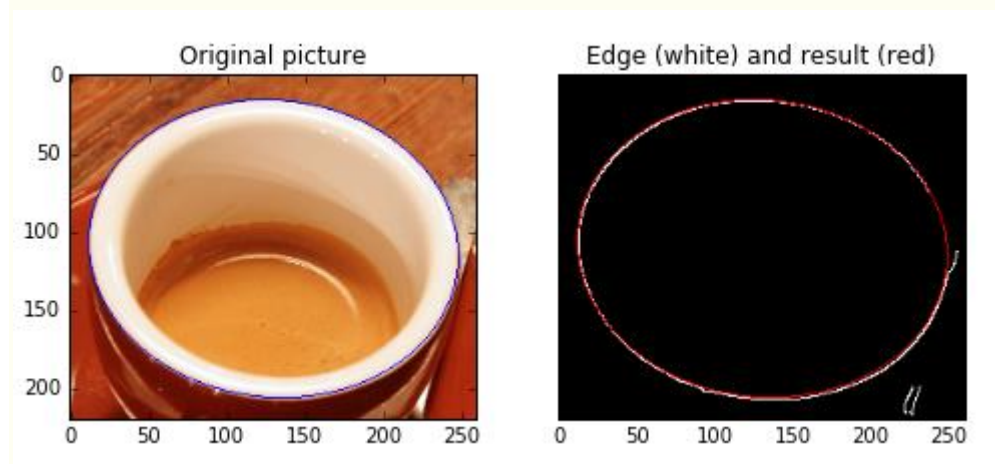
#分别用白色表示 canny 边缘，用红色表示检测出的椭圆，进行对比
edges = color.gray2rgb(edges)
edges[cy, cx] = (250, 0, 0)

fig2, (ax1, ax2) = plt.subplots(ncols=2, nrows=1, figsize=(8, 4))
```

```
ax1.set_title('Original picture')
ax1.imshow(image_rgb)

ax2.set_title('Edge (white) and result (red)')
ax2.imshow(edges)

plt.show()
```



霍夫椭圆变换速度非常慢，应避免图像太大。

十七、 边缘与轮廓

在前面的，我们已经讲解了很多算子用来检测边缘，其中用得最多的 **canny** 算子边缘检测。

本篇我们讲解一些其它方法来检测轮廓。

1、查找轮廓 (find_contours)

measure 模块中的 **find_contours()**函数，可用来检测二值图像的边缘轮廓。

函数原型为：

`skimage.measure.find_contours(array, level)`

array: 一个二值数组图像

level: 在图像中查找轮廓的级别值

返回轮廓列表集合，可用 **for** 循环取出每一条轮廓。

例 1：

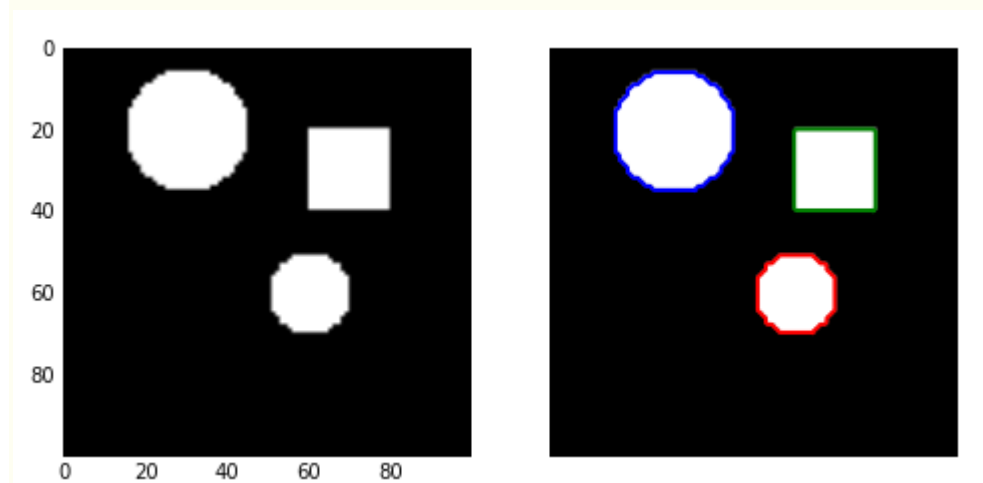
```
import numpy as np
import matplotlib.pyplot as plt
from skimage import measure, draw

#生成二值测试图像
img=np.zeros([100,100])
img[20:40,60:80]=1 #矩形
rr,cc=draw.circle(60,60,10) #小圆
rr1,cc1=draw.circle(20,30,15) #大圆
img[rr,cc]=1
img[rr1,cc1]=1

#检测所有图形的轮廓
contours = measure.find_contours(img, 0.5)

#绘制轮廓
fig, (ax0,ax1) = plt.subplots(1,2,figsize=(8,8))
ax0.imshow(img,plt.cm.gray)
ax1.imshow(img,plt.cm.gray)
for n, contour in enumerate(contours):
    ax1.plot(contour[:, 1], contour[:, 0], linewidth=2)
ax1.axis('image')
ax1.set_xticks([])
ax1.set_yticks([])
plt.show()
```

结果如下：不同的轮廓用不同的颜色显示



例 2:

```
import matplotlib.pyplot as plt
from skimage import measure, data, color

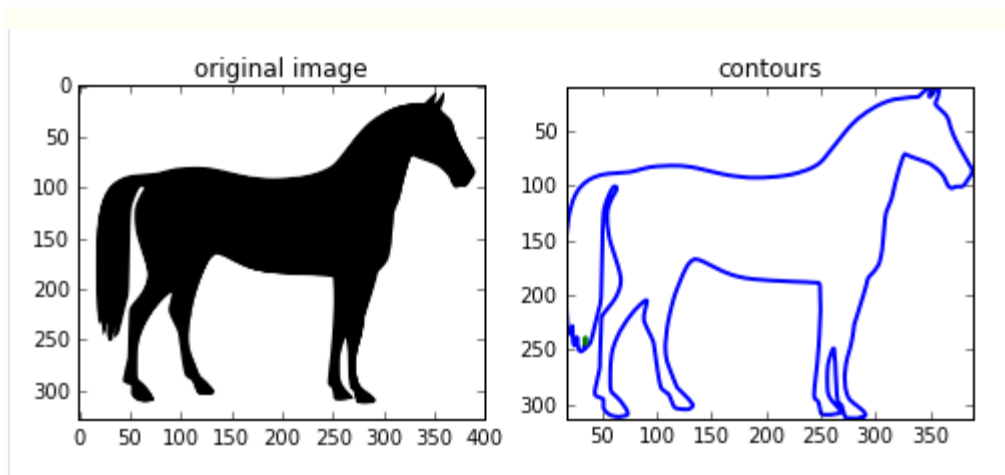
#生成二值测试图像
img=color.rgb2gray(data.horse())

#检测所有图形的轮廓
contours = measure.find_contours(img, 0.5)

#绘制轮廓
fig, axes = plt.subplots(1, 2, figsize=(8, 8))
ax0, ax1= axes.ravel()
ax0.imshow(img, plt.cm.gray)
ax0.set_title('original image')

rows, cols=img.shape
ax1.axis([0, rows, cols, 0])
for n, contour in enumerate(contours):
    ax1.plot(contour[:, 1], contour[:, 0], linewidth=2)
ax1.axis('image')
ax1.set_title('contours')
plt.show()
```





2、逼近多边形曲线

逼近多边形曲线有两个函数：`subdivide_polygon()`和 `approximate_polygon()`

`subdivide_polygon()`采用 B 样条 (B-Splines)来细分多边形的曲线，该曲线通常在凸包线的内部。

函数格式为：

`skimage.measure.subdivide_polygon(coords, degree=2, preserve_ends=False)`

coords: 坐标点序列。

degree: B 样条的度数，默认为 2

preserve_ends: 如果曲线为非闭合曲线，是否保存开始和结束点坐标，默认为 `false`

返回细分为的坐标点序列。

`approximate_polygon()` 是基于 Douglas-Peucker 算法的一种近似曲线模拟。它根据指定的容忍值来近似一条多边形曲线链，该曲线也在凸包线的内部。

函数格式为：

`skimage.measure.approximate_polygon(coords, tolerance)`

coords: 坐标点序列

tolerance: 容忍值

返回近似的多边形曲线坐标序列。

例：



```
import numpy as np
import matplotlib.pyplot as plt
from skimage import measure, data, color
```

#生成二值测试图像

```

hand = np.array([[1.64516129, 1.16145833],
                 [1.64516129, 1.59375],
                 [1.35080645, 1.921875],
                 [1.375, 2.18229167],
                 [1.68548387, 1.9375],
                 [1.60887097, 2.55208333],
                 [1.68548387, 2.69791667],
                 [1.76209677, 2.56770833],
                 [1.83064516, 1.97395833],
                 [1.89516129, 2.75],
                 [1.9516129, 2.84895833],
                 [2.01209677, 2.76041667],
                 [1.99193548, 1.99479167],
                 [2.11290323, 2.63020833],
                 [2.2016129, 2.734375],
                 [2.25403226, 2.60416667],
                 [2.14919355, 1.953125],
                 [2.30645161, 2.36979167],
                 [2.39112903, 2.36979167],
                 [2.41532258, 2.1875],
                 [2.1733871, 1.703125],
                 [2.07782258, 1.16666667]])

#检测所有图形的轮廓
new_hand = hand.copy()
for _ in range(5):
    new_hand = measure.subdivide_polygon(new_hand, degree=2)

# approximate subdivided polygon with Douglas-Peucker algorithm
appr_hand = measure.approximate_polygon(new_hand, tolerance=0.02)

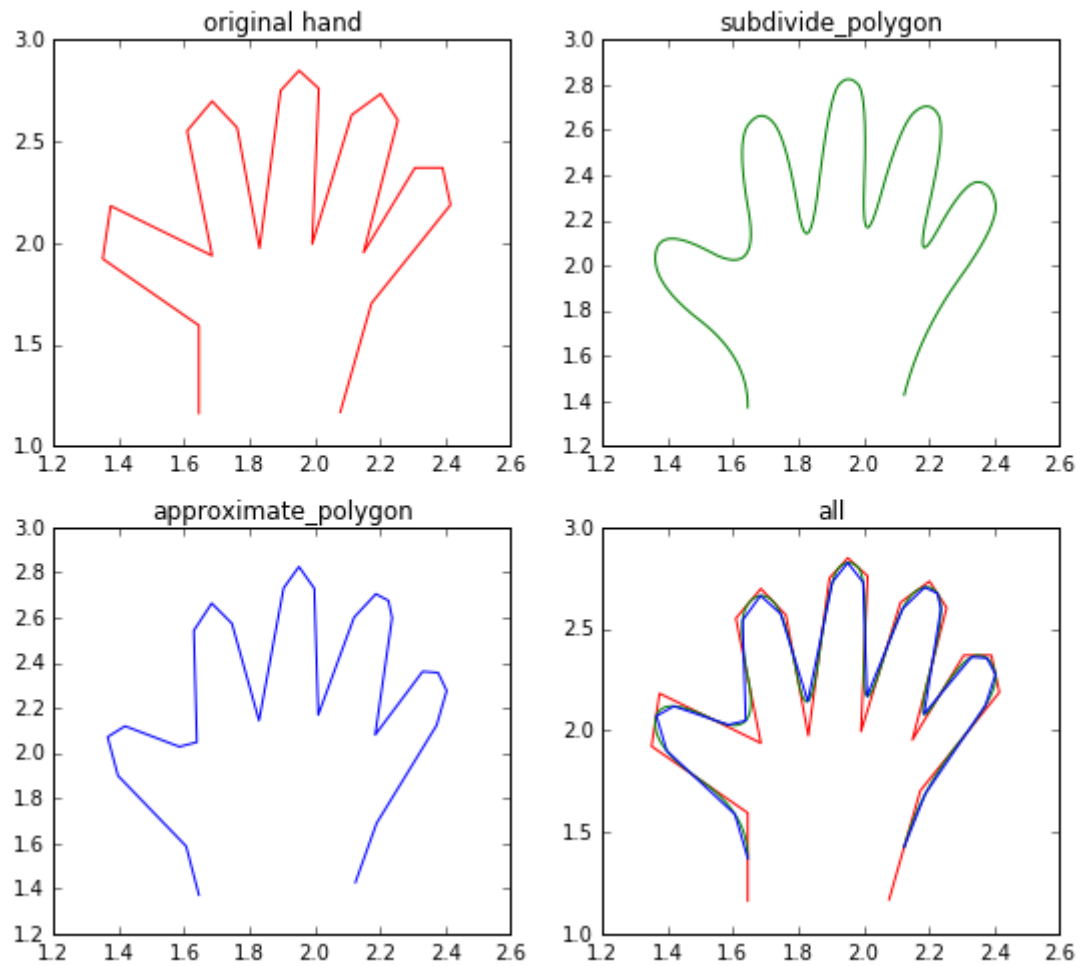
print("Number of coordinates:", len(hand), len(new_hand),
      len(appr_hand))

fig, axes = plt.subplots(2, 2, figsize=(9, 8))
ax0, ax1, ax2, ax3 = axes.ravel()

ax0.plot(hand[:, 0], hand[:, 1], 'r')
ax0.set_title('original hand')
ax1.plot(new_hand[:, 0], new_hand[:, 1], 'g')
ax1.set_title('subdivide_polygon')
ax2.plot(appr_hand[:, 0], appr_hand[:, 1], 'b')
ax2.set_title('approximate_polygon')

```

```
ax3.plot(hand[:, 0], hand[:, 1], 'r')
ax3.plot(new_hand[:, 0], new_hand[:, 1], 'g')
ax3.plot(appr_hand[:, 0], appr_hand[:, 1], 'b')
ax3.set_title('all')
```



十八、高级形态学处理

形态学处理，除了最基本的膨胀、腐蚀、开/闭运算、黑/白帽处理外，还有一些更高级的运用，如凸包，连通区域标记，删除小块区域等。

1、凸包

凸包是指一个凸多边形，这个凸多边形将图片中所有的白色像素点都包含在内。

函数为：

`skimage.morphology.convex_hull_image(image)`

输入为二值图像，输出一个逻辑二值图像。在凸包内的点为 **True**，否则为 **False**

例：



```
import matplotlib.pyplot as plt
from skimage import data,color,morphology

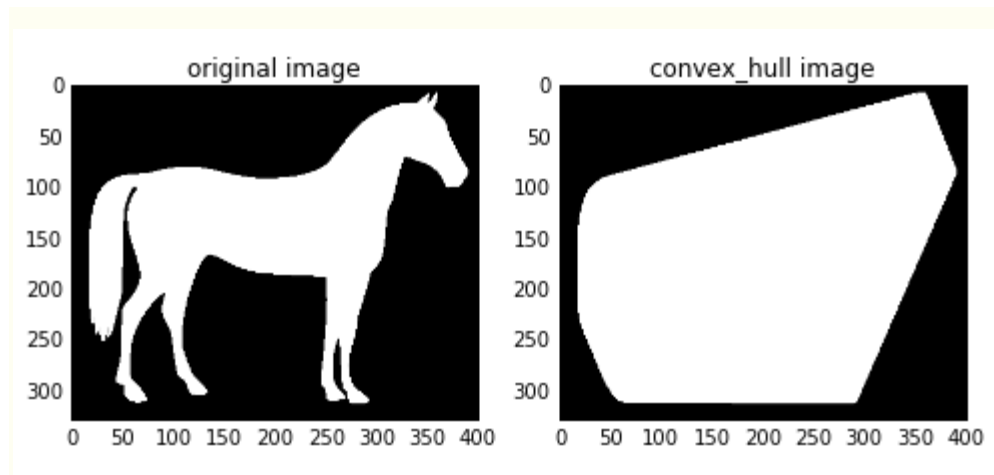
#生成二值测试图像
img=color.rgb2gray(data.horse())
img=(img<0.5)*1

chull = morphology.convex_hull_image(img)

#绘制轮廓
fig, axes = plt.subplots(1,2,figsize=(8,8))
ax0, ax1= axes.ravel()
ax0.imshow(img,plt.cm.gray)
ax0.set_title('original image')

ax1.imshow(chull,plt.cm.gray)
ax1.set_title('convex_hull image')
```





`convex_hull_image()`是将图片中的所有目标看作一个整体，因此计算出来只有一个最小凸多边形。如果图中有多个目标物体，每一个物体需要计算一个最小凸多边形，则需要使用 `convex_hull_object()` 函数。

函数格式: `skimage.morphology.convex_hull_object(image, neighbors=8)`

输入参数 `image` 是一个二值图像，`neighbors` 表示是采用 4 连通还是 8 连通，默认为 8 连通。

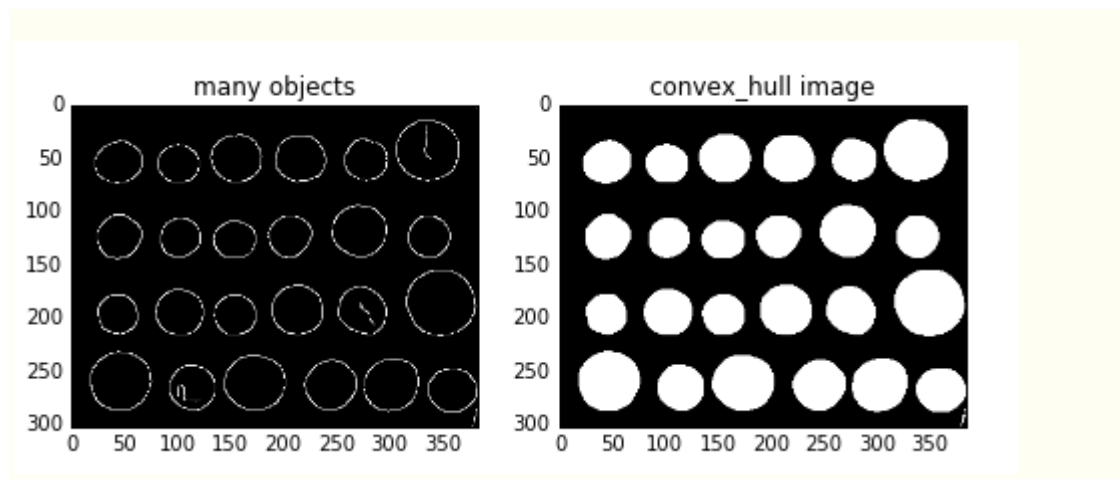
例:

```
import matplotlib.pyplot as plt
from skimage import data,color,morphology,feature

#生成二值测试图像
img=color.rgb2gray(data.coins())
#检测 canny 边缘,得到二值图片
edgs=feature.canny(img, sigma=3, low_threshold=10, high_threshold=50)

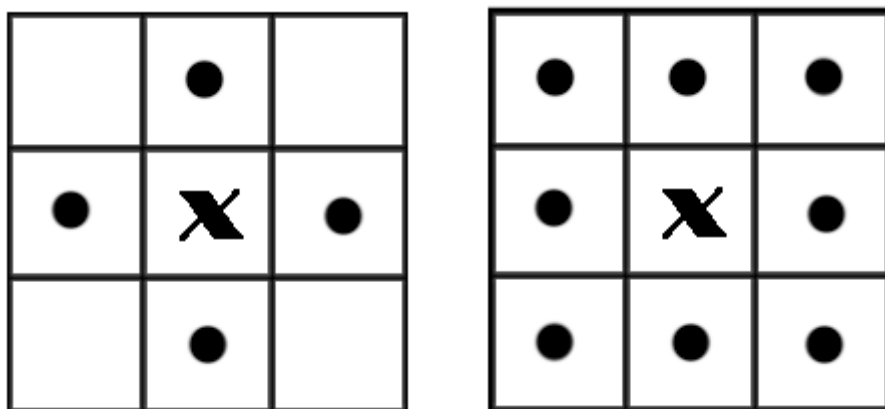
chull = morphology.convex_hull_object(edgs)

#绘制轮廓
fig, axes = plt.subplots(1,2,figsize=(8,8))
ax0, ax1= axes.ravel()
ax0.imshow(edgs,plt.cm.gray)
ax0.set_title('many objects')
ax1.imshow(chull,plt.cm.gray)
ax1.set_title('convex_hull image')
plt.show()
```



2、连通区域标记

在二值图像中，如果两个像素点相邻且值相同（同为 0 或同为 1），那么就认为这两个像素点在一个相互连通的区域内。而同一个连通区域的所有像素点，都用同一个数值来进行标记，这个过程就叫连通区域标记。在判断两个像素是否相邻时，我们通常采用 4 连通或 8 连通判断。在图像中，最小的单位是像素，每个像素周围有 8 个邻接像素，常见的邻接关系有 2 种：4 邻接与 8 邻接。4 邻接一共 4 个点，即上下左右，如下左图所示。8 邻接的点一共有 8 个，包括了对角线位置的点，如下右图所示。



在 `skimage` 包中，我们采用 `measure` 子模块下的 `label()` 函数来实现连通区域标记。

函数格式：

```
skimage.measure.label (image, connectivity=None)
```

参数中的 `image` 表示需要处理的二值图像，`connectivity` 表示连接的模式，1 代表 4 邻接，2 代表 8 邻接。

输出一个标记数组 (`labels`)，从 0 开始标记。



```
import numpy as np
import scipy.ndimage as ndi
```

```

from skimage import measure,color
import matplotlib.pyplot as plt

#编写一个函数来生成原始二值图像
def microstructure(l=256):
    n = 5
    x, y = np.ogrid[0:l, 0:l] #生成网络
    mask = np.zeros((l, l))
    generator = np.random.RandomState(1) #随机数种子
    points = l * generator.rand(2, n**2)
    mask[(points[0]).astype(np.int), (points[1]).astype(np.int)] = 1
    mask = ndi.gaussian_filter(mask, sigma=l/(4.*n)) #高斯滤波
    return mask > mask.mean()

data = microstructure(l=128)*1 #生成测试图片

labels=measure.label(data,connectivity=2) #8 连通区域标记
dst=color.label2rgb(labels) #根据不同的标记显示不同的颜色
print('regions number:', labels.max()+1) #显示连通区域块数(从 0 开始标记)

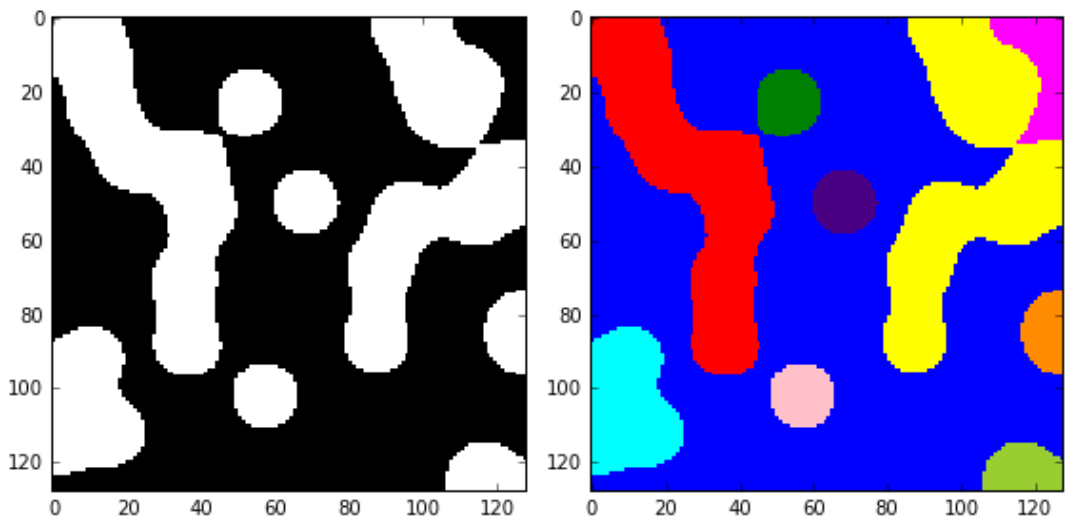
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
ax1.imshow(data, plt.cm.gray, interpolation='nearest')
ax1.axis('off')
ax2.imshow(dst,interpolation='nearest')
ax2.axis('off')

fig.tight_layout()
plt.show()

```

在代码中，有些地方乘以 1，则可以将 bool 数组快速地转换为 int 数组。

结果如图：有 10 个连通的区域，标记为 0-9



如果想分别对每一个连通区域进行操作，比如计算面积、外接矩形、凸包面积等，则需要调用 `measure` 子模块的 `regionprops`（）函数。该函数格式为：

`skimage.measure.regionprops(label_image)`

返回所有连通区块的属性列表，常用的属性列表如下表：

属性名称	类型	描述
area	int	区域内像素点总数
bbox	tuple	边界外接框 (min_row, min_col, max_row, max_col)
centroid	array	质心坐标
convex_area	int	凸包内像素点总数
convex_image	ndarray	和边界外接框同大小的凸包
coords	ndarray	区域内像素点坐标
Eccentricity	float	离心率
equivalent_diameter	float	和区域面积相同的圆的直径
euler_number	int	区域欧拉数

extent	float	区域面积和边界外接框面积的比率
filled_area	int	区域和外接框之间填充的像素点总数
perimeter	float	区域周长
label	int	区域标记

3、删除小块区域

有些时候，我们只需要一些大块区域，那些零散的、小块的区域，我们就需要删除掉，则可以使用 `morphology` 子模块的 `remove_small_objects()` 函数。

函数格式：

`skimage.morphology.remove_small_objects(ar, min_size=64, connectivity=1, in_place=False)`

参数：

ar: 待操作的 `bool` 型数组。

min_size: 最小连通区域尺寸，小于该尺寸的都将被删除。默认为 64。

connectivity: 邻接模式，1 表示 4 邻接，2 表示 8 邻接

in_place: `bool` 型值，如果为 `True`，表示直接在输入图像中删除小块区域，否则进行复制后再删除。默认为 `False`。

返回删除了小块区域的二值图像。



```
import numpy as np
import scipy.ndimage as ndi
from skimage import morphology
import matplotlib.pyplot as plt

#编写一个函数来生成原始二值图像
def microstructure(l=256):
    n = 5
    x, y = np.ogrid[0:l, 0:l] #生成网络
    mask = np.zeros((l, l))
    generator = np.random.RandomState(1) #随机数种子
    points = l * generator.rand(2, n**2)
    mask[(points[0]).astype(np.int), (points[1]).astype(np.int)] = 1
    mask = ndi.gaussian_filter(mask, sigma=l/(4.*n)) #高斯滤波
    return mask > mask.mean()
```

```
data = microstructure(l=128) #生成测试图片

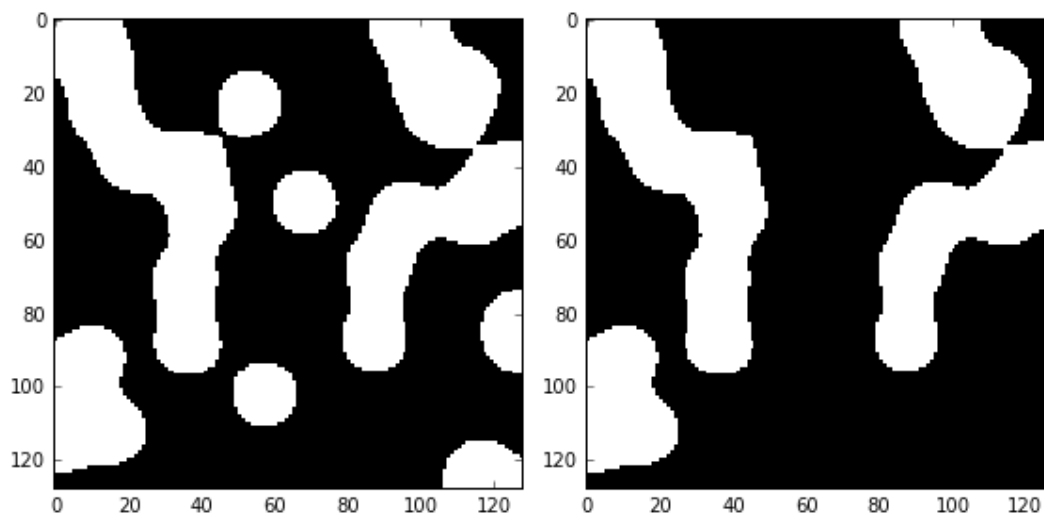
dst=morphology.remove_small_objects(data,min_size=300,connectivity=1)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
ax1.imshow(data, plt.cm.gray, interpolation='nearest')
ax2.imshow(dst,plt.cm.gray,interpolation='nearest')

fig.tight_layout()
plt.show()
```



在此例中，我们将面积小于 300 的小块区域删除（由 1 变为 0），结果如下图：



4、综合示例：阈值分割+闭运算+连通区域标记+删除小区块+分色显示



```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from skimage import data,filter,segmentation,measure,morphology,color

#加载并裁剪硬币图片
image = data.coins()[50:-50, 50:-50]

thresh =filter.threshold_otsu(image) #阈值分割
bw =morphology.closing(image > thresh, morphology.square(3)) #闭运算

cleared = bw.copy() #复制
```

```

segmentation.clear_border(cleared) #清除与边界相连的目标物

label_image =measure.label(cleared) #连通区域标记
borders = np.logical_xor(bw, cleared) #异或
label_image[borders] = -1
image_label_overlay =color.label2rgb(label_image, image=image) #不同
标记用不同颜色显示

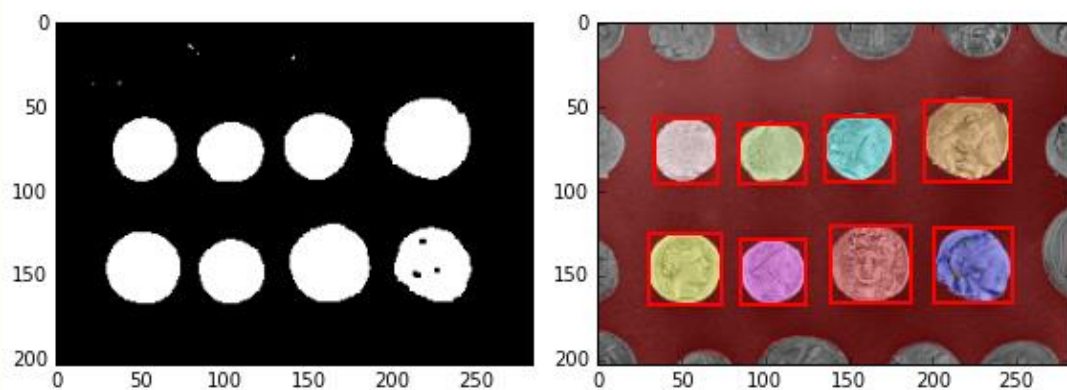
fig, (ax0, ax1)= plt.subplots(1,2, figsize=(8, 6))
ax0.imshow(cleared,plt.cm.gray)
ax1.imshow(image_label_overlay)

for region in measure.regionprops(label_image): #循环得到每一个连通区
域属性集

    #忽略小区域
    if region.area < 100:
        continue

    #绘制外包矩形
    minr, minc, maxr, maxc = region.bbox
    rect = mpatches.Rectangle((minc, minr), maxc - minc, maxr - minr,
                               fill=False, edgecolor='red',
                               linewidth=2)
    ax1.add_patch(rect)
fig.tight_layout()
plt.show()

```



十九、 骨架提取与分水岭算法

骨架提取与分水岭算法也属于形态学处理范畴，都放在 `morphology` 子模块内。

1、骨架提取


骨架提取，也叫二值图像细化。这种算法能将一个连通区域细化成一个像素的宽度，用于特征提取和目标拓扑表示。

`morphology` 子模块提供了两个函数用于骨架提取，分别是 `Skeletonize()` 函数和 `medial_axis()` 函数。我们先来看 `Skeletonize()` 函数。

格式为：`skimage.morphology.skeletonize(image)`

输入和输出都是一幅二值图像。

例 1：

```

from skimage import morphology, draw
import numpy as np
import matplotlib.pyplot as plt

#创建一个二值图像用于测试
image = np.zeros((400, 400))

#生成目标对象 1(白色 U 型)
image[10:-10, 10:100] = 1
image[-100:-10, 10:-10] = 1
image[10:-10, -100:-10] = 1

#生成目标对象 2 (X 型)
rs, cs = draw.line(250, 150, 10, 280)
for i in range(10):
    image[rs + i, cs] = 1
rs, cs = draw.line(10, 150, 250, 280)
for i in range(20):
    image[rs + i, cs] = 1

#生成目标对象 3 (O 型)
ir, ic = np.indices(image.shape)
circle1 = (ic - 135)**2 + (ir - 150)**2 < 30**2
circle2 = (ic - 135)**2 + (ir - 150)**2 < 20**2
image[circle1] = 1
image[circle2] = 0
```

#实施骨架算法

```
skeleton = morphology.skeletonize(image)
```

#显示结果

```
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))
```

```
ax1.imshow(image, cmap=plt.cm.gray)
```

```
ax1.axis('off')
```

```
ax1.set_title('original', fontsize=20)
```

```
ax2.imshow(skeleton, cmap=plt.cm.gray)
```

```
ax2.axis('off')
```

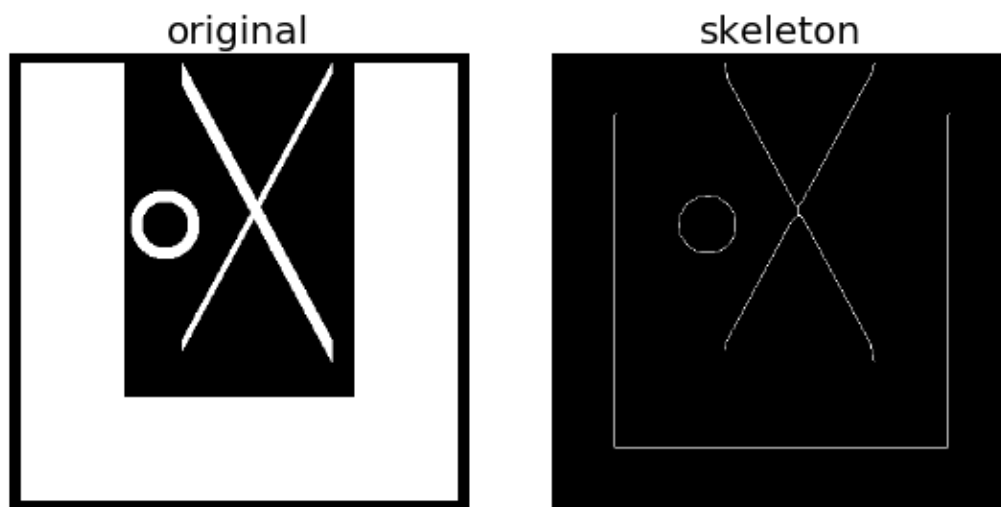
```
ax2.set_title('skeleton', fontsize=20)
```

```
fig.tight_layout()
```

```
plt.show()
```



生成一幅测试图像，上面有三个目标对象，分别进行骨架提取，结果如下：



例 2：利用系统自带的马图片进行骨架提取



```
from skimage import morphology, data, color
import matplotlib.pyplot as plt
```

```
image = color.rgb2gray(data.horse())
```

```
image = 1 - image #反相
```

#实施骨架算法

```
skeleton = morphology.skeletonize(image)
```

#显示结果

```
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))
```

```
ax1.imshow(image, cmap=plt.cm.gray)
```

```
ax1.axis('off')
```

```
ax1.set_title('original', fontsize=20)
```

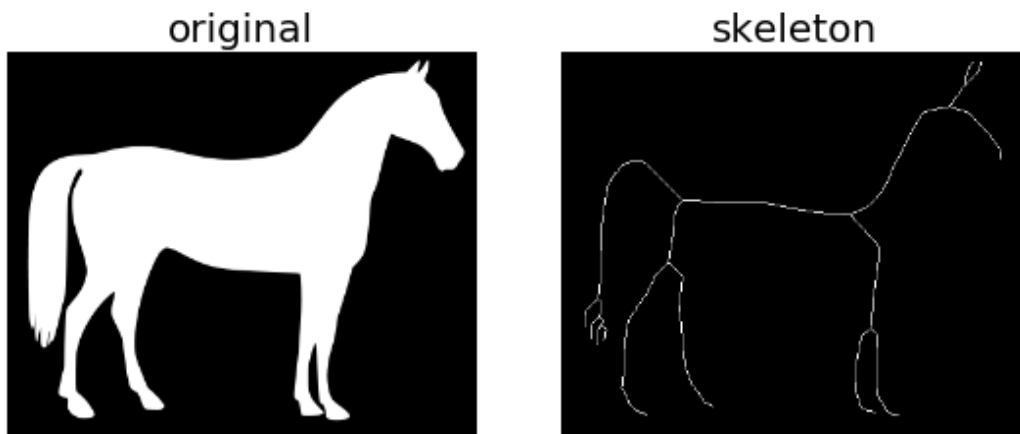
```
ax2.imshow(skeleton, cmap=plt.cm.gray)
```

```
ax2.axis('off')
```

```
ax2.set_title('skeleton', fontsize=20)
```

```
fig.tight_layout()
```

```
plt.show()
```



`medial_axis` 就是中轴的意思，利用中轴变换方法计算前景（1 值）目标对象的宽度，格式为：

```
skimage.morphology.medial_axis(image, mask=None, return_distance=False)
```

mask: 掩模。默认为 `None`，如果给定一个掩模，则在掩模内的像素值才执行骨架算法。

return_distance: bool 型值，默认为 `False`。如果为 `True`，则除了返回骨架，还将距离变换值也同时返回。这里的距离指的是中轴线上的所有点与背景点的距离。



```
import numpy as np
```

```
import scipy.ndimage as ndi
```

```
from skimage import morphology
```

```
import matplotlib.pyplot as plt
```

#编写一个函数，生成测试图像

```
def microstructure(l=256):
```

```
    n = 5
```

```

x, y = np.ogrid[0:1, 0:1]
mask = np.zeros((1, 1))
generator = np.random.RandomState(1)
points = 1 * generator.rand(2, n**2)
mask[(points[0]).astype(np.int), (points[1]).astype(np.int)] = 1
mask = ndi.gaussian_filter(mask, sigma=1/(4.*n))
return mask > mask.mean()

data = microstructure(l=64) #生成测试图像

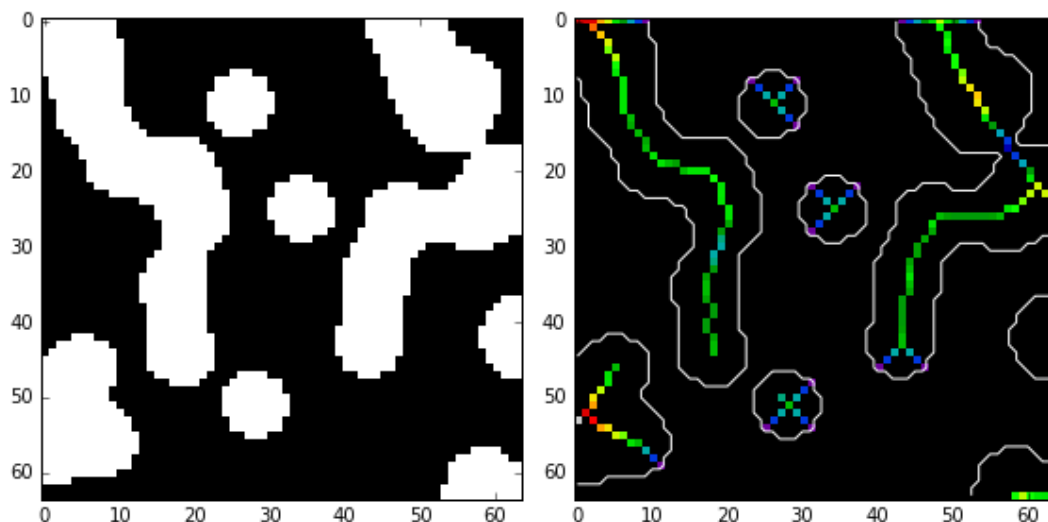
#计算中轴和距离变换值
skel, distance = morphology.medial_axis(data, return_distance=True)

#中轴上的点到背景像素点的距离
dist_on_skel = distance * skel

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
ax1.imshow(data, cmap=plt.cm.gray, interpolation='nearest')
#用光谱色显示中轴
ax2.imshow(dist_on_skel, cmap=plt.cm.spectral,
interpolation='nearest')
ax2.contour(data, [0.5], colors='w') #显示轮廓线

fig.tight_layout()
plt.show()

```



2、分水岭算法

分水岭在地理学上就是指一个山脊，水通常会沿着山脊的两边流向不同的“汇水盆”。分水岭算法是一种用于图像分割的经典算法，是基于拓扑理论的数学形态学的分割方法。如果图像中的目标物体是连在一起的，则分割起来会更困难，分水岭算法经常用于处理这类问题，通常会取得比较好的效果。

分水岭算法可以和距离变换结合，寻找“汇水盆地”和“分水岭界限”，从而对图像进行分割。二值图像的距离变换就是每一个像素点到最近非零值像素点的距离，我们可以使用 **scipy** 包来计算距离变换。

在下面的例子中，需要将两个重叠的圆分开。我们先计算圆上的这些白色像素点到黑色背景像素点的距离变换，选出距离变换中的最大值作为初始标记点（如果是反色的话，则是取最小值），从这些标记点开始的两个汇水盆越集越大，最后相交于分山岭。从分山岭处断开，我们就得到了两个分离的圆。

例 1：基于距离变换的分山岭图像分割

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage as ndi
from skimage import morphology, feature

#创建两个带有重叠圆的图像
x, y = np.indices((80, 80))
x1, y1, x2, y2 = 28, 28, 44, 52
r1, r2 = 16, 20
mask_circle1 = (x - x1)**2 + (y - y1)**2 < r1**2
mask_circle2 = (x - x2)**2 + (y - y2)**2 < r2**2
image = np.logical_or(mask_circle1, mask_circle2)

#现在我们用分水岭算法分离两个圆
distance = ndi.distance_transform_edt(image) #距离变换
local_maxi = feature.peak_local_max(distance, indices=False,
footprint=np.ones((3, 3)),
                                labels=image) #寻找峰值
markers = ndi.label(local_maxi)[0] #初始标记点
labels = morphology.watershed(-distance, markers, mask=image) #基于距离变换的分水岭算法

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(8, 8))
axes = axes.ravel()
ax0, ax1, ax2, ax3 = axes

ax0.imshow(image, cmap=plt.cm.gray, interpolation='nearest')
ax0.set_title("Original")
```

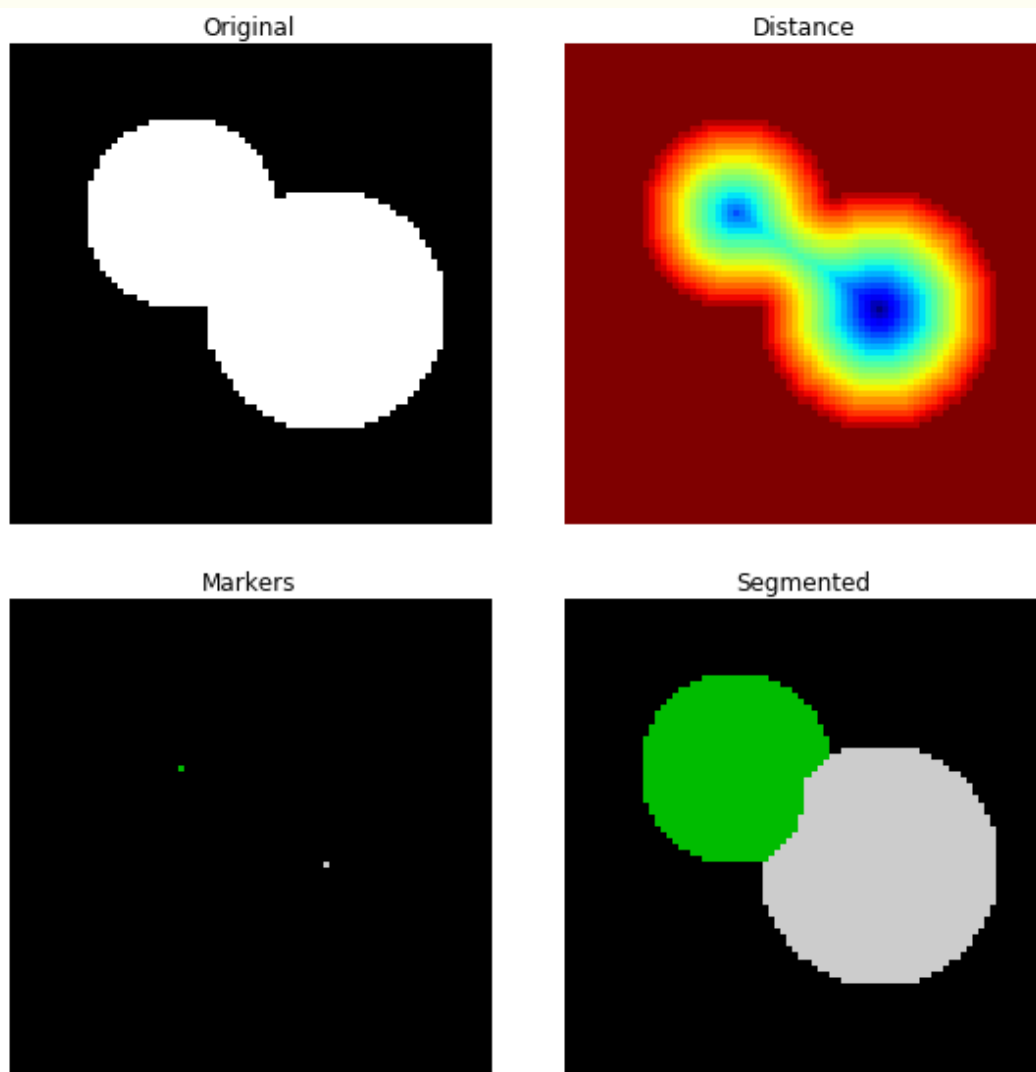
```

ax1.imshow(-distance, cmap=plt.cm.jet, interpolation='nearest')
ax1.set_title("Distance")
ax2.imshow(markers, cmap=plt.cm.spectral, interpolation='nearest')
ax2.set_title("Markers")
ax3.imshow(labels, cmap=plt.cm.spectral, interpolation='nearest')
ax3.set_title("Segmented")

for ax in axes:
    ax.axis('off')

fig.tight_layout()
plt.show()

```



分水岭算法也可以和梯度相结合，来实现图像分割。一般梯度图像在边缘处有较高的像素值，而在其它地方则有较低的像素值，理想情况下，分水岭恰好在边缘。因此，我们可以根据梯度来寻找分水岭。

例 2：基于梯度的分水岭图像分割



```
import matplotlib.pyplot as plt
from scipy import ndimage as ndi
from skimage import morphology, color, data, filter

image = color.rgb2gray(data.camera())
denoised = filter.rank.median(image, morphology.disk(2)) #过滤噪声

#将梯度值低于 10 的作为开始标记点
markers = filter.rank.gradient(denoised, morphology.disk(5)) < 10
markers = ndi.label(markers)[0]

gradient = filter.rank.gradient(denoised, morphology.disk(2)) #计算梯度
labels = morphology.watershed(gradient, markers, mask=image) #基于梯度的分水岭算法

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(6, 6))
axes = axes.ravel()
ax0, ax1, ax2, ax3 = axes

ax0.imshow(image, cmap=plt.cm.gray, interpolation='nearest')
ax0.set_title("Original")
ax1.imshow(gradient, cmap=plt.cm.spectral, interpolation='nearest')
ax1.set_title("Gradient")
ax2.imshow(markers, cmap=plt.cm.spectral, interpolation='nearest')
ax2.set_title("Markers")
ax3.imshow(labels, cmap=plt.cm.spectral, interpolation='nearest')
ax3.set_title("Segmented")

for ax in axes:
    ax.axis('off')

fig.tight_layout()
plt.show()
```

Original



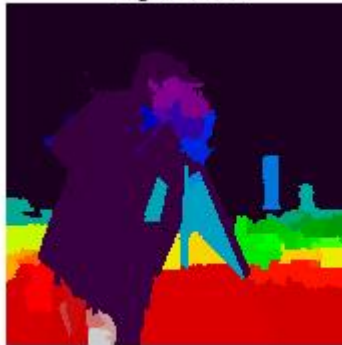
Gradient



Markers



Segmented

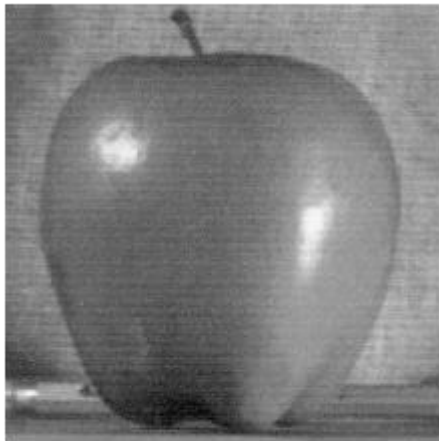


二十、频率滤波

1. `import matplotlib.pyplot as plt`
2. `import numpy as np`
3. `import cv2`
4. `%matplotlib inline`

首先读入这次需要使用的图像

1. `img = cv2.imread('apple.jpg',0) #直接读为灰度图像`
2. `plt.imshow(img,cmap="gray")`
3. `plt.axis("off")`
4. `plt.show()`



使用 numpy 带的 fft 库完成从频率域到空间域的转换。

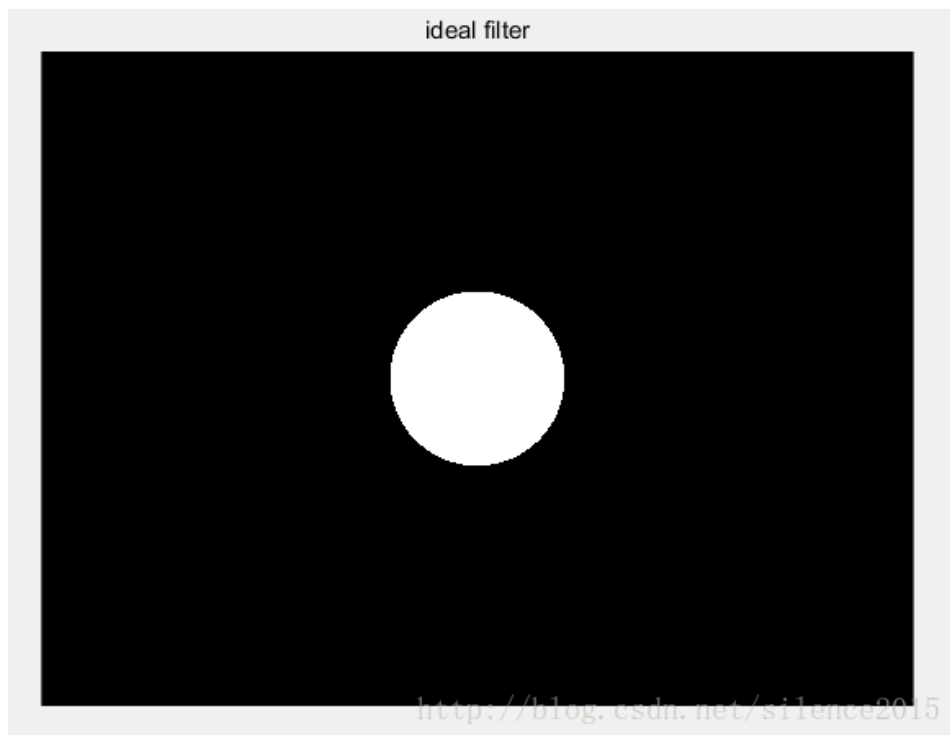
1. `f = np.fft.fft2(img)`
2. `fshift = np.fft.fftshift(f)`

低通滤波器

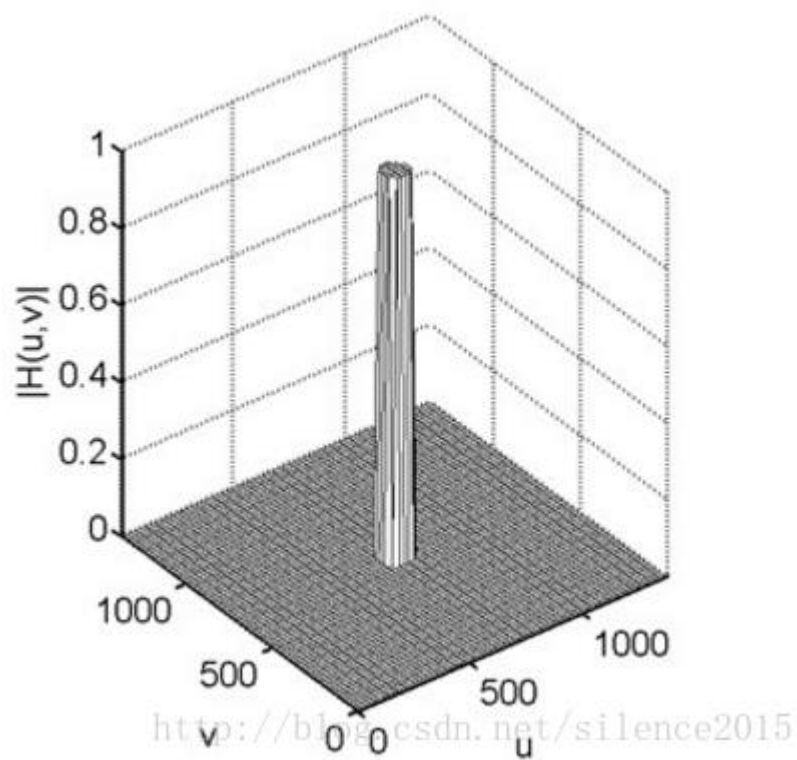
低通滤波器的公式如下

$$H(u,v)=\begin{cases} 1, & \text{if } D(u,v) \leq D_0 \\ 0, & \text{if } D(u,v) \geq D_0 \end{cases}$$

其中 $D(u,v)$ 为频率域上 (u,v) 点到中心的距离， D_0 由自己设置



白点就是所允许通过的频率范围
3d 图像如下

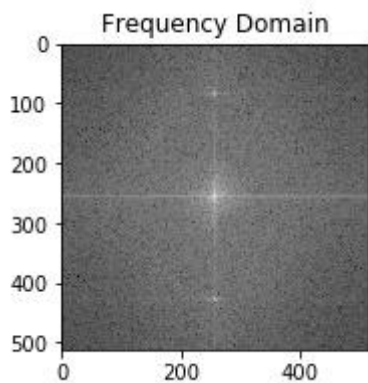


我们先把苹果转化成频率域看下效果

```

1. #取绝对值：将复数变化成实数
2. #取对数的目的是为了将数据变化到0-255
3. s1 = np.log(np.abs(fshift))
4. plt.subplot(121),plt.imshow(s1,'gray')
5. plt.title('Frequency Domain')
6. plt.show()

```



matplotlib 对于不是 uint8 的图像会自动把图像的数值缩放到 0-255 上，更多可以查看[对该问题的讨论](#)

我们在频率域上试着取不同的 d0

再将其反变换到空间域看下效果

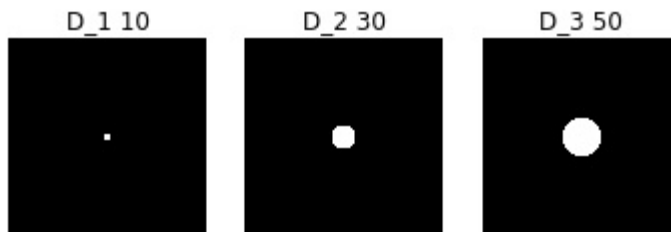
```

1. def make_transform_matrix(d,image):
2.     transfor_matrix = np.zeros(image.shape)
3.     center_point = tuple(map(lambda x:(x-1)/2,s1.shape))
4.     for i in range(transfor_matrix.shape[0]):
5.         for j in range(transfor_matrix.shape[1]):
6.             def cal_distance(pa,pb):
7.                 from math import sqrt
8.                 dis = sqrt((pa[0]-pb[0])**2+(pa[1]-pb[1])**2)
9.                 return dis
10.            dis = cal_distance(center_point,(i,j))
11.            if dis <= d:
12.                transfor_matrix[i,j]=1
13.            else:
14.                transfor_matrix[i,j]=0
15.        return transfor_matrix
16.
17. d_1 = make_transform_matrix(10,fshift)
18. d_2 = make_transform_matrix(30,fshift)
19. d_3 = make_transform_matrix(50,fshift)

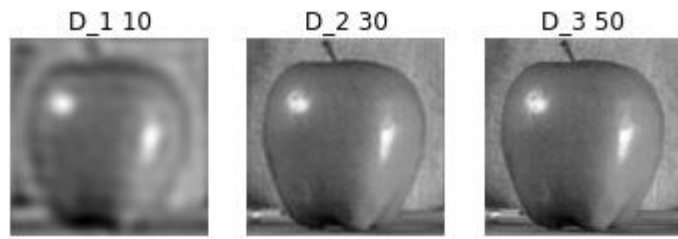
```

设定距离分别为 10,30, 50 其通过的频率的范围如图

```
1. plt.subplot(131)
2. plt.axis("off")
3. plt.imshow(d_1,cmap="gray")
4. plt.title('D_1 10')
5. plt.subplot(132)
6. plt.axis("off")
7. plt.title('D_2 30')
8. plt.imshow(d_2,cmap="gray")
9. plt.subplot(133)
10. plt.axis("off")
11. plt.title("D_3 50")
12. plt.imshow(d_3,cmap="gray")
13. plt.show()
```



```
1. img_d1 = np.abs(np.fft.ifft2(np.fft.ifftshift(fshift*d_1)))
2. img_d2 = np.abs(np.fft.ifft2(np.fft.ifftshift(fshift*d_2)))
3. img_d3 = np.abs(np.fft.ifft2(np.fft.ifftshift(fshift*d_3)))
4. plt.subplot(131)
5. plt.axis("off")
6. plt.imshow(img_d1,cmap="gray")
7. plt.title('D_1 10')
8. plt.subplot(132)
9. plt.axis("off")
10. plt.title('D_2 30')
11. plt.imshow(img_d2,cmap="gray")
12. plt.subplot(133)
13. plt.axis("off")
14. plt.title("D_3 50")
15. plt.imshow(img_d3,cmap="gray")
16. plt.show()
```



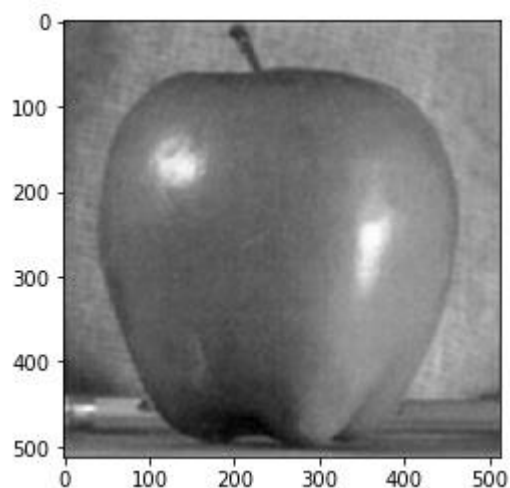
讲上面过程整理得到频率域低通滤波器的代码如下

```

1. def lowPassFilter(image,d):
2.     f = np.fft.fft2(image)
3.     fshift = np.fft.fftshift(f)
4.
5.     def make_transform_matrix(d):
6.         transfor_matrix = np.zeros(image.shape)
7.         center_point = tuple(map(lambda x:(x-1)/2,s1.shape))
8.         for i in range(transfor_matrix.shape[0]):
9.             for j in range(transfor_matrix.shape[1]):
10.                def cal_distance(pa,pb):
11.                    from math import sqrt
12.                    dis = sqrt((pa[0]-pb[0])**2+(pa[1]-pb[1])**2)
13.                    return dis
14.                dis = cal_distance(center_point,(i,j))
15.                if dis <= d:
16.                    transfor_matrix[i,j]=1
17.                else:
18.                    transfor_matrix[i,j]=0
19.            return transfor_matrix
20.        d_matrix = make_transform_matrix(d)
21.        new_img = np.abs(np.fft.ifft2(np.fft.ifftshift(fshift*d_matrix)))
22.        return new_img

plt.imshow(lowPassFilter(img,60),cmap="gray")

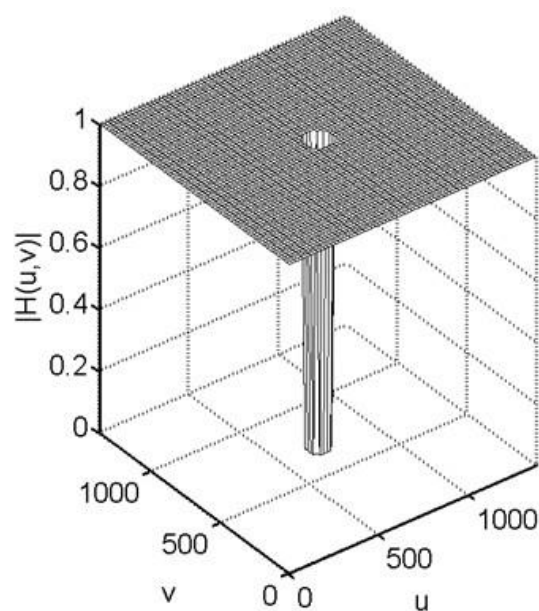
```



高通滤波器

高通滤波器同低通滤波器非常类似，只不过二者通过的波正好是相反的

$$H(u,v) = \begin{cases} 0, & \text{if } D(u,v) \leq D_0 \\ 1, & \text{if } D(u,v) \geq D_0 \end{cases}$$



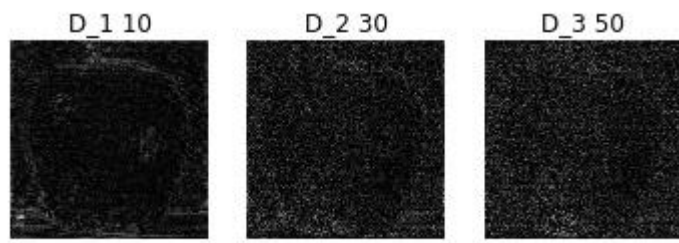
c). Ideal Highpass filter($D=60$)

```
1. def highPassFilter(image,d):
2.     f = np.fft.fft2(image)
3.     fshift = np.fft.fftshift(f)
4.     def make_transform_matrix(d):
5.         transfor_matrix = np.zeros(image.shape)
```

```

6.     center_point = tuple(map(lambda x:(x-1)/2,s1.shape))
7.     for i in range(transfor_matrix.shape[0]):
8.         for j in range(transfor_matrix.shape[1]):
9.             def cal_distance(pa,pb):
10.                 from math import sqrt
11.                 dis = sqrt((pa[0]-pb[0])**2+(pa[1]-pb[1])**2)
12.                 return dis
13.             dis = cal_distance(center_point,(i,j))
14.             if dis <= d:
15.                 transfor_matrix[i,j]=0
16.             else:
17.                 transfor_matrix[i,j]=1
18.         return transfor_matrix
19.     d_matrix = make_transform_matrix(d)
20.     new_img = np.abs(np.fft.ifft2(np.fft.ifftshift(fshift*d_matrix)))
21.     return new_img
1. img_d1 = highPassFilter(img,10)
2. img_d2 = highPassFilter(img,30)
3. img_d3 = highPassFilter(img,50)
4. plt.subplot(131)
5. plt.axis("off")
6. plt.imshow(img_d1,cmap="gray")
7. plt.title('D_1 10')
8. plt.subplot(132)
9. plt.axis("off")
10. plt.title('D_2 30')
11. plt.imshow(img_d2,cmap="gray")
12. plt.subplot(133)
13. plt.axis("off")
14. plt.title("D_3 50")
15. plt.imshow(img_d3,cmap="gray")
16. plt.show()

```

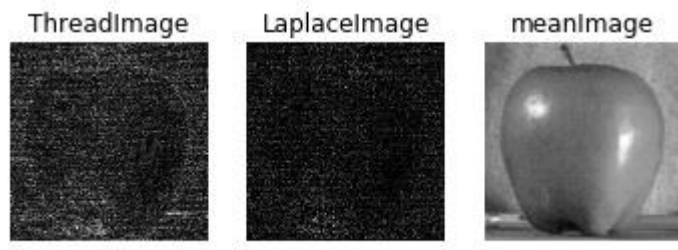


显然当 $D_0=10$

时，苹果的边缘最清楚

不同滤波的比较

```
import imagefilter
1. thread_img = imagefilter.RobertsAlogrithm(img)
2. laplace_img = imagefilter.LaplaceAlogrithm(img,"fourfields")
3. mean_img = cv2.blur(img,(3,3))
4. plt.subplot(131)
5. plt.imshow(thread_img,cmap="gray")
6. plt.title("ThreadImage")
7. plt.axis("off")
8. plt.subplot(132)
9. plt.imshow(laplace_img,cmap="gray")
10. plt.axis("off")
11. plt.title("LaplaceImage")
12. plt.subplot(133)
13. plt.imshow(mean_img,cmap="gray")
14. plt.title("meanImage")
15. plt.axis("off")
16. plt.show()
```



空间域上的平均滤波和低通滤波一样，只要起去掉无关信息，平滑图像的作用。

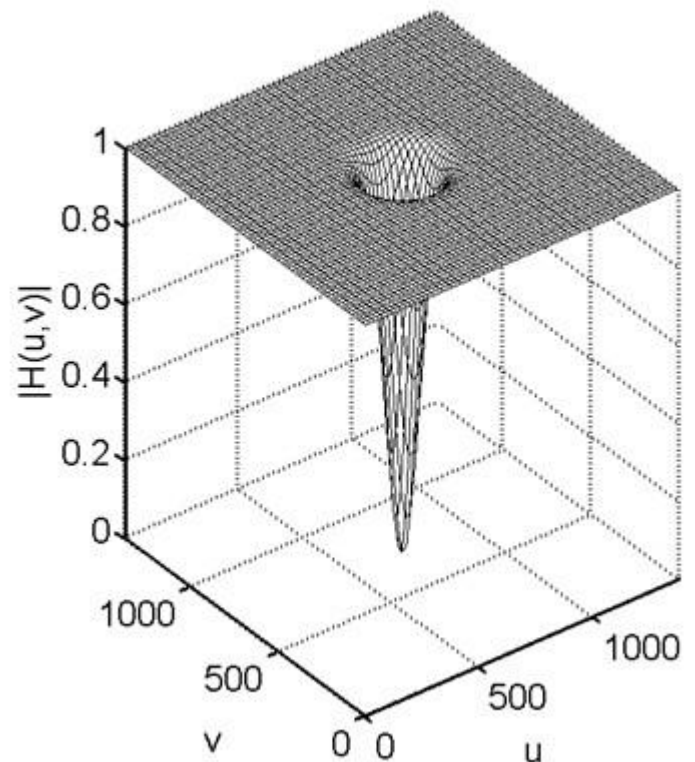
Roberts，Laplace 等滤波则起的提取边缘的作用。

频率域高通滤波器

高斯高通滤波器

频率域高斯高通滤波器的公式如下

$$H(u,v)=1-e^{-D(u,v)/D_0}$$



c) Gaussian Highpass ($D_0=60$)

```

1. def GaussianHighFilter(image,d):
2.     f = np.fft.fft2(image)
3.     fshift = np.fft.fftshift(f)
4.     def make_transform_matrix(d):
5.         transfor_matrix = np.zeros(image.shape)
6.         center_point = tuple(map(lambda x:(x-1)/2,s1.shape))
7.         for i in range(transfor_matrix.shape[0]):
8.             for j in range(transfor_matrix.shape[1]):
9.                 def cal_distance(pa,pb):
10.                     from math import sqrt
11.                     dis = sqrt((pa[0]-pb[0])**2+(pa[1]-pb[1])**2)
12.                     return dis
13.                 dis = cal_distance(center_point,(i,j))
14.                 transfor_matrix[i,j] = 1-np.exp(-(dis**2)/(2*(d**2)))
15.             return transfor_matrix
16.     d_matrix = make_transform_matrix(d)
17.     new_img = np.abs(np.fft.ifft2(np.fft.ifftshift(fshift*d_matrix)))
18.     return new_img

```

使用高斯滤波器 d 分别为 10,30,50 实现的效果

```

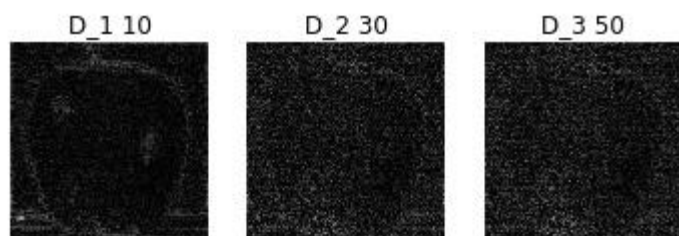
1. img_d1 = GaussianHighFilter(img,10)

```

```

2. img_d2 = GaussianHighFilter(img,30)
3. img_d3 = GaussianHighFilter(img,50)
4. plt.subplot(131)
5. plt.axis("off")
6. plt.imshow(img_d1,cmap="gray")
7. plt.title('D_1 10')
8. plt.subplot(132)
9. plt.axis("off")
10. plt.title('D_2 30')
11. plt.imshow(img_d2,cmap="gray")
12. plt.subplot(133)
13. plt.axis("off")
14. plt.title("D_3 50")
15. plt.imshow(img_d3,cmap="gray")
16. plt.show()

```



高斯低通滤波器

频率域高斯低通滤波器的公式如下

$$H(u,v)=e^{-D^2(u,v)/D_0^2}$$

```

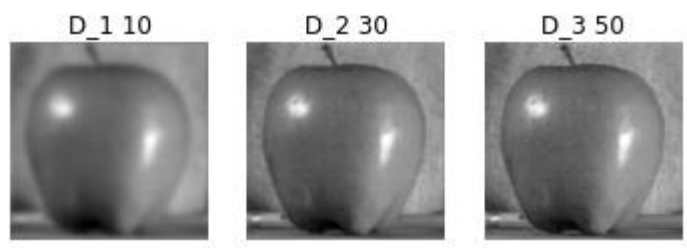
1. def GaussianLowFilter(image,d):
2.     f = np.fft.fft2(image)
3.     fshift = np.fft.fftshift(f)
4.     def make_transform_matrix(d):
5.         transfor_matrix = np.zeros(image.shape)
6.         center_point = tuple(map(lambda x:(x-1)/2,s1.shape))
7.         for i in range(transfor_matrix.shape[0]):
8.             for j in range(transfor_matrix.shape[1]):
9.                 def cal_distance(pa,pb):

```

```

10.         from math import sqrt
11.         dis = sqrt((pa[0]-pb[0])**2+(pa[1]-pb[1])**2)
12.         return dis
13.         dis = cal_distance(center_point,(i,j))
14.         transfor_matrix[i,j] = np.exp(-(dis**2)/(2*(d**2)))
15.         return transfor_matrix
16.     d_matrix = make_transform_matrix(d)
17.     new_img = np.abs(np.fft.ifft2(np.fft.ifftshift(fshift*d_matrix)))
18.     return new_img
1. img_d1 = GaussianLowFilter(img,10)
2. img_d2 = GaussianLowFilter(img,30)
3. img_d3 = GaussianLowFilter(img,50)
4. plt.subplot(131)
5. plt.axis("off")
6. plt.imshow(img_d1,cmap="gray")
7. plt.title('D_1 10')
8. plt.subplot(132)
9. plt.axis("off")
10. plt.title('D_2 30')
11. plt.imshow(img_d2,cmap="gray")
12. plt.subplot(133)
13. plt.axis("off")
14. plt.title("D_3 50")
15. plt.imshow(img_d3,cmap="gray")
16. plt.show()

```



空间域的高斯滤波

通常空间域使用高斯滤波来平滑图像，在上一篇已经写过，直接使用上篇文章的代码。

```

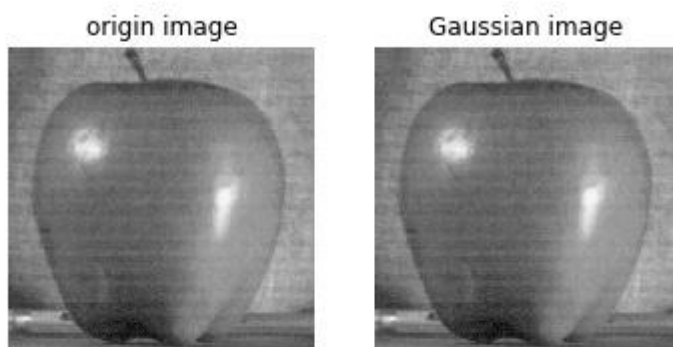
1. def GaussianOperator(roi):
2.     GaussianKernel = np.array([[1,2,1],[2,4,2],[1,2,1]])
3.     result = np.sum(roi*GaussianKernel/16)
4.     return result

```

```

5.
6. def GaussianSmooth(image):
7.     new_image = np.zeros(image.shape)
8.     image = cv2.copyMakeBorder(image,1,1,1,1,cv2.BORDER_DEFAULT)
9.     for i in range(1,image.shape[0]-1):
10.        for j in range(1,image.shape[1]-1):
11.            new_image[i-1,j-1] =GaussianOperator(image[i-1:i+2,j-
12.            1:j+2])
13.
14. new_apple = GaussianSmooth(img)
15. plt.subplot(121)
16. plt.axis("off")
17. plt.title("origin image")
18. plt.imshow(img,cmap="gray")
19. plt.subplot(122)
20. plt.axis("off")
21. plt.title("Gaussian image")
22. plt.imshow(img,cmap="gray")
23. plt.subplot(122)
24. plt.axis("off")
25. plt.show()

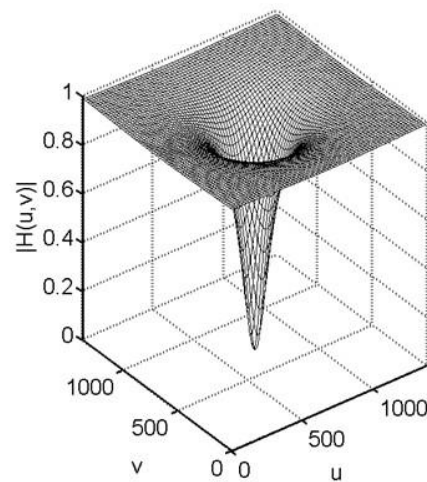
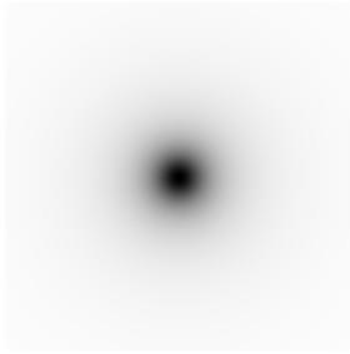
```



巴特沃斯滤波器

无论是低通滤波器，高通滤波器都是粗暴的一刀切，正如之前那么多空间域的滤波器一样，我们希望它通过的频率和与中心线性相关。

$$h(u,v)=11+(D_0/D(u,v))^{2n}$$



c) Butterworth Lowpass ($D_0=100, n=1$)

```

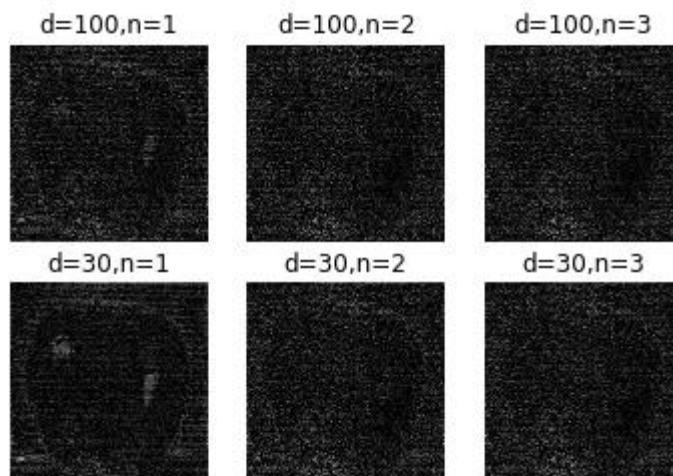
1. def butterworthPassFilter(image,d,n):
2.     f = np.fft.fft2(image)
3.     fshift = np.fft.fftshift(f)
4.
5.     def make_transform_matrix(d):
6.         transfor_matrix = np.zeros(image.shape)
7.         center_point = tuple(map(lambda x:(x-1)/2,s1.shape))
8.         for i in range(transfor_matrix.shape[0]):
9.             for j in range(transfor_matrix.shape[1]):
10.                 def cal_distance(pa,pb):
11.                     from math import sqrt
12.                     dis = sqrt((pa[0]-pb[0])**2+(pa[1]-pb[1])**2)
13.                     return dis
14.                 dis = cal_distance(center_point,(i,j))
15.                 transfor_matrix[i,j] = 1/((1+(d/dis)**n))
16.             return transfor_matrix
17.         d_matrix = make_transform_matrix(d)
18.         new_img = np.abs(np.fft.ifft2(np.fft.ifftshift(fshift*d_matrix)))
19.         return new_img
20.
21. plt.subplot(231)
22. butter_100_1 = butterworthPassFilter(img,100,1)
23. plt.imshow(butter_100_1,cmap="gray")
24. plt.title("d=100,n=1")
25. plt.axis("off")
26. plt.subplot(232)
27. butter_100_2 = butterworthPassFilter(img,100,2)
28. plt.imshow(butter_100_2,cmap="gray")
29. plt.title("d=100,n=2")
30. plt.axis("off")

```

```

11. plt.subplot(233)
12. butter_100_3 = butterworthPassFilter(img,100,3)
13. plt.imshow(butter_100_3,cmap="gray")
14. plt.title("d=100,n=3")
15. plt.axis("off")
16. plt.subplot(234)
17. butter_100_1 = butterworthPassFilter(img,30,1)
18. plt.imshow(butter_100_1,cmap="gray")
19. plt.title("d=30,n=1")
20. plt.axis("off")
21. plt.subplot(235)
22. butter_100_2 = butterworthPassFilter(img,30,2)
23. plt.imshow(butter_100_2,cmap="gray")
24. plt.title("d=30,n=2")
25. plt.axis("off")
26. plt.subplot(236)
27. butter_100_3 = butterworthPassFilter(img,30,3)
28. plt.imshow(butter_100_3,cmap="gray")
29. plt.title("d=30,n=3")
30. plt.axis("off")
31. plt.show()

```



可以明显的观察出过大的 n 造成的振铃现象

```

1. butter_5_1 = butterworthPassFilter(img,5,1)
2. plt.imshow(butter_5_1,cmap="gray")
3. plt.title("d=5,n=3")
4. plt.axis("off")
5. plt.show()

```

$d=5, n=3$

