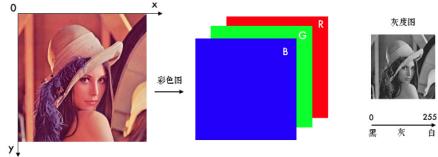


OpenCV

参考文献：<https://codec.wang/docs/opencv/start/open-camera>

opencv



```
1 # 形状中包括行数、列数和通道数  
2 height, width,  
    channels =  
    img.shape  
3 # img 是灰度图的话: height, width  
    = img.shape
```

裁剪图片【Cut Image】



```
1 x1,y1,x2,y2 =  
    115,130,200,200  
2 image_cut =  
    image[y1:y2,x1:x2]  
3  
4 img[y,x]获取/设置像  
    素点值,  
5 img.shape: 图片的形  
    状 (行数、列数、通道  
    数)  
6 img.dtype: 图像的数  
    据类型。
```

图像混合

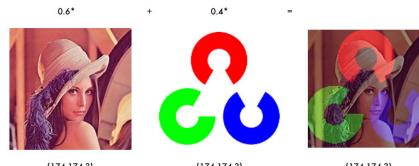
图片相加

要叠加两张图片，可以用 `cv2.add()` 函数，相加两幅图片的形状（高度/宽度/通道数）必须相同。numpy 中可以直接用 `res = img + img1` 相加，但这两者的结果并不相同。如果是二值化图片（只有 0 和 255 两种值），两者结果是一样的（用 numpy 的方式更简便一些）

图像混合

图像混合

`cv2.addWeighted()` 也是一种图片相加的操作，只不过两幅图片的权重不一样



按位操作

如果将两幅图片直接相加会改变图片的颜色，如果用图像混合，则会改变图片的透明度，所以我们需要用按位操作



腐蚀与膨胀

形态学操作一般作用于二值化图，腐蚀和膨胀是针对图片中的白色部分

形态学操作：改变物体的形状



腐蚀

腐蚀的效果是把图片“变瘦”，其原理是在原图的小区域内取局部最小值。因为是二值化图，只有 0 和 255，所以小区域内有一个是 0 该像素点就为 0：



膨胀

膨胀与腐蚀相反，取的是局部最大值，效果是把图片“变胖”

开/闭运算

先腐蚀后膨胀叫开运算（因为先腐蚀会分开物体，这样容易记住），其作用是：分离物体，消除小区域。这类形态学操作用

`cv2.morphologyEx()` 函数实现

```

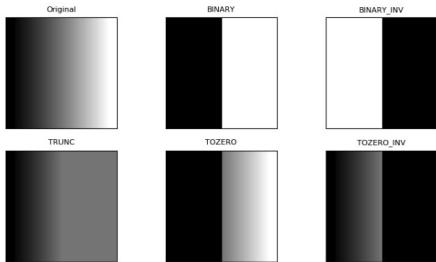
7 img[y1:y2,x1:x2]进行 ROI 截取,
8 cv2.split()/cv2.merge()通道分割/合并。
9 更推荐的获取单通道方式: b = img[:, :, 0]

```

HSV 颜色模型常用于颜色识别。要想知道某种颜色在 HSV 下的值，可以将它的 BGR 值用 `cvtColor()` 转换得到

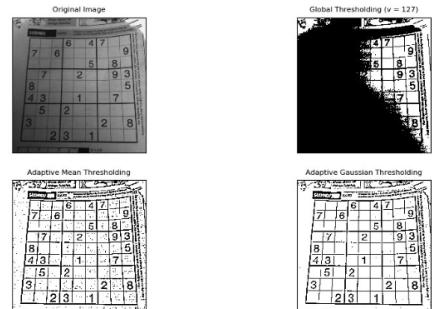
固定阈值分割

`cv2.threshold`



自适应阈值

`cv2.adaptiveThreshold()`



`cv2.resize()` 缩放图片，可以按指定大小缩放，也可以按比例缩放。

`cv2.flip()` 翻转图片，可以指定水平/垂直/水平垂直翻转三种方式。

平移/旋转是靠仿射变换

`cv2.warpAffine()` 实现

```

1 img1 =
  cv2.imread('lena.jpg')
2 img2 =
  cv2.imread('opencv-logo-white.png')
3
4 # 把 logo 放在左上角, 所以我们只关心这一块区域
5 rows, cols =
  img2.shape[:2]
6 roi = img1[:rows, :cols]
7
8 # 创建掩膜
9 img2gray =
  cv2.cvtColor(img2,
  cv2.COLOR_BGR2GRAY)
10 ret, mask =
  cv2.threshold(img2gray, 10, 255,
  cv2.THRESH_BINARY)
11 mask_inv =
  cv2.bitwise_not(mask)
12
13 # 保留除 logo 外的背景
14 img1_bg =
  cv2.bitwise_and(roi,
  roi,
  mask=mask_inv)
15 dst =
  cv2.add(img1_bg,
  img2) # 进行融合
16 img1[:rows, :cols] = dst # 融合后放在原图上

```

闭运算则相反：先膨胀后腐蚀
(先膨胀会使白色的部分扩张，以至于消除/"闭合"物体里面的小黑洞，所以叫闭运算)



其他形态学操作

- 形态学梯度：膨胀图减去腐蚀图，`dilation - erosion`，这样会得到物体的轮廓：
- 顶帽：原图减去开运算后的图【留下图中的小点】
- 黑帽：闭运算后的图减去原图【留下物体内的小点】

轮廓

寻找轮廓的操作一般用于二值化图，所以通常会使用阈值分割或 Canny 边缘检测先得到二值图

经验之谈：寻找轮廓是针对白色物体的，一定要保证物体是白色，而背景是黑色，不然很多人在寻找轮廓时会找到图片最外面的一个框

使用

`cv2.findContours()` 寻找轮廓：

```

1 img =
  cv2.imread('handwriting.jpg')
2 img_gray =
  cv2.cvtColor(img,

```

平滑图像【低通滤波】

滤波与模糊

的。

`cv2.line()` 画直线,
`cv2.circle()` 画圆,
`cv2.rectangle()` 画矩形, `cv2.ellipse()` 画椭圆, `cv2.polyline()` 画多边形, `cv2.putText()` 添加文字。

画多条直线时,

`cv2.polyline()` 要比 `cv2.line()` 高效很多

常用图片格式

- `bmp`
 - 全称: Bitmap
 - 不压缩
- `jpg`
 - 全称: Joint Photographic Experts Group
 - 有损压缩方式
- `png`
 - 全称: Portable Network Graphics
 - 无损压缩方式

OpenCV 中的图像是以 BGR 的通道顺序存储的, 但 Matplotlib 是以 RGB 模式显示的

`img[:, :, 0]` 表示图片的蓝色通道, `img[:, :, ::-1]` 就表示 BGR 翻转, 变成 RGB, 说明一下:

熟悉 Python 的童鞋应该知道, 对一个字符串 s 翻转可以这样写: `s[::-1]`, 'abc' 变成 'cba', -1 表示逆序。图片是二维的, 所以完整地复制一幅图像就是:

```
img2 = img[:, :] 写全就是: img2  
= img[0:height, 0:width  
]
```

- 它们都属于卷积, 不同滤波方法之间只是卷积核不同(对线性滤波而言)
- 低通滤波器是模糊, 高通滤波器是锐化

低通滤波器就是允许低频信号通过, 在图像中边缘和噪点都相当于高频部分, 所以低通滤波器用于去除噪点、平滑和模糊图像。高通滤波器则反之, 用来增强图像边缘, 进行锐化处理。

常见噪声有 **椒盐噪声** 和 **高斯噪声**, 椒盐噪声可以理解为斑点, 随机出现在图像中的黑点或白点; 高斯噪声可以理解为拍摄图片时由于光照等原因造成的声音。

均值滤波

均值滤波是一种最简单的滤波处理, 它取的是卷积核区域内元素的 **均值**, 用 `cv2.blur()` 实现, 如 3×3 的卷积核:

$$kernel = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

```
img = cv2.imread('lena.jpg')  
blur = cv2.blur(img, (3, 3)) # 均值滤波
```

所有的滤波函数都有一个可选参数 `borderType`, 这个参数就是 **卷积基底**(图片边框)中所说的边框填充方式。

方框滤波

方框滤波跟均值滤波很像, 如 3×3 的滤波核如下:

$$k = a \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

用 `cv2.boxFilter()` 函数实现, 当可选参数 `normalize = True` 的时候, 方框滤波就是 **均值滤波**, 上式中的 a 就等于 1/9; `normalize = False` 的时候, `a=1`, 相当于求区域内的像素和。

```
# 前面的均值滤波也可以用方框滤波实现: normalize=True  
blur = cv2.boxFilter(img, -1, (3, 3), normalize=True)
```

高斯滤波

前面两种滤波方式, 卷积核内的每个值都一样, 也就是说图像区域中每个像素的权重也就一样。高斯滤波的卷积核权重并不相同: 中间像素点权重最高, 越远离中心的像素权重越小

```
cv2.COLOR_BGR2GRAY  
)
```

```
3 ret, thresh =  
cv2.threshold(img_  
gray, 0, 255,  
cv2.THRESH_BINARY_  
INV +  
cv2.THRESH_OTSU)
```

4

```
5 # 寻找二值化图中的轮廓
```

```
6 image, contours,  
hierarchy =  
cv2.findContours(  
7 thresh,  
cv2.RETR_TREE,  
cv2.CHAIN_APPROX_S  
IMPLE)
```

```
8 print(len(contours  
))) # 结果应该为 2
```

函数有 3 个返回值, `image` 还是原来的二值化图片, `hierarchy` 是轮廓间的层级关系

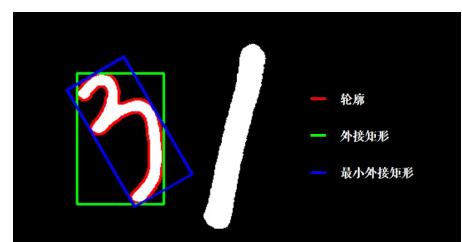
(**番外篇: 轮廓层级**), 这两个暂时不用理会。我们主要看 `contours`, 它就是找到的轮廓了, 以数组形式存储, 记录了每条轮廓的所有像素点的坐标 (x, y)

`findContours()` 返回值 `contours`
中包含了轮廓所有像素点坐标:

```
[[173, 40]],  
[[172, 41]],  
[[169, 41]],  
[[168, 42]],  
[[162, 42]],  
[[161, 43]],  
.....
```

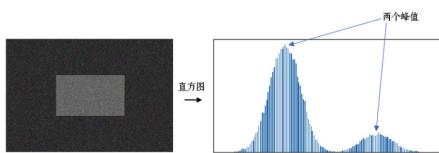


轮廓特征



而图片是有三个通道，相当于一个长度为 3 的字符串，所以通道翻转与图片复制组合起来便是 `img[:, :, ::-1]`

直方图就是每个值（0~255）的像素点个数统计



仿射变换

基本的图像变换就是二维坐标系的变换：从一种二维坐标 (x,y) 到另一种二维坐标 (u,v) 的线性变换：

$$\begin{aligned} u &= a_1x + b_1y + c_1 \\ v &= a_2x + b_2y + c_2 \end{aligned}$$

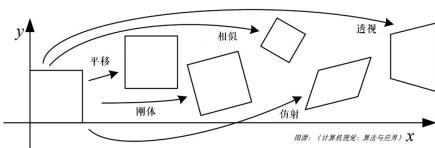
如果写成矩阵的形式，就是：

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

作如下定义：

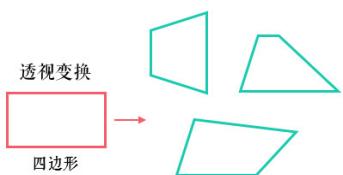
$$R = \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix}, t = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}, T = [R \quad t]$$

矩阵 $T(2 \times 3)$ 就称为仿射变换的变换矩阵， R 为线性变换矩阵， t 为平移矩阵，简单来说，仿射变换就是线性变换 + 平移。变换后直线依然还是直线，平行线依然是平行线，直线间的相对位置关系不变，因此非共线的三个对应点便可确定唯一的一个仿射变换，线性变换 4 个自由度 + 平移 2 个自由度 → 仿射变换自由度为 6。



透视变换

透视变换 (Perspective Transformation) 是将二维的图片投影到一个三维视平面上，然后再转换到二维坐标下，所以也称为投影映射 (Projective Mapping)。简单来说就是二维 → 三维 → 二维的一个过程。



OpenCV 中首先根据变换前后的四个点用 `cv2.getPerspectiveTransform()` 生成 3×3 的变换矩阵

OpenCV 中对应函数为 `cv2.GaussianBlur(src, ksize, sigmaX)`:

```
img = cv2.imread('gaussian_noise.bmp')
# 均值滤波 vs 高斯滤波
blur = cv2.blur(img, (5, 5)) # 均值滤波
gaussian = cv2.GaussianBlur(img, (5, 5), 1) # 高斯滤波
```

参数 3 σ_x 值越大，模糊效果越明显，高斯滤波相比均值滤波效率要慢，但可以有效消除高斯噪音，能保留更多的图像细节，所以经常被称为最有用的滤波器。均值滤波与高斯滤波的对比结果如下：(均值滤波丢失的细节更多)：



中值滤波

中值又叫中位数，是所有数排序后取中间的值。中值滤波就是用区域内的中值来代替本像素值，所以那种孤立的斑点，如 0 或 255 很容易消除掉，适用于去除椒盐噪声和斑点噪声。中值是一种非线性操作，效率相比前面几种线性滤波要慢

```
img = cv2.imread('salt_noise.bmp', 0)
# 均值滤波 vs 中值滤波
blur = cv2.blur(img, (5, 5)) # 均值滤波
median = cv2.medianBlur(img, 5) # 中值滤波
```



双边滤波

模糊操作基本都会损失掉图像细节信息，尤其前面介绍的线性滤波器，图像的边缘信息很难保留下。然而，边缘 (edge) 信息是图像中很重要的一个特征，所以这才有了**双边滤波**。用

`cv2.bilateralFilter()`

函数实现：



轮廓面积

```
1 area =
cv2.contourArea(cnt)
t) # 4386.5
```

注意轮廓特征计算的结果并不等同于像素点的个数，而是根据几何方法算出来的，所以有小数。

如果统计二值图中像素点个数，应尽量避免循环，可以使

用 `cv2.countNonZero()`，更加高效。

轮廓周长

```
1 perimeter =
cv2.arcLength(cnt,
True) # 585.7
```

参数 2 表示轮廓是否封闭，显然我们的轮廓是封闭的，所以是 True

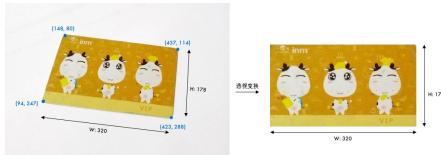
外接矩形

```
1 x, y, w, h =
cv2.boundingRect(cnt) # 外接矩形
2 cv2.rectangle(img,
color1, (x, y),
(x + w, y + h),
(0, 255, 0), 2)
3
4 rect =
cv2.minAreaRect(cnt) # 最小外接矩形
5 box =
np.int0(cv2.boxPoi
```

阵，然后再用

```
cv2.warpPerspective()
```

进行透视变换。实战演练一下：



```
1 img =
2   cv2.imread('card.jpg')
3
4 # 原图中卡片的四个角
5 pts1 =
6   np.float32([[148,
7     80], [437, 114],
8     [94, 247], [423,
9     288]])
10 # 变换后分别在左上、
11   右上、左下、右下四个
12   点
13 pts2 =
14   np.float32([[0,
15     0], [320, 0], [0,
16     178], [320, 178]])
17
18 # 生成透视变换矩阵
19 M =
20   cv2.getPerspectiveTransform(pts1,
21     pts2)
22
23 # 进行透视变换，参
24   数 3 是目标图像大小
25 dst =
26   cv2.warpPerspective(
27     img, M, (320,
28     178))
29
30 plt.subplot(121),
31 plt.imshow(img[:, :, ::-1]),
32 plt.title('input')
```

- 在不知道用什么滤波器好的时候，优先高斯滤波

```
cv2.GaussianBlur()
```

，然后均值滤波

```
cv2.blur()。
```

- 斑点和椒盐噪声优先使用中值滤波

```
cv2.medianBlur()。
```

- 要去除噪点的同时尽可能保留更多的边缘信息，使用双边滤波

```
cv2.bilateralFilter()。
```

- 线性滤波方式：均值滤波、方框滤波、高斯滤波（速度相对快）。

- 非线性滤波方式：中值滤波、双边滤波（速度相对慢）。

nts(rect)) # 矩形的四个角点取整

```
6 cv2.drawContours(i
7 mg_color1, [box],
8 0, (255, 0, 0), 2)
```

最小外接圆

```
1 (x, y), radius =
2   cv2.minEnclosingCir
3 cle(cnt)
```

```
2 (x, y, radius) =
3   np.int0((x, y,
4     radius)) # 圆心和半径取整
```

```
3 cv2.circle(img_col
4 or2, (x, y),
5     radius, (0, 0,
6     255), 2)
```



拟合椭圆

```
1 ellipse =
2   cv2.fitEllipse(cnt
3   )
```

```
2 cv2.ellipse(img_co
3 lor2, ellipse,
4   (255, 255, 0), 2)
```

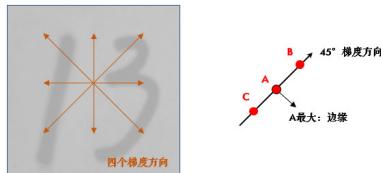
形状匹配

```
1 img =
2   cv2.imread('shapes
3 .jpg', 0)
```

```

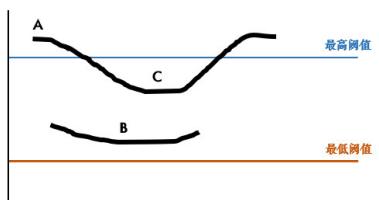
14 plt.subplot(122),
    plt.imshow(dst[:, :, ::-1]),
    plt.title('output')
)
15 plt.show()

```



比如，A点在 45° 方向上大于B/C点，那就保留它，把B/C设置为0。

4. 滞后阈值：经过前面三步，就只剩下0和可能的边缘梯度值了，为了最终确定下来，需要设定高低阈值：



- 像素点的值大于最高阈值，那肯定是边缘（上图A）
- 同理像素值小于最低阈值，那肯定不是边缘
- 像素值介于两者之间，如果与高于最高阈值的点连接，也算边缘，所以上图中C算，B不算

Canny 推荐的高低阈值比在 2:1 到 3:1 之间。

先阈值分割后检测

其实很多情况下，阈值分割后再检测边缘，效果会更好：

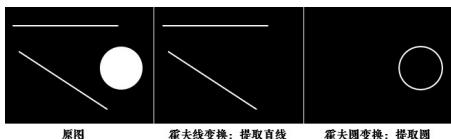
```

1 _, thresh =
    cv2.threshold(img,
        0, 255,
        cv2.THRESH_BINARY
        + cv2.THRESH_OTSU)
2 edges =
    cv2.Canny(thresh,
        30, 70)

```

霍夫变换

霍夫变换常用来在图像中提取直线和圆等几何形状，我来做个简易的解释：



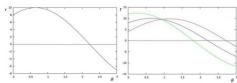
学过几何的都知道，直线可以分别用直角坐标系和极坐标系来表示：



那么经过某个点(x0,y0)的所有直线都可以用这个式子来表示：

$$r_0 = x_0 \cdot \cos \theta + y_0 \cdot \sin \theta$$

也就是说每一个(r,θ)都表示一条经过(x0,y0)直线。那么同一条直线上的点必然会有同样的(r,θ)。如果将某个点所有的(r,θ)绘制成下面的曲线，那么同一条直线上的点的(r,θ)曲线会相交于一点：



OpenCV 中首先计算(r,θ)累加数，累加数超过一定值后则认为在同一直线上。

霍夫直线变换

OpenCV 中用

`cv2.HoughLines()` 在二值图上实现霍夫变换，函数返回的是一组直线的(r,θ)数据

```

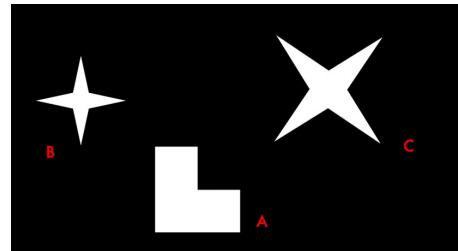
1 import cv2
2 import numpy as np
3
4 # 1. 加载图片，转为二
    值图
5 img =
    cv2.imread('shapes
        .jpg')
6 drawing =
    np.zeros(img.shape
        [:],
        dtype=np.uint8)

```

```

2 _, thresh =
    cv2.threshold(img,
        0, 255,
        cv2.THRESH_BINARY
        + cv2.THRESH_OTSU)
3 image, contours,
    hierarchy =
    cv2.findContours(t
        hresh, 3, 2)
4 img_color =
    cv2.cvtColor(thres
        h,
        cv2.COLOR_GRAY2BGR
    ) # 用于绘制的彩色
    图

```



直方图

直方图就是图像中每个像素值的个数统计，比如说一幅灰度图中像素值为0的有多少个，1的有多少个……



在计算直方图之前，有几个术语先来了解一下：

- dims:** 要计算的通道数，对于灰度图 dims=1，普通彩图 dims=3
- range:** 要计算的像素值范围，一般为[0,256)

```

7 gray =
    cv2.cvtColor(img,
    cv2.COLOR_BGR2GRAY
)
8 edges =
    cv2.Canny(gray,
    50, 150)
9
10 # 2. 霍夫直线变换
11 lines =
    cv2.HoughLines(edges, 0.8, np.pi /
    180, 90)

```

函数中：

- 参数 1：要检测的二值图（一般是阈值分割或边缘检测后的图）
- 参数 2：距离 r 的精度，值越大，考虑越多的线
- 参数 3：角度 θ 的精度，值越小，考虑越多的线
- 参数 4：累加数阈值，值越小，考虑越多的线



统计概率霍夫直线变换

前面的方法又称为标准霍夫变换，它会计算图像中的每一个点，计算量比较大，另外它得到的是整一条线 (r 和 θ)，并不知道原图中直线的端点。所以提出了统计概率霍夫直线变换(Probabilistic Hough Transform)，是一种改进的霍夫变换

```

3
4 cv2.imshow('canny',
    , np.hstack((img,
    thresh, edges)))
5 cv2.waitKey(0)

```



模板匹配

模板匹配就是用来在大图中找小图，也就是说在一副图像中寻找另外一张模板图像的位置用 `cv2.matchTemplate()` 实现模板匹配

```

1 # 相关系数匹配方法:
cv2.TM_CCOEFF
2 res =
cv2.matchTemplate(
img, template,
cv2.TM_CCOEFF)
3 min_val, max_val,
min_loc, max_loc
=
cv2.minMaxLoc(res)
4
5 left_top =
max_loc # 左上角
6 right_bottom =
(left_top[0] + w,
left_top[1] + h)
# 右下角
7 cv2.rectangle(img,
left_top,
right_bottom,
255, 2) # 画出矩形
位置

```

- `bins`: 子区段数目，如果我们统计 $0 \sim 255$ 每个像素值，`bins=256`；如果划分区间，比如 $0 \sim 15, 16 \sim 31 \dots 240 \sim 255$ 这样 16 个区间，`bins=16`

OpenCV 中直方图计算

使用

```
cv2.calcHist(images,
channels, mask,
histSize, ranges)
```

计算，其中：

- 参数 1：要计算的原图，以方括号的传入，如：`[img]`
- 参数 2：类似前面提到的 `dims`，灰度图写 `[0]` 就行，彩色图 B/G/R 分别传入 `[0]/[1]/[2]`
- 参数 3：要计算的区域，计算整幅图的话，写 `None`
- 参数 4：前面提到的 `bins`
- 参数 5：前面提到的 `range`

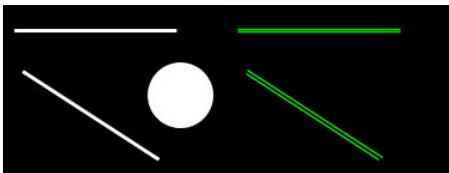
Numpy 中直方图计算

其中 `ravel()` 函数将二维矩阵展平变成一维数组

```
1 hist, bins =
np.histogram(img.r
avel(), 256, [0,
256]) # 性能:
0.020628 s
```

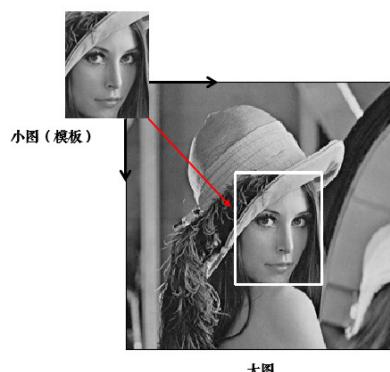
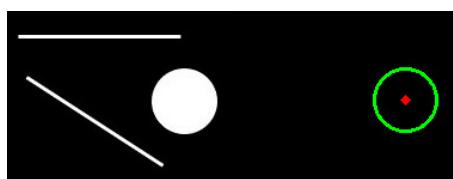
经验之谈：Numpy 中还有一种更高效的方式：

```
1 hist =
np.bincount(img.ra
```



霍夫圆变换

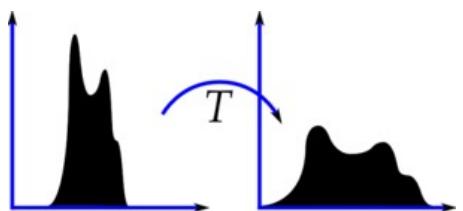
霍夫圆变换跟直线变换类似，只不过线是用 (r, θ) 表示，圆是用 $(x_{center}, y_{center}, r)$ 来表示，从二维变成了三维，数据量变大了很多；所以一般使用霍夫梯度法减少计算量



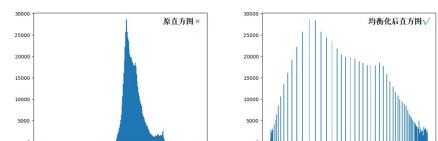
```
vel(),
minlength=256) #
性能: 0.003163 s
```

直方图均值化

一幅效果好的图像通常在直方图上的分布比较均匀，直方图均衡化就是用来改善图像的全局亮度和对比度。其实从观感上就可以发现，前面那幅图对对比度不高，偏灰白



OpenCV 中用 `cv2.equalizeHist()` 实现均衡化



```
1 import cv2
2 import numpy as np
3
4 img =
5 cv2.imread('lena.jpg')
6 # 此处需注意，请参考
7 # 后面的解释
8 res =
9 np.uint8(np.clip((1.5 * img + 10),
10 0, 255))
11 tmp =
12 np.hstack((img,
13 res)) # 两张图片横向合并(便于对比显示)
14
15 cv2.imshow('image',
16 , tmp)
17 cv2.waitKey(0)
```



自适应均衡化

自适应均衡化就是用来解决这一问题的：它在每一个小区域内（默认 8×8 ）进行直方图均衡化。当然，如果有噪点的话，噪点会被放大，需要对小区域内的对比度进行了限制，所以这个算法全称叫：**对比度受限的自适应直方图均衡化 CLAHE**

```
1 # 自适应均衡化，参数  
    可选  
2 clahe =  
    cv2.createCLAHE(cl  
    ipLimit=2.0,  
    tileGridSize=(8,  
    8))  
3 cl1 =  
    clahe.apply(img)
```

