

OpenCV

图像阈值处理

```
ret, dst = cv2.threshold(src, thresh, maxval, type)
• src: 输入图, 只能输入单通道图像, 通常来说为灰度图
• dst: 输出图
• thresh: 阈值
• maxval: 当像素值超过了阈值(或者小于阈值, 根据type来决定), 所赋予的值
• type: 二值化操作的类型, 包含以下5种类型: cv2.THRESH_BINARY; cv2.THRESH_BINARY_INV; cv2.THRESH_TRUNC; cv2.THRESH_TOZERO; cv2.THRESH_TOZERO_INV
• cv2.THRESH_BINARY 超过阈值部分取maxval(最大值), 否则取0
• cv2.THRESH_BINARY_INV THRESH_BINARY的反转
• cv2.THRESH_TRUNC 大于阈值部分设为阈值, 否则不变
• cv2.THRESH_TOZERO 大于阈值部分不改变, 否则设为0
• cv2.THRESH_TOZERO_INV THRESH_TOZERO的反转
```

cv2.THRESH_BINARY 超过阈值部分取maxval(最大值), 否则取0

该方法具有两个返回参数, 第一个参数表示成功与否

TRUNC表示截断

图像平滑

```
blur = cv2.blur(img, (3,3)) # 均值滤波
box = cv2.boxFilter(img, -1, (3,3), normalize=True) # 方框
滤波
# 高斯滤波, 离处理点越近的权重越大, 相当于加权平均
guass = cv2.GaussianBlur(img, (5, 5), 1)
median = cv2.medianBlur(img, 5) # 中值滤波
```

形态学-腐蚀操作

- 通常是二值的

```
kernel = np.ones((5, 5), np.uint8)
erosion = cv2.erode(img, kernel, iteration=2) # 腐蚀
dilate = cv2.dilate(img, kernel, iteration=1) # 膨胀
```

开运算与闭运算

```

opening = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
gradient = cv2.morphologyEx(img, cv2.MORPH_GRADIENT,
kernel)

```

礼貌与黑帽

礼貌 = 原始输入 - 开运算结果

黑帽 = 闭运算 - 原始输入

```
tophat = cv2.morphologyEx(img, cv2.MORPH_TOPHAT, kernel)
```

```
blackhat = cv2.morphologyEx(img, cv2.MORPH_BLACKHAT,
kernel)
```

图像梯度-Sobel算子

图像梯度-Sobel算子

$$\boxed{\mathbf{G}_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \boxed{\mathbf{G}_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

```
dst = cv2.Sobel(src, ddepth, dx, dy, ksize)
```

- ddepth: 图像的深度
- dx和dy分别表示水平和竖直方向
- ksize是Sobel算子的大小

```
In [3]: def cv_show(img, name):
    cv2.imshow(name, img)
    cv2.waitKey()
    cv2.destroyAllWindows()
```

```
In [4]: sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)
cv_show(sobelx, 'sobelx')
```

白到黑是整数，黑到白就是负数了，所有的负数会被截断成0，所以要取绝对值

```
[5]: sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)
sobelx = cv2.convertScaleAbs(sobelx)
cv_show(sobelx, 'sobelx')
```

```
[6]: sobely = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)
sobely = cv2.convertScaleAbs(sobely)
cv_show(sobely, 'sobely')
```

分别计算x和y，再求和

```
[7]: sobelxy = cv2.addWeighted(sobelx, 0.5, sobely, 0.5, 0)
cv_show(sobelxy, 'sobelxy')
```

图像梯度-Scharr算子

$$\mathbf{G}_x = \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix} * \mathbf{A}$$

图像梯度-laplacian算子

$$\mathbf{G} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Canny边缘检测

Canny边缘检测

- 1) 使用高斯滤波器，以平滑图像，滤除噪声。
- 2) 计算图像中每个像素点的梯度强度和方向。
- 3) 应用非极大值 (Non-Maximum Suppression) 抑制，以消除边缘检测带来的杂散响应。
- 4) 应用双阈值 (Double-Threshold) 检测来确定真实的和潜在的边缘。
- 5) 通过抑制孤立的弱边缘最终完成边缘检测。

$$H = \begin{bmatrix} 0.0924 & 0.1192 & 0.0924 \\ 0.1192 & 0.1538 & 0.1192 \\ 0.0924 & 0.1192 & 0.0924 \end{bmatrix} \quad \text{--这里还进行是归一化处理}$$

NSM将最明显的提取出来

$$e = H * A = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} * \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \text{sum} \left(\begin{bmatrix} a \times h_{11} & b \times h_{12} & c \times h_{13} \\ d \times h_{21} & e \times h_{22} & f \times h_{23} \\ g \times h_{31} & h \times h_{32} & i \times h_{33} \end{bmatrix} \right)$$

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \arctan(G_y / G_x)$$

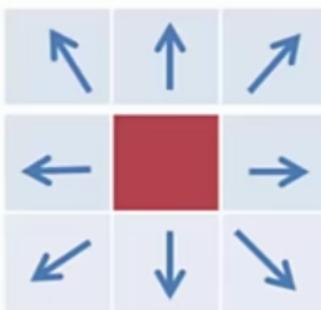
$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$G_x = S_x * A = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \text{sum} \left(\begin{bmatrix} -a & 0 & c \\ -2d & 0 & 2f \\ -g & 0 & i \end{bmatrix} \right).$$

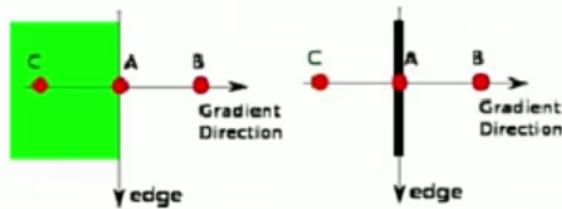
$$G_x = S_x * A = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \text{sum} \left(\begin{bmatrix} -a & 0 & c \\ -2d & 0 & 2f \\ -g & 0 & i \end{bmatrix} \right).$$

$$G_y = S_y * A = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \text{sum} \left(\begin{bmatrix} a & 2b & c \\ 0 & 0 & 0 \\ -g & -2h & -i \end{bmatrix} \right).$$

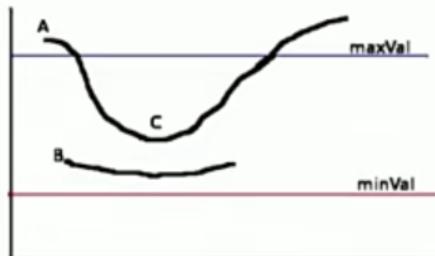




为了简化计算，由于一个像素周围有八个像素，把一个像素的梯度方向离散为八个方向，这样就只需计算前后即可，不用插值了。



4: 双阈值检测



梯度值>maxVal: 则处理为边界

minVal<梯度值<maxVal: 连有边界则保留，否则舍弃

梯度值<minVal: 则舍弃

```
15]: img=cv2.imread("lena.jpg",cv2.IMREAD_GRAYSCALE)
v1=cv2.Canny(img,80,150)
v2=cv2.Canny(img,50,100)

res = np.hstack((v1,v2))
cv_show(res,'res')
```

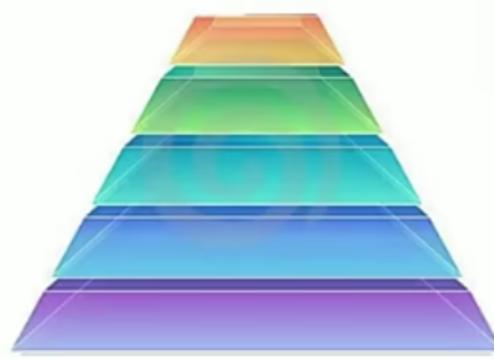
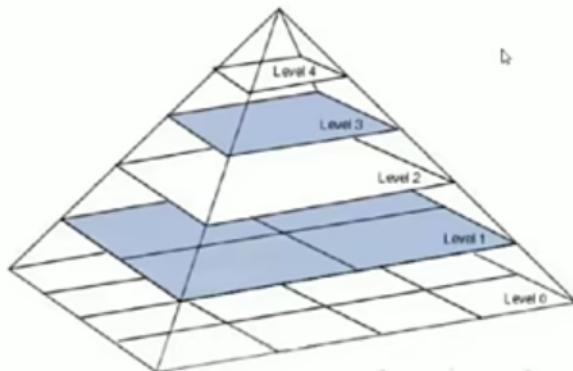
```
18]: img=cv2.imread("car.png",cv2.IMREAD_GRAYSCALE)
v1=cv2.Canny(img,120,250)
v2=cv2.Canny(img,50,100)

res = np.hstack((v1,v2))
cv_show(res,'res')
```

图像金字塔

图像金字塔

- 高斯金字塔
- 拉普拉斯金字塔



- 向下采样(缩小)

- $$\frac{1}{16} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$
 - 将 G_i 与高斯内核卷积
 - 将所有偶数行和列去除。

- 高斯金字塔：向上采样方法（放大）

- 将图像在每个方向扩大为原来的两倍，新增的行和列以0填充
- 使用先前同样的内核(乘以4)与放大后的图像卷积，获得近似值

```
In [42]: img=cv2.imread("AM.png")
cv_show(img,'img')
print (img.shape)
```

(442, 340, 3)

```
In [36]: up=cv2.pyrUp(img)
cv_show(up,'up')
print (up.shape)
```

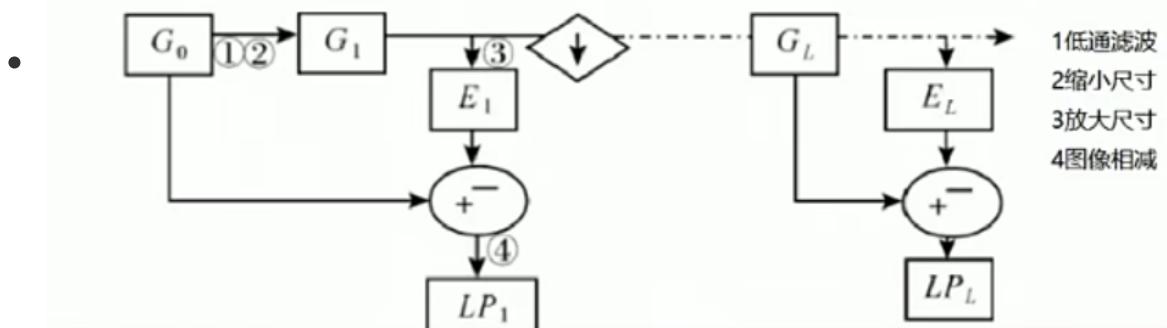
(884, 680, 3)

```
In [37]: down=cv2.pyrDown(img)
cv_show(down,'down')
print (down.shape)
```

(221, 170, 3)

拉普拉斯金字塔

$$L_i = G_i - \text{PyrUp}(\text{PyrDown}(G_i))$$



- 拉普拉斯金字塔

图像轮廓

```
cv2.findContours(img, mode, method)
mode:轮廓检测模式
    • RETR_EXTERNAL : 只检索最外面的轮廓;
    • RETR_LIST: 检索所有的轮廓，并将其保存到一条链表当中;
    • RETR_CCOMP: 检索所有的轮廓，并将他们组织为两层：顶层是各部分的外部边界，第二层是空洞的边界;
    • RETR_TREE: 检索所有的轮廓，并重构嵌套轮廓的整个层次;
method:轮廓逼近方法
    • CHAIN_APPROX_NONE: 以Freeman链码的方式输出轮廓，所有其他方法输出多边形（顶点的序列）。
    • CHAIN_APPROX_SIMPLE:压缩水平的、垂直的和斜的部分，也就是，函数只保留他们的终点部分。
```

- 边缘零零散散的，轮廓是一个整体

- `cv2.findContours(img, mode, method)`



- 为了更高的准确率，使用二值图像。

```
: img = cv2.imread('contours.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)
cv_show(thresh, 'thresh')

: binary, contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
```

- 绘制轮廓

```
In [6]: #传入绘制图像，轮廓，轮廓索引，颜色模式，线条厚度
#注意需要copy，不然图会变...
draw_img = img.copy()
res = cv2.drawContours(draw_img, contours, -1, (0, 0, 255), 2)
cv_show(res, 'res')
```

```
In [7]: draw_img = img.copy()
res = cv2.drawContours(draw_img, contours, 0, (0, 0, 255), 2)
cv_show(res, 'res')
```

- 1 表示绘制所有轮廓，会对原始图像直接进行变化，可以先使用copy()进行深层复制

- 轮廓特征

```
In [8]: cnt = contours[0]
```

```
In [9]: #面积
cv2.contourArea(cnt)
```

```
Out[9]: 8500.5
```

```
In [10]: #周长。True表示闭合的  
cv2.arcLength(cnt, True)
```

```
Out[10]: 437.9482651948929
```

- cv2.contourArea(cnt)
- cv2.arcLength(cnt)

轮廓近似



- 用直线代替曲线
- 不断的使用直线去划分和代替

```
i]: img = cv2.imread('contours2.png')

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)
binary, contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
cnt = contours[0]

draw_img = img.copy()
res = cv2.drawContours(draw_img, [cnt], -1, (0, 0, 255), 2)
cv_show(res, 'res')
```

```
j]: epsilon = 0.1*cv2.arcLength(cnt, True)
approx = cv2.approxPolyDP(cnt, epsilon, True)

draw_img = img.copy()
res = cv2.drawContours(draw_img, [approx], -1, (0, 0, 255), 2)
cv_show(res, 'res')
```

- 传入轮廓，和一个阈值（一般是按照周长的百分比来计算的）
- 阈值指定的越小，近似的就越接近

边界矩形

```
In [105]: img = cv2.imread('contours.png')

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)
binary, contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
cnt = contours[0]

x,y,w,h = cv2.boundingRect(cnt)[1]
img = cv2.rectangle(img, (x,y), (x+w,y+h), (0,255,0), 2)
cv_show(img, 'img')
```

```
In [106]: area = cv2.contourArea(cnt)
x, y, w, h = cv2.boundingRect(cnt)
rect_area = w * h
extent = float(area) / rect_area
print ('轮廓面积与边界矩形比', extent)

Out[106]: 0.5154317244724715
```

外接圆

```
In [112]: (x,y),radius = cv2.minEnclosingCircle(cnt)
center = (int(x),int(y))
radius = int(radius)
img = cv2.circle(img,center,radius,(0,255,0),2)
cv_show(img,'img')
```

模板匹配

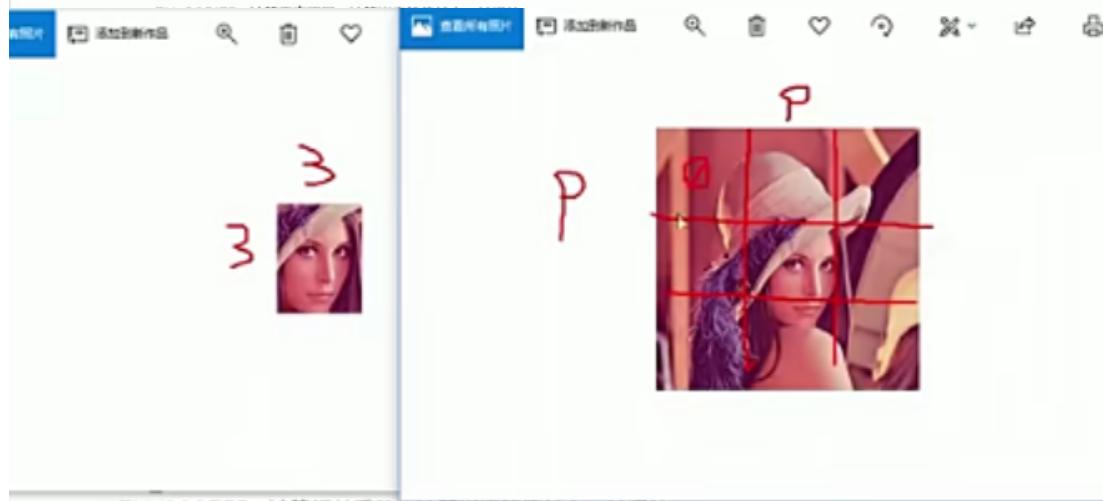
模板匹配

模板匹配和卷积原理很像，模板在原图像上从原点开始滑动，计算模板与（图像被模板覆盖的地方）的差别程度，这个差别程度的计算方法在opencv里有6种，然后将每次计算的结果放入一个矩阵里，作为结果输出。假如原图形是AxB大小，而模板是axb大小，则输出结果的矩阵是(A-a+1)x(B-b+1)

```
In [3]: # 模板匹配
img = cv2.imread('lena.jpg', 0)
template = cv2.imread('face.jpg', 0)
h, w = template.shape[:2]
```

```
In [12]: img.shape
Out[12]: (263, 263)
```

```
In [11]: template.shape
Out[11]: (110, 85)
```



- 对应位置挨个进行比较

```
In [11]: template.shape
```

```
Out[11]: (110, 85)
```

- TM_SQDIFF: 计算平方不同，计算出来的值越小，越相关
- TM_CCORR: 计算相关性，计算出来的值越大，越相关
- TM_CCOEFF: 计算相关系数，计算出来的值越大，越相关
- TM_SQDIFF_NORMED: 计算归一化平方不同，计算出来的值越接近0，越相关
- TM_CCORR_NORMED: 计算归一化相关性，计算出来的值越接近1，越相关
- TM_CCOEFF_NORMED: 计算归一化相关系数，计算出来的值越接近1，越相关

公式：https://docs.opencv.org/3.3.1/d7/dbd/group__imgproc__object.html#gaa3a7850640f1fe1f58fe91a2d7583695d

```
In [4]: methods = ['cv2.TM_CCOEFF', 'cv2.TM_CCOEFF_NORMED', 'cv2.TM_CCORR',
               'cv2.TM_CCORR_NORMED', 'cv2.TM_SQDIFF', 'cv2.TM_SQDIFF_NORMED']
```

```
In [10]: res = cv2.matchTemplate(img, template, 1)
res.shape
```

```
Out[10]: (154, 179)
```

```
In [23]: res = cv2.matchTemplate(img, template, cv2.TM_SQDIFF)
res.shape

Out[23]: (154, 179)

In [24]: min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)

In [25]: min_val

Out[25]: 39168.0

• In [26]: max_val

Out[26]: 74403584.0

In [27]: min_loc

Out[27]: (107, 89)

In [28]: max_loc

Out[28]: (159, 62)
```

- 使用归一化的结果相对会更加公平一些

```
In [8]: for meth in methods:
    img2 = img.copy()

    # 匹配方法的真值
    method = eval(meth)
    print (method)
    res = cv2.matchTemplate(img, template, method)
    min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)

    # 如果是平方差匹配TM_SQDIFF或归一化平方差匹配TM_SQDIFF_NORMED, 取最小值
    if method in [cv2.TM_SQDIFF, cv2.TM_SQDIFF_NORMED]:
        top_left = min_loc
    else:
        top_left = max_loc
    bottom_right = (top_left[0] + w, top_left[1] + h)

    # 画矩形
    cv2.rectangle(img2, top_left, bottom_right, 255, 2)

    plt.subplot(121), plt.imshow(res, cmap='gray')
    plt.xticks([]), plt.yticks([]) # 隐藏坐标轴
    plt.subplot(122), plt.imshow(img2, cmap='gray')
    plt.xticks([]), plt.yticks([])
    plt.suptitle(meth)
    plt.show()
```

匹配多个对象

```
In [17]: img_rgb = cv2.imread('mario.jpg')
img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_BGR2GRAY)
template = cv2.imread('mario_coin.jpg', 0)
h, w = template.shape[:2]

res = cv2.matchTemplate(img_gray, template, cv2.TM_CCOEFF_NORMED)
threshold = 0.8
# 收匹配程度大于%80的坐标
loc = np.where(res >= threshold)
for pt in zip(*loc[::-1]): # *号表示可选参数
    bottom_right = (pt[0] + w, pt[1] + h)
    cv2.rectangle(img_rgb, pt, bottom_right, (0, 0, 255), 2)

cv2.imshow('img_rgb', img_rgb)
cv2.waitKey(0)
```

Out[17]: -1

直方图与傅里叶变换

直方图



- 对整个图中的像素点进行统计

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

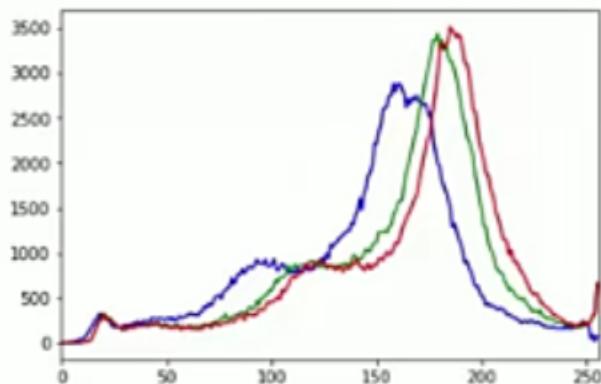
• images: 原图像图像格式为uint8 或 float32, 当传入函数时应用中括号[]括来例如[img]
• channels: 同样用中括号括来它会告诉函数我们统幅图像的直方图。如果原图像是灰度图它的值就是[0]如果是彩色图像的传入的参数可以是[0][1][2]它们分别对应着BGR。
• mask: 掩模图像。统整幅图像的直方图就把它为None。但是如果你想统图像某一分的直方图的你就制作一个掩模图像并使用它。
• histSize: BIN 的数目。也应用中括号括来
• ranges: 像素值范围常为 [0,256]
```

```
In [42]: img = cv2.imread('cat.jpg', 0) #0表示灰度图
hist = cv2.calcHist([img], [0], None, [256], [0, 256])
hist.shape
```

Out[42]: (256, 1)

```
In [43]: plt.hist(img.ravel(), 256)
plt.show()
```

```
In [14]: img = cv2.imread('cat.jpg')
color = ('b', 'g', 'r')
for i, col in enumerate(color):
    hist = cv2.calcHist([img], [i], None, [256], [0, 256])
    plt.plot(hist, color = col)
    plt.xlim([0, 256])
```



- mask掩码

mask操作

```
In [16]: # 创建mask
mask = np.zeros(img.shape[:2], np.uint8)
mask[100:300, 100:400] = 255
cv_show(mask, 'mask')
```

```
In [17]: img = cv2.imread('cat.jpg', 0)
cv_show(img, 'img')
```

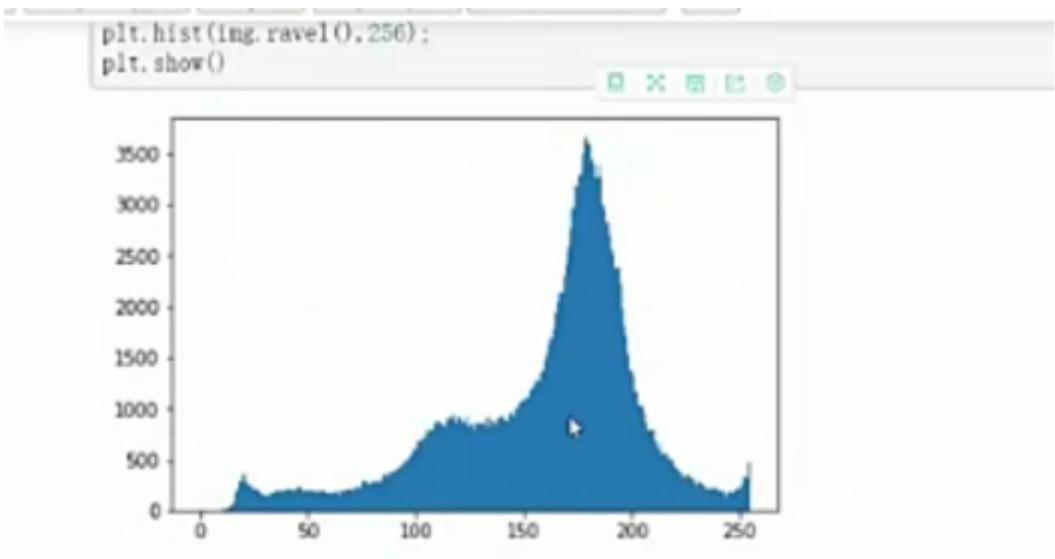
```
In [18]: masked_img = cv2.bitwise_and(img, img, mask=mask) #与&/
cv_show(masked_img, 'masked_img')
```

```
In [20]: hist_full = cv2.calcHist([img], [0], None, [256], [0, 256])
hist_mask = cv2.calcHist([img], [0], mask, [256], [0, 256])
```

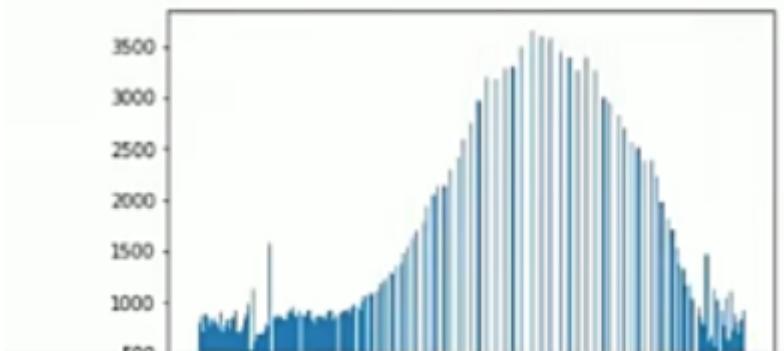
```
In [21]: plt.subplot(221), plt.imshow(img, 'gray')
plt.subplot(222), plt.imshow(mask, 'gray')
plt.subplot(223), plt.imshow(masked_img, 'gray')
plt.subplot(224), plt.plot(hist_full), plt.plot(hist_mask)
plt.xlim([0, 256])
plt.show()
```

- 直方图均衡化

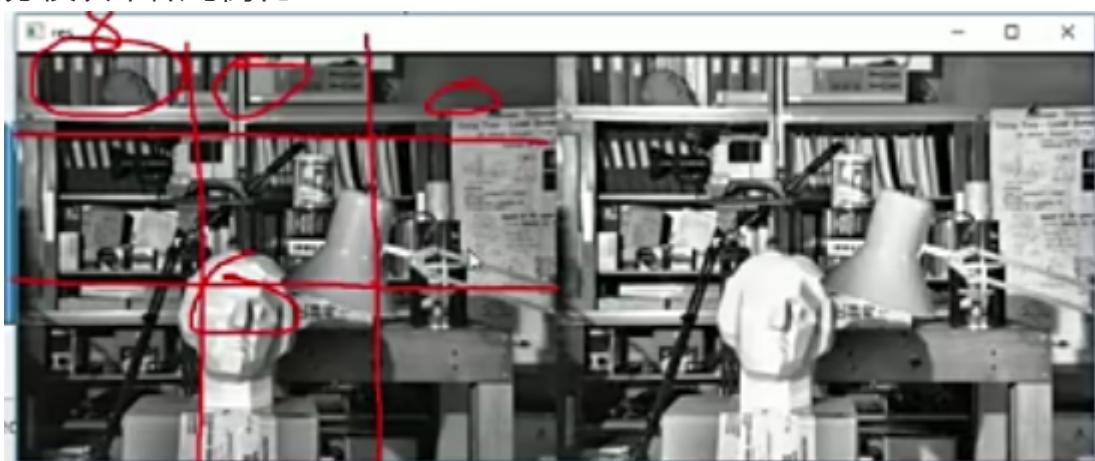




```
[77]: equ = cv2.equalizeHist(img)
plt.hist(equ.ravel(), 256)
plt.show()
```



- 分模块来做均衡化



- 自适应均衡化

自适应直方图均衡化

```
In [58]: clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
```

```
In [65]: res_clahe = clahe.apply(img)
res = np.hstack((img, equ, res_clahe))
cv_show(res, 'res')
```

傅里叶变换

傅里叶变换

我们生活在时间的世界中，早上7:00起来吃早饭，8:00去挤地铁，9:00开始上班。。以时间为参照就是时域分析。
但是在频域中一切都是静止的！

<https://zhuanlan.zhihu.com/p/19763358>

傅里叶变换的作用

- 高频：变化剧烈的灰度分量，例如边界
- 低频：变化缓慢的灰度分量，例如一片大海

滤波

- 低通滤波器：只保留低频，会使得图像模糊
 - 高通滤波器：只保留高频，会使得图像细节增强
-
- opencv中主要就是cv2.dft()和cv2.idft()，输入图像需要先转换成np.float32 格式。
 - 得到的结果中频率为0的部分会在左上角，通常要转换到中心位置，可以通过shift变换来实现。
 - cv2.dft()返回的结果是双通道的（实部，虚部），通常还需要转换成图像格式才能展示（0.255）。

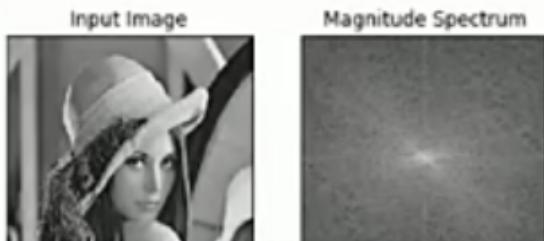
```
In [10]: import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('lena.jpg')  

img_float32 = np.float32(img)

dft = cv2.dft(img_float32, flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
# 得到灰度图能表示的形式
magnitude_spectrum = 20*np.log(cv2.magnitude(dft_shift[:, :, 0], dft_shift[:, :, 1]))

plt.subplot(121),plt.imshow(img, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(magnitude_spectrum, cmap = 'gray')
plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])
plt.show()
```



```
In [9]: import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('lena.jpg',0)

img_float32 = np.float32(img)

dft = cv2.dft(img_float32, flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)

rows, cols = img.shape
crow, ccol = int(rows/2), int(cols/2)      # 中心位置

# 低通滤波
mask = np.zeros((rows, cols, 2), np.uint8)
mask[crow-30:crow+30, ccol-30:ccol+30] = 1

# IDFT
fshift = dft_shift*mask
f_ishift = np.fft.ifftshift(fshift)
img_back = cv2.idft(f_ishift)
img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])

plt.subplot(121),plt.imshow(img, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(img_back, cmap = 'gray')
plt.title('Result'), plt.xticks([]), plt.yticks([])
```

```
In [11]: img = cv2.imread('lena.jpg',0)

img_float32 = np.float32(img)

dft = cv2.dft(img_float32, flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)

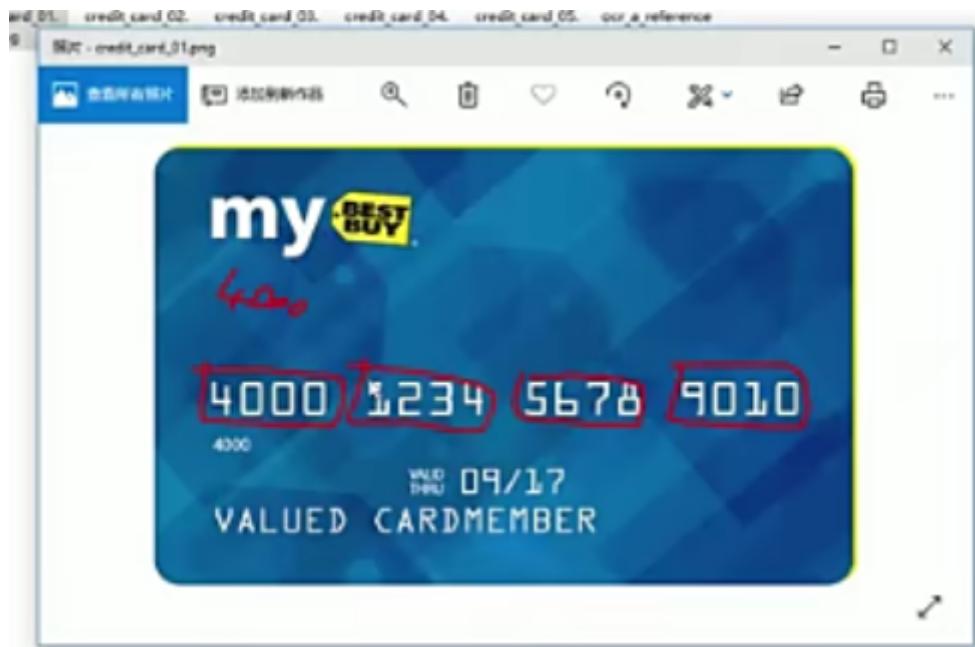
rows, cols = img.shape
crow, ccol = int(rows/2), int(cols/2)      # 中心位置

# 高通滤波
mask = np.ones((rows, cols, 2), np.uint8)
mask[crow-30:crow+30, ccol-30:ccol+30] = 0

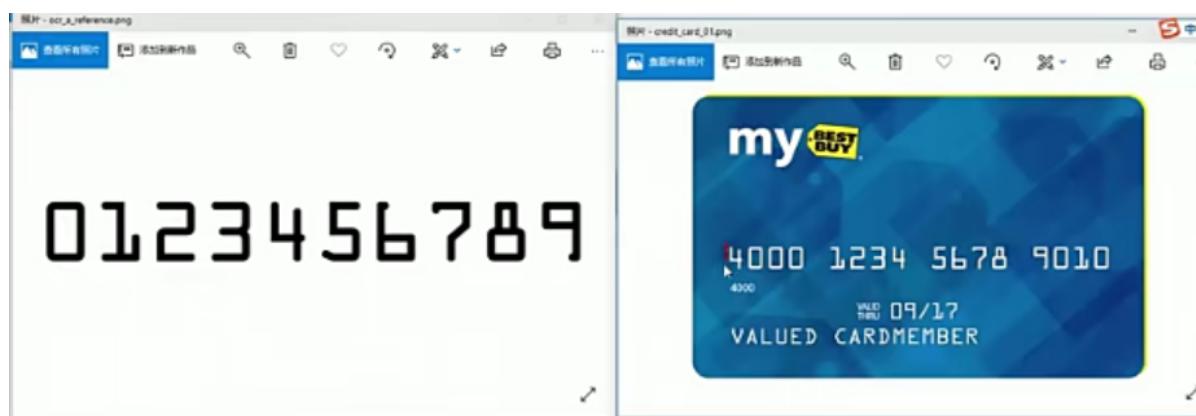
# IDFT
fshift = dft_shift*mask
f_ishift = np.fft.ifftshift(fshift)
img_back = cv2.idft(f_ishift)
img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])

plt.subplot(121),plt.imshow(img, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(img_back, cmap = 'gray')
plt.title('Result'), plt.xticks([]), plt.yticks([])
```

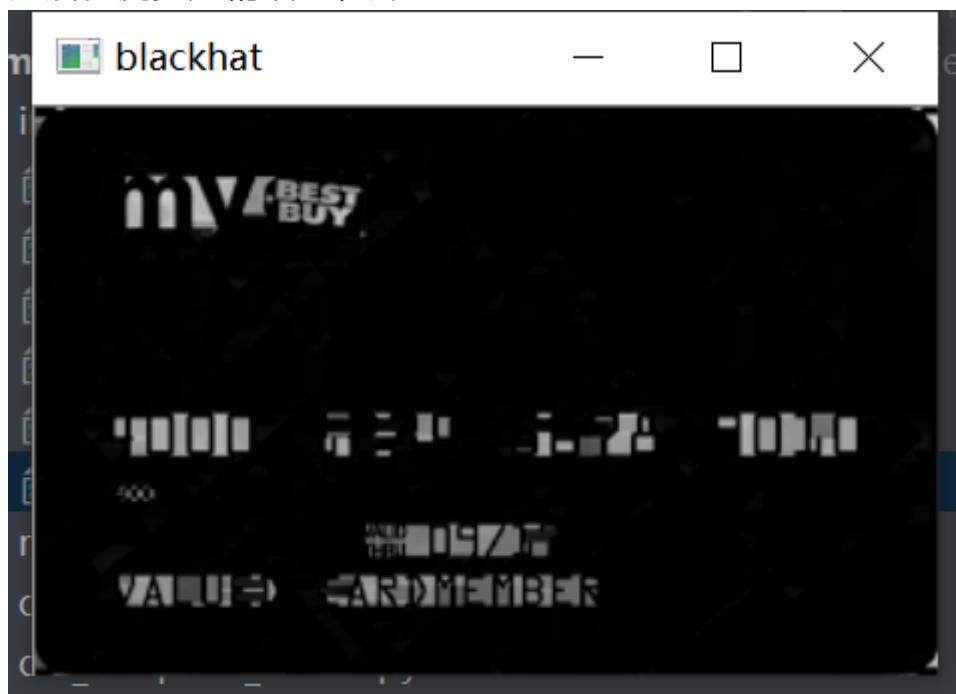
项目实战 - 信用卡号识别

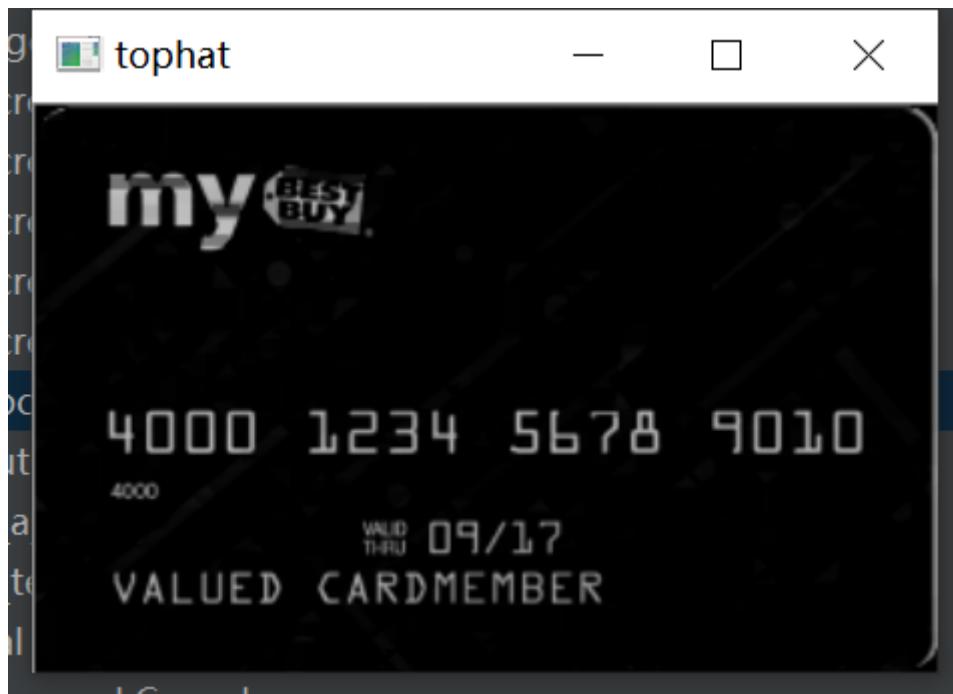


模板匹配



- 第一步检测大致的轮廓
- 然后检测更加精细的轮廓





图像特征

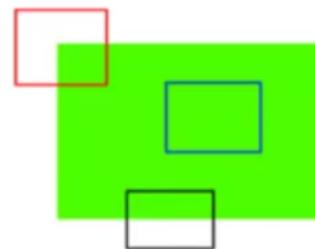
- harris角点检测



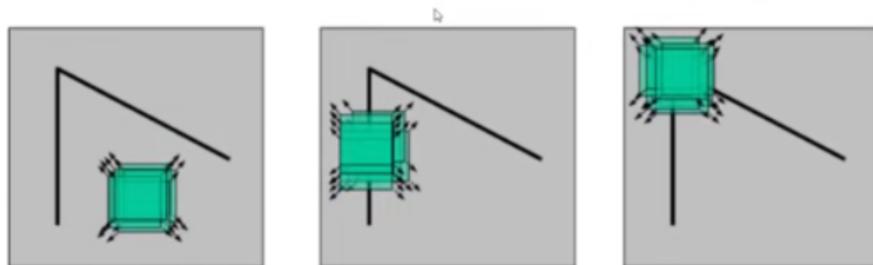
A 和 B 是平而且它们的图像中很多地方存在。

C 和 D 是边界，可以大致找到位置。

E 和 F 是角点，可以迅速定位到。



基本原理



对于图像 $I(x,y)$, 当在点 (x,y) 处平移 $(\Delta x, \Delta y)$ 后的自相似性:

对于图像 $I(x,y)$, 当在点 (x,y) 处平移 $(\Delta x, \Delta y)$ 后的自相似性:

$$c(x, y; \Delta x, \Delta y) = \sum_{(u,v) \in W(x,y)} w(u, v) (I(u, v) - I(u + \Delta x, v + \Delta y))^2$$

$W(x,y)$ 是以点 (x,y) 为中心的窗口, 既可是常数, 也可以是高斯加权函数:

- 比较的是对应位置的窗口的灰度值的情况
- u, v 是属于窗口当中的
- 把结果进行了扩大的操作

基于泰勒展开，对图像 $I(x, y)$ 在平移 $(\Delta x, \Delta y)$ 后进行一阶近似：

- $$\underline{I(u + \Delta x, v + \Delta y) = I(u, v) + I_x(u, v)\Delta x + I_y(u, v)\Delta y + O(\Delta x^2, \Delta y^2)} \approx I(u, v) + I_x(u, v)\Delta x + I_y(u, v)\Delta y$$

其中， I_x, I_y 是 $I(x, y)$ 的偏导数

近似可得：

$$c(x, y; \Delta x, \Delta y) \approx \sum_w (I_x(u, v)\Delta x + I_y(u, v)\Delta y)^2 = [\Delta x, \Delta y] M(x, y) \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

其中 M ：

$$M(x, y) = \sum_w \begin{bmatrix} I_x(x, y)^2 & I_x(x, y)I_y(x, y) \\ I_x(x, y)I_y(x, y) & I_y(x, y)^2 \end{bmatrix} = \begin{bmatrix} \sum_w I_x(x, y)^2 & \sum_w I_x(x, y)I_y(x, y) \\ \sum_w I_x(x, y)I_y(x, y) & \sum_w I_y(x, y)^2 \end{bmatrix} = \begin{bmatrix} A & C \\ C & B \end{bmatrix}$$

化简可得：

$$c(x, y; \Delta x, \Delta y) \approx A\Delta x^2 + 2C\Delta x\Delta y + B\Delta y^2$$

$$A = \sum_w I_x^2, B = \sum_w I_y^2, C = \sum_w I_x I_y$$

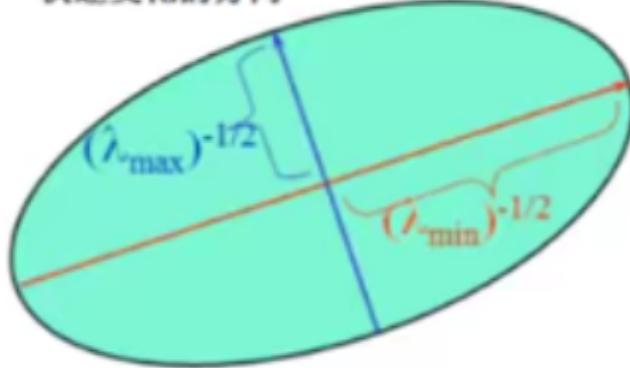
二次项函数本质上就是一个椭圆函数，椭圆方程为：

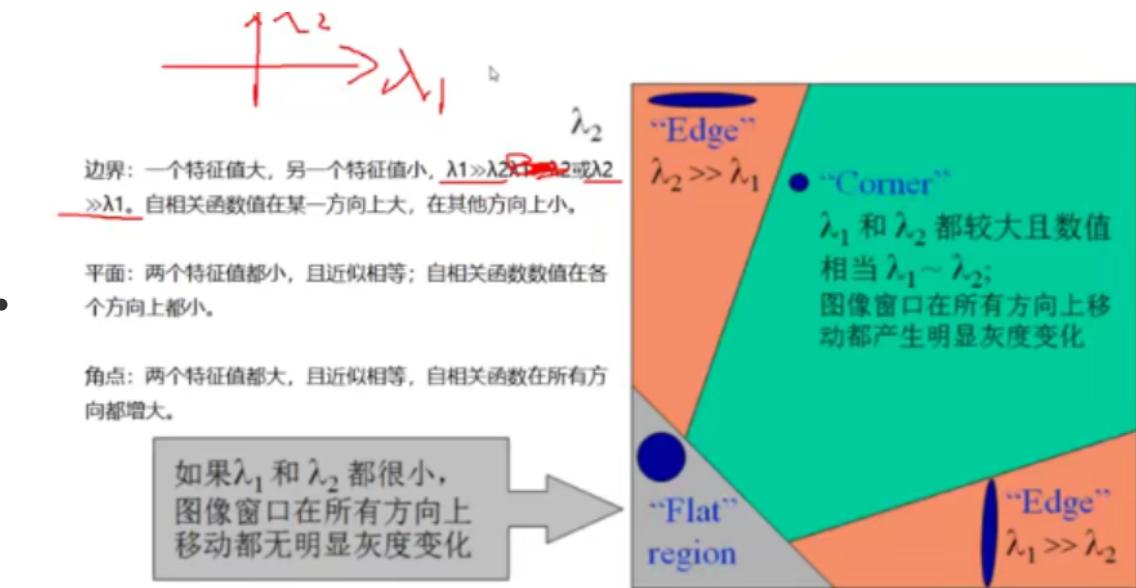
$$[\Delta x, \Delta y] M(x, y) \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = 1$$

- 实对称矩阵一定是可以对角化的

快速变化的方向

缓慢变化的方向





- 一个特征值大，一个特征值小，应该是边界
- 两个特征值都小，则是平面
- 两个特征值都大，则是角点

角点响应R值：

$$R = \det M - \alpha (\text{trace } M)^2$$

$$\det M = \lambda_1 \lambda_2 \quad \text{trace } M = \lambda_1 + \lambda_2$$

- $R > 0$: 角点

`cv2.cornerHarris()`

- img: 数据类型为 float32 的入图像
- blockSize: 角点检测中指定区域的大小
- ksize: Sobel 求导中使用的窗口大小
- k: 阈值参数为 [0.04, 0.06]



```
[23]: import cv2
import numpy as np

img = cv2.imread('chessboard.jpg')
print('img.shape:', img.shape)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
gray = np.float32(gray)
dst = cv2.cornerHarris(gray, 2, 3, 0.04)
print('dst.shape:', dst.shape)

img.shape: (512, 512, 3)
dst.shape: (512, 512)
```

```
[24]: img[dst>0.01*dst.max()]=[0,0,255]
cv2.imshow('dst',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

图像特征--sift

图像尺度空间

在一定的范围内，无论物体是大还是小，人眼都可以分辨出来，然而计算机要有相同的能力却很难，所以要让机器能够对物体在不同尺度下有一个统一的认识，就需要考虑图像在不同的尺度下都存在的特点。

尺度空间的获取通常使用高斯模糊来实现

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

其中，G是高斯函数

$$1 - e^{-\frac{x^2+y^2}{2\sigma^2}}$$

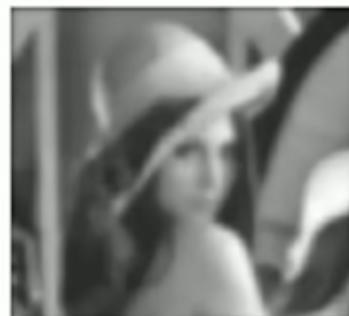
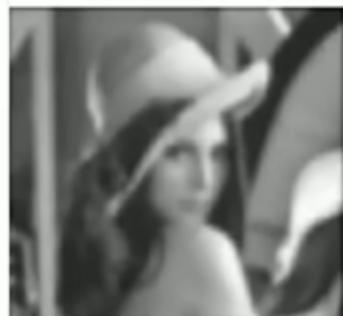
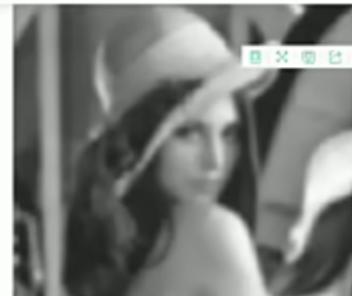
0.0000	0.0004	0.0133	0.0004	0.0000
06586	24781	0373	24781	06586
0.0004	0.0273	0.1098	0.0273	0.0004
24781	984	78	984	24781
0.0133	0.1098	0.4406	0.1098	0.0133
0373	78	55	78	0373

$$L(x, y, \sigma) = G(x, y, \sigma) * \underbrace{I(x, y)}_{l_c}$$

其中，G是高斯函数

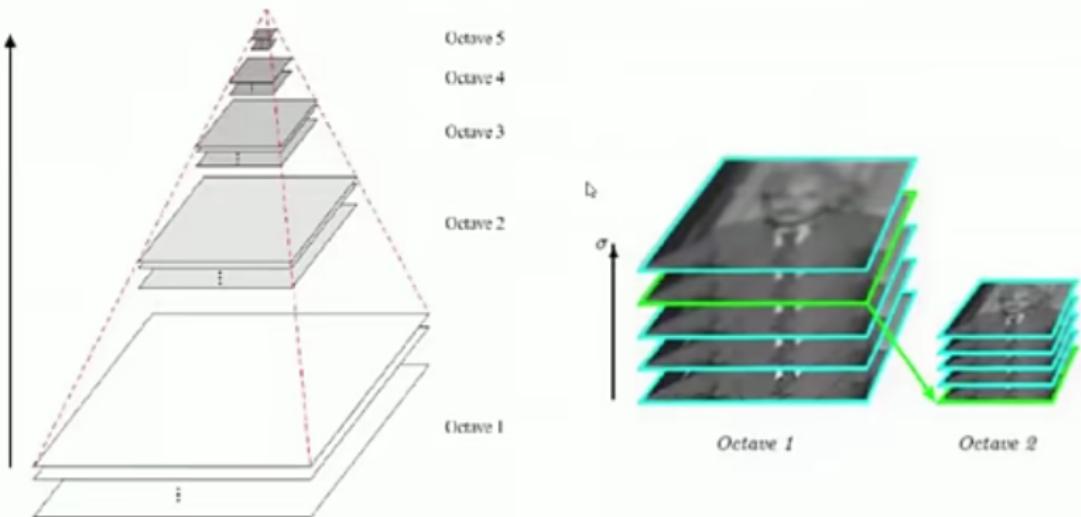
$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

0.0000	0.0004	0.0133	0.0004	0.0000
06586	24781	0373	24781	06586
0.0004	0.0273	0.1098	0.0273	0.0004
24781	984	78	984	24781
0.0133	0.1098	0.4406	0.1098	0.0133
0373	78	55	78	0373
0.0004	0.0273	0.1098	0.0273	0.0004
24781	984	78	984	24781
0.0000	0.0004	0.0133	0.0004	0.0000
06586	24781	0373	24781	06586

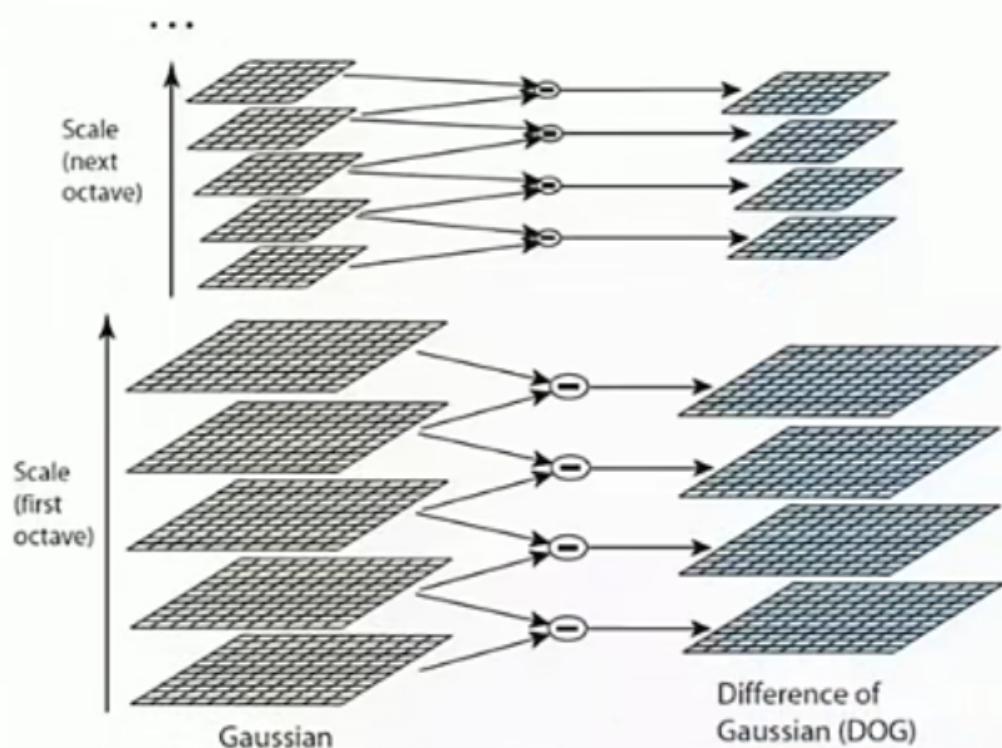


σ 决定图像模糊的程度

多分辨率金字塔



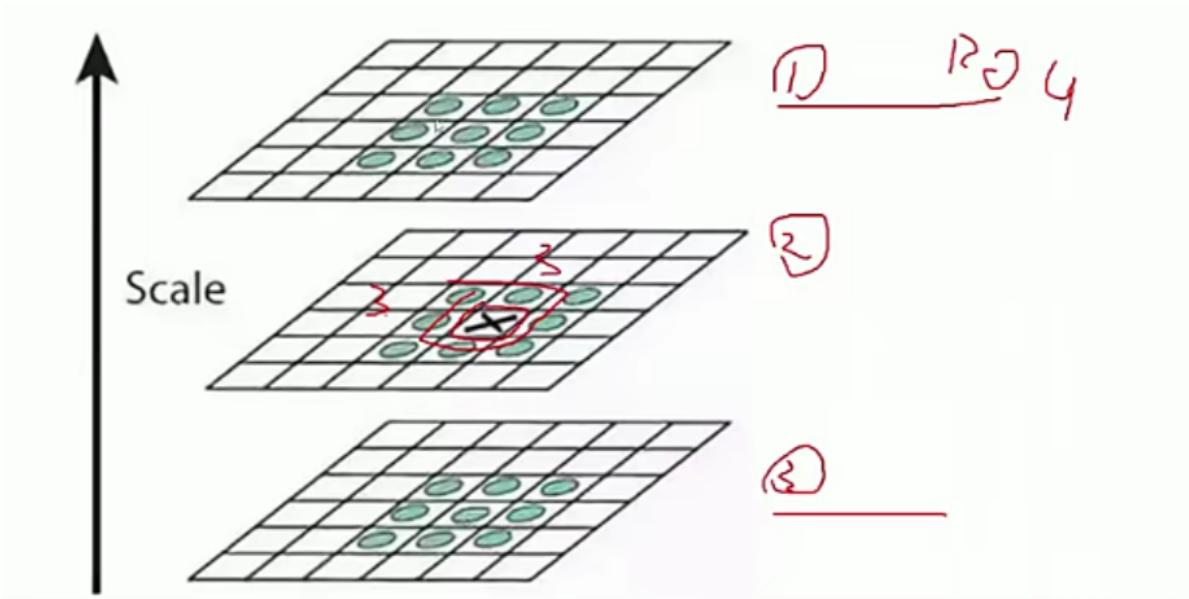
高斯差分金字塔 (DOG)



认为差别较大的地方的更加重要

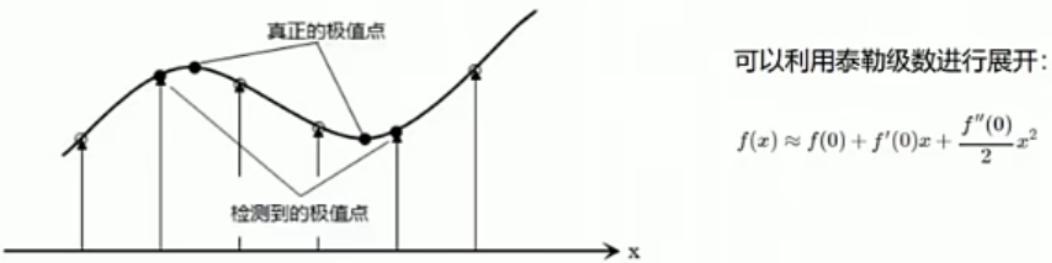
DOG定义公式：

$$D(x, y, \sigma) = [G(x, y, k\sigma) - G(x, y, \sigma)] * I(x, y) = L(x, y, k\sigma) - L(x, y, \sigma)$$



关键点的精确定位

这些候选关键点是DOG空间的局部极值点，而且这些极值点均为离散的点。精确定位极值点的一种方法是，对尺度空间DoG函数进行曲线拟合，计算其极值点，从而实现关键点的精确定位。



$$D(\Delta x, \Delta y, \Delta \sigma) = D(x, y, \sigma) + \left[\frac{\partial D}{\partial x} \quad \frac{\partial D}{\partial y} \quad \frac{\partial D}{\partial \sigma} \right] \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta \sigma \end{bmatrix} + \frac{1}{2} [\Delta x \quad \Delta y \quad \Delta \sigma] \begin{bmatrix} \frac{\partial^2 D}{\partial x^2} & \frac{\partial^2 D}{\partial x \partial y} & \frac{\partial^2 D}{\partial x \partial \sigma} \\ \frac{\partial^2 D}{\partial y \partial x} & \frac{\partial^2 D}{\partial y^2} & \frac{\partial^2 D}{\partial y \partial \sigma} \\ \frac{\partial^2 D}{\partial \sigma \partial x} & \frac{\partial^2 D}{\partial \sigma \partial y} & \frac{\partial^2 D}{\partial \sigma^2} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta \sigma \end{bmatrix}$$

$$D(x) = D + \frac{\partial D^T}{\partial x} \Delta x + \frac{1}{2} \Delta x^T \frac{\partial^2 D^T}{\partial x^2} \Delta x \quad \Delta x = - \frac{\partial^2 D^{-1}}{\partial x^2} \frac{\partial D(x)}{\partial x}$$

对极值的位置进行调优

消除边界响应

Hessian矩阵：

$$H(x, y) = \begin{bmatrix} D_{xx}(x, y) & D_{xy}(x, y) \\ D_{xy}(x, y) & D_{yy}(x, y) \end{bmatrix}$$

令 $\alpha = \lambda_{\max}$ 为最大的特征值， $\beta = \lambda_{\min}$ 为最小的特征值

$$\frac{\lambda}{\beta} \geq 10$$

$$Tr(H) = D_{xx} + D_{yy} = \alpha + \beta$$

$$Det(H) = D_{xx}D_{yy} - (D_{xy})^2 = \alpha\beta$$

$$\frac{Tr(H)^2}{Det(H)} = \frac{(\alpha + \beta)^2}{\alpha\beta} = \frac{(\gamma + 1)^2}{\gamma}$$

Lowe在论文中给出的 $y=10$ ，也就是说对于主曲率比值大于10的特征点将被删除。

特征点的主方向

每个点 $L(x,y)$ 的梯度的模 $m(x,y)$ 以及方向 $\theta(x,y)$:

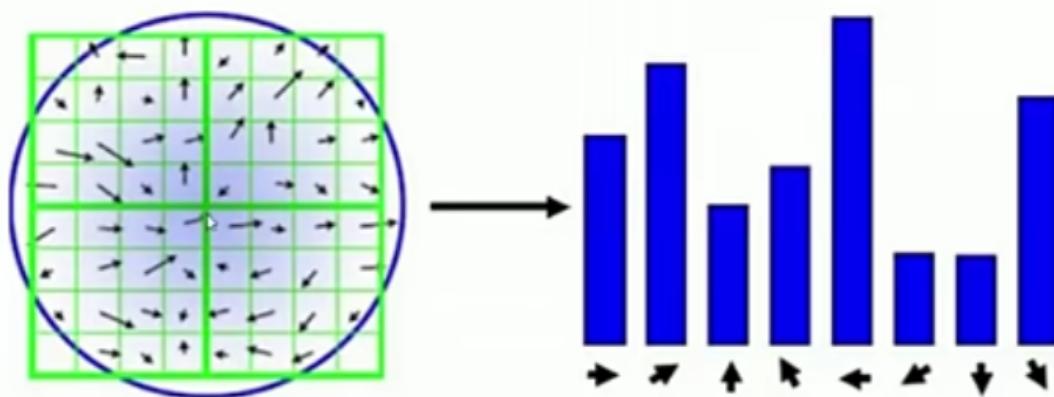
$$m(x,y) = \sqrt{[L(x+1,y) - L(x-1,y)]^2 + [L(x,y+1) - L(x,y-1)]^2}$$

$$\theta(x,y) = \arctan \frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}$$

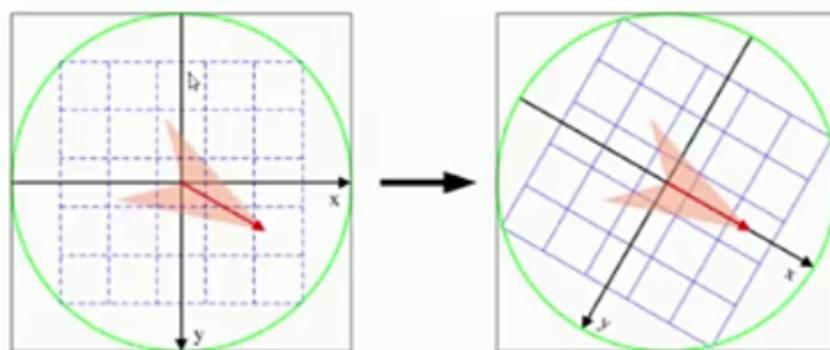
每个特征点可以得到三个信息 (x,y,m,θ) , 即位置、尺度和方向。具有多个方向的关键点可以被复制成多份, 然后将方向值分别赋给复制后的特征点, 一个特征点就产生了多个坐标、尺度相等, 但是方向不同的特征点。

生成特征描述

在完成关键点的梯度计算后, 使用直方图统计领域内像素的梯度和方向。



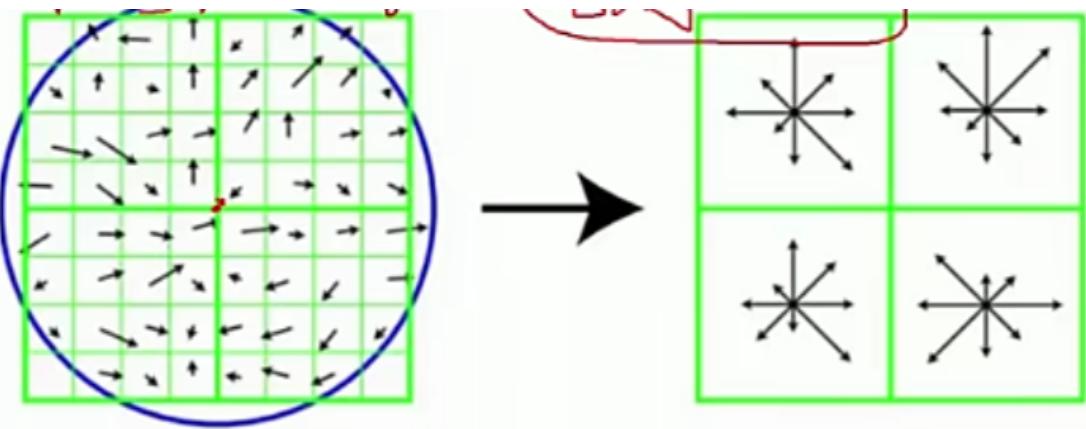
为了保证特征矢量的旋转不变性, 要以特征点为中心 在附近领域内将坐标轴旋转 θ 角度, 即将坐标轴旋转为特征点的主方向。



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

旋转之后的主方向为中心取 8×8 的窗口, 求每个像素的梯度幅值和方向, 前头方向代表梯度方向, 长度代表梯度幅值, 然后利用高斯窗口对其进行加权运算, 最后在每个 4×4 的小块上绘制8个方向的梯度直方图, 计算每个梯度方向的累加值, 即可形成一个种子点, 即每个特征的由4个种子点组成, 每个种子点有8个方向的向量信息。

旋转到主方向，保证特征的不变性



论文中建议对每个关键点使用 4×4 共16个种子点来描述。这样一个关键点就会产生128维的SIFT特征向量。

尺度空间 -> 极值点检测 -> 精确点 -> 消除边界响应 ->

opencv SIFT函数 3.4.1

```
In [1]: import cv2
import numpy as np

img = cv2.imread('test_1.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

In [2]: cv2.__version__ #3.4.1.15 pip install opencv-python==3.4.1.15 pip install opencv-contrib-python==3.4.1.15
Out[2]: '3.4.1'

得到特征点

In [3]: sift = cv2.xfeatures2d.SIFT_create()
kp = sift.detect(gray, None)

In [4]: img = cv2.drawKeypoints(gray, kp, img)
cv2.imshow('drawKeypoints', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

pip list

pip uninstall opencv-3.4.3

```
cv2.destroyAllWindows()

计算特征

In [21]: kp, des = sift.compute(gray, kp)

In [22]: print(np.array(kp).shape)
(6827,)

In [10]: des.shape
Out[10]: (6827, 128)

In [11]: des[0]

Out[11]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  21.,  8.,  0.,
       0.,  0.,  0.,  0., 157., 31.,  3.,  1.,  0.,  0.,
       2.,  63.,  75.,  7.,  20., 35., 31., 74., 23., 66.,  0.,
       0.,  1.,  3.,  4.,  1.,  0.,  0., 76., 15., 13., 27.,
       8.,  1.,  0.,  2., 157., 112., 50., 31.,  2.,  0.,  0.,
       9.,  49.,  42., 157., 157., 12.,  4.,  1.,  5.,  1., 13.,
      7., 12., 41., 5.,  0.,  0., 104., 8.,  5., 10., 53.]
```

案例实战--全景拼接

特征匹配

Brute-Force暴力匹配

```
In [4]: import cv2  
import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline
```

```
In [5]: img1 = cv2.imread('box.png', 0)  
img2 = cv2.imread('box_in_scene.png', 0)
```

```
In [7]: def cv_show(name, img):  
    cv2.imshow(name, img)  
    cv2.waitKey(0)  
    cv2.destroyAllWindows()
```

```
In [10]: cv_show('img1', img1)
```

```
In [11]: cv_show('img2', img2)
```

```
In [12]: sift = cv2.xfeatures2d.SIFT_create()
```

```
In [14]: kp1, des1 = sift.detectAndCompute(img1, None)  
kp2, des2 = sift.detectAndCompute(img2, None)
```

```
In [26]: x crossCheck表示两个特征点要互相匹，例如A中的第i个特征点与B中的第j个特征点最近的，并且B中的第j个特征点到A中的第i个特征点也是xYORL2: 归一化数组的(欧几里德距离)，如果其他特征计算方法需要考虑不同的匹配计算方式  
bf = cv2.BFMatcher(crossCheck=True)
```

看哪两个特征向量最为匹配

1对1的匹配

```
In [28]: matches = bf.match(des1, des2)  
matches = sorted(matches, key=lambda x: x.distance)
```

```
In [29]: img3 = cv2.drawMatches(img1, kp1, img2, kp2, matches[:10], None, flags=2)
```

```
In [30]: cv_show('img3', img3)
```

k对最佳匹配

```
In [32]: bf = cv2.BFMatcher()
matches = bf.knnMatch(des1, des2, k=2)

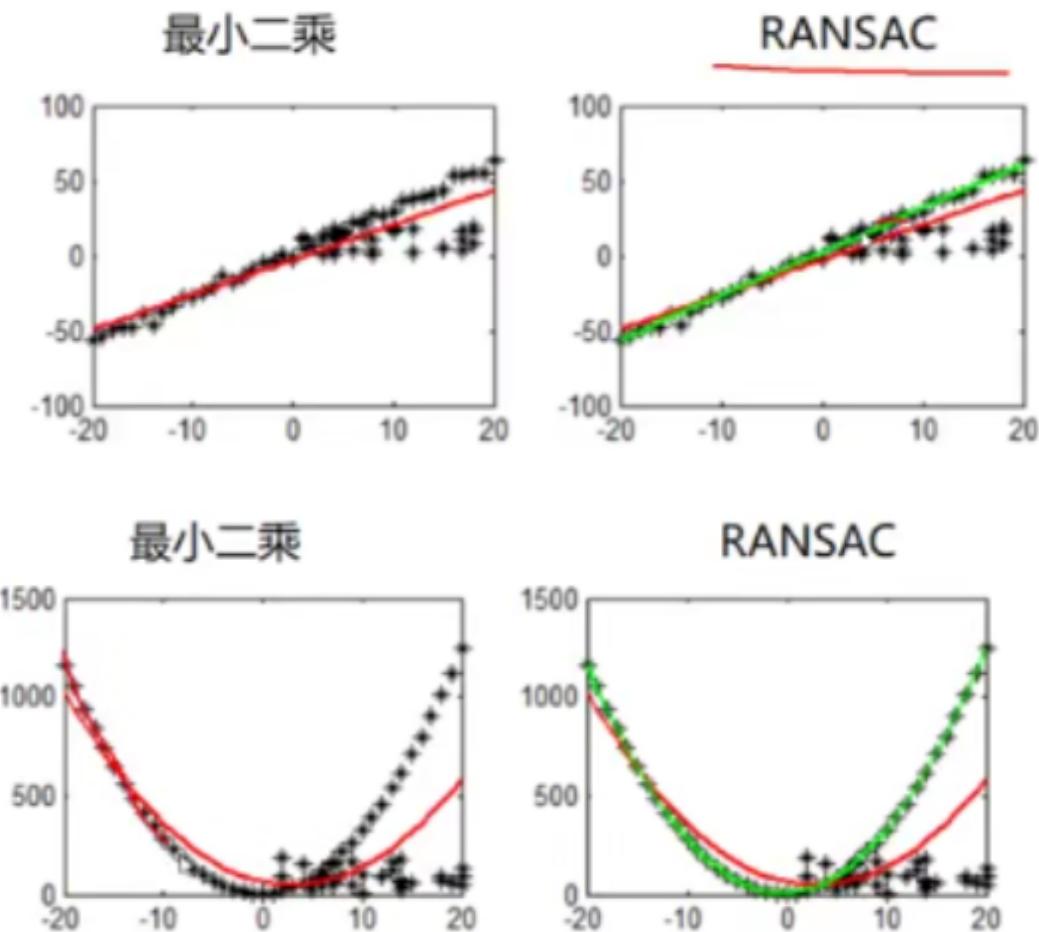
In [33]: good = []
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        good.append([m])

In [34]: img3 = cv2.drawMatchesKnn(img1, kp1, img2, kp2, good, None, flags=2)

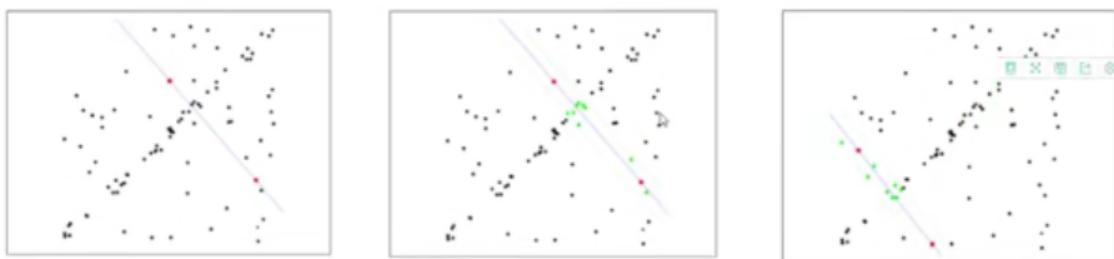
In [35]: cv_show('img3', img3)
```

如果需要更快速完成操作，可以尝试使用 `cv2.FlannBasedMatcher`

随机抽样一致算法 (Random sample consensus, RANSAC)



选择初始样本点进行拟合，给定一个容忍范围，不断进行迭代



每一次拟合后，容差范围内都有对应的数据点数，找出数据点个数最多的情况，就是最终的拟合结果

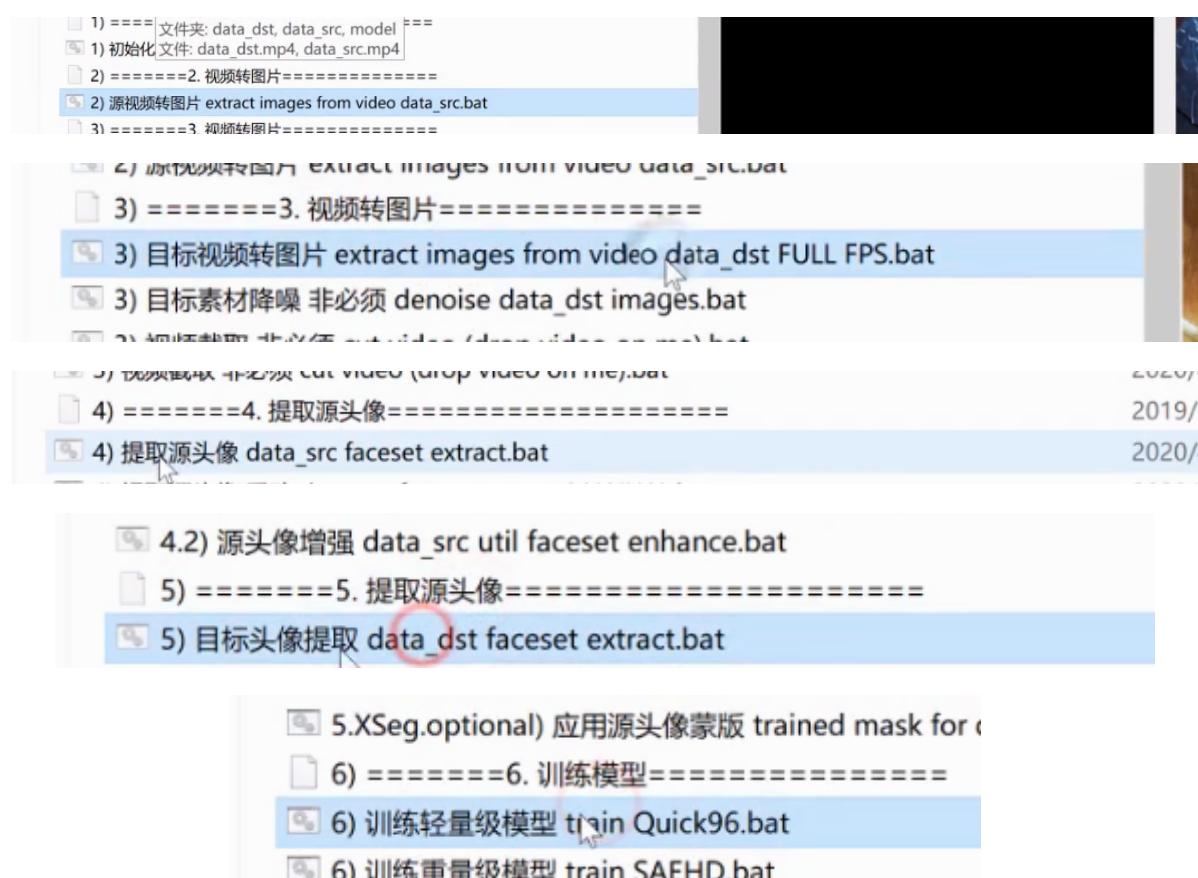
单应性矩阵 H

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_1, y_1, 1, 0, 0, 0, -x'_1 x_1, -x'_1 y_1 \\ 0, 0, 0, x_1, y_1, 1, -y'_1 x_1, -y'_1 y_1 \\ x_2, y_2, 1, 0, 0, 0, -x'_2 x_2, -x'_2 y_2 \\ 0, 0, 0, x_2, y_2, 1, -y'_2 x_2, -y'_2 y_2 \\ x_3, y_3, 1, 0, 0, 0, -x'_3 x_3, -x'_3 y_3 \\ 0, 0, 0, x_3, y_3, 1, -y'_3 x_3, -y'_3 y_3 \\ \vdots, \vdots, \vdots, \vdots, \vdots, \vdots, \vdots, \vdots, \vdots \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ \vdots, \vdots, \vdots, \vdots, \vdots, \vdots, \vdots, \vdots, \vdots \end{bmatrix}$$

最少需要8个未知数，需要4对特征点

找特征点 \Rightarrow 计算单应性矩阵 \Rightarrow 变换 \Rightarrow 拼接



启动训练程序.

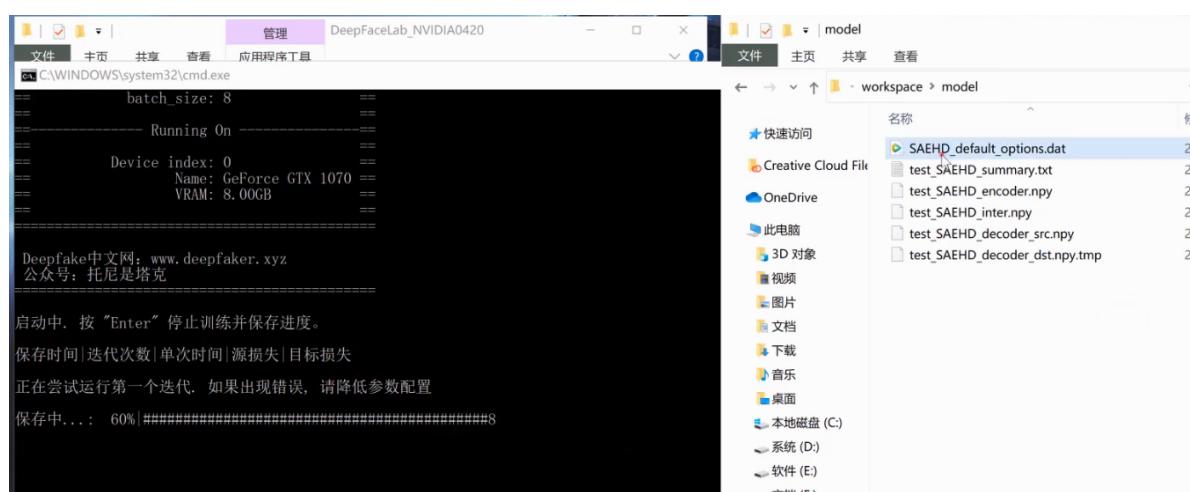
[new] 没有发现模型. 输入一个名词新建模型 : test

首次运行模型.

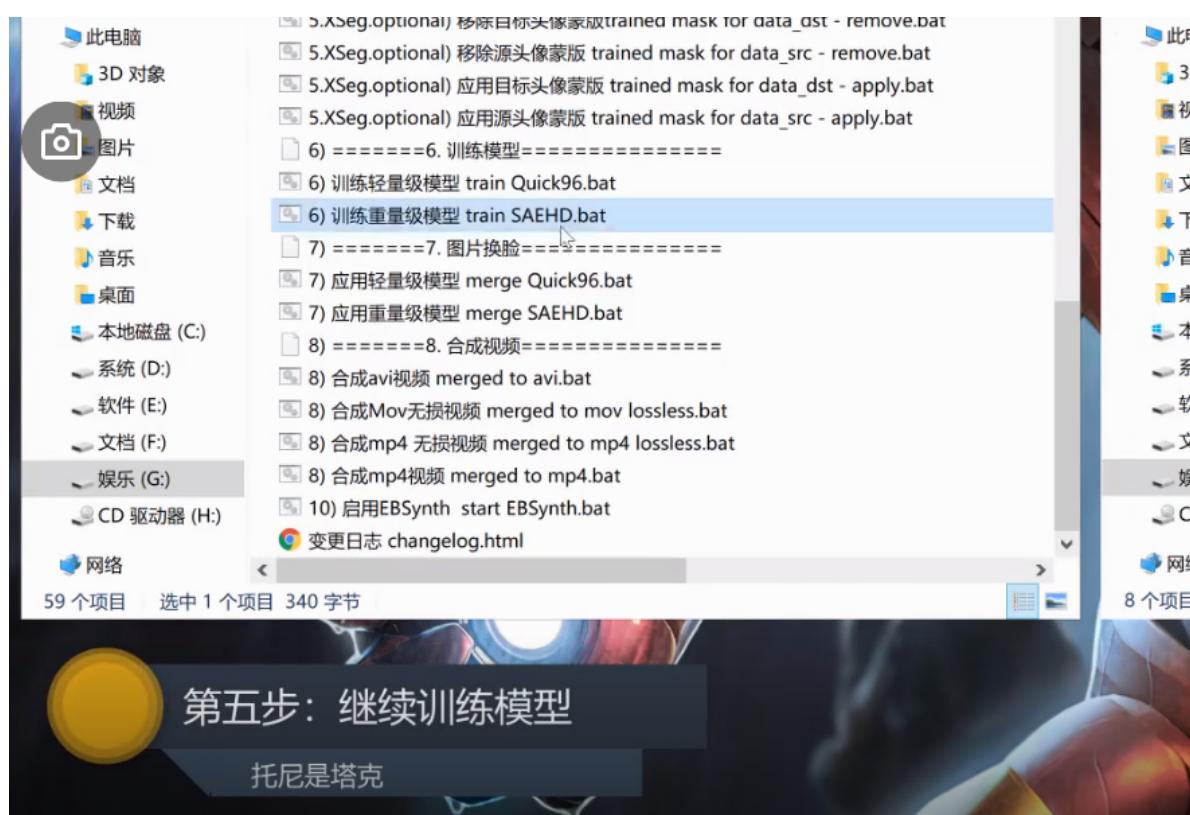
选择CPU或者显卡(separated by comma).

[CPU] : CPU
[0] : GeForce GTX 1070

[0] 选择哪一个显卡? :



按S保存进度



```
启动训练程序.

选择一个模型, 或者输入一个名称去新建模型.
[r] : 重命名
[d] : 删除

[0] : test - latest
:
0
加载名为 test_SAEHD 的模型...

选择CPU或者显卡(separated by comma).

[CPU] : CPU
[0] : GeForce GTX 1070
[0] 选择哪一个显卡? :
```





- ⑦ 应用重量级模型 merge SAEHD.bat
- ⑧ ======8. 合成视频=====
- ⑧ 合成avi视频 merged to avi.bat
- ⑧ 合成Mov无损视频 merged to mov lossless.bat
- ⑧ 合成mp4 无损视频 merged to mp4 lossless.bat
- ⑧ 合成mp4视频 merged to mp4.bat (highlighted)
- ⑩ 启用EBSynth start EBSynth.bat
- 变更日志 changelog.html

背景建模

帧差法

由于场景中的目标在运动，目标的影像在不同图像帧中的位置不同。该类算法对时间上连续的两帧图像进行差分运算，不同帧对应的像素点相减，判断灰度差的绝对值，当绝对值超过一定阈值时，即可判断为运动目标，从而实现目标的检测功能。

$$D_s(x, y) = |f_s(x, y) - f_{ref}(x, y)|$$

$$R_s(x, y) = \begin{cases} 255, & D_s(x, y) > T \\ 0, & \text{else} \end{cases}$$



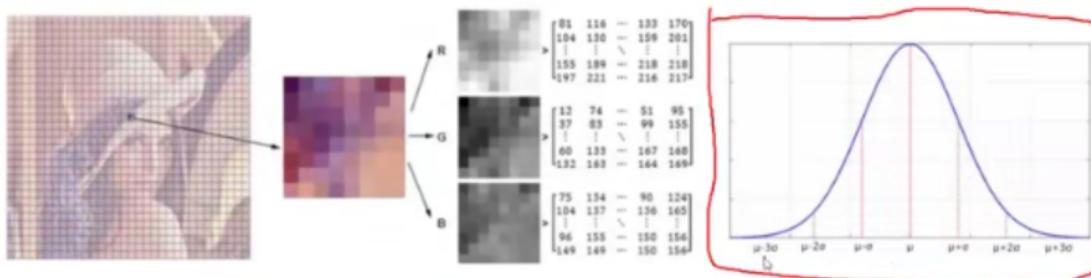
- 保证背景基本不发生变化 => 相机是静止的
- 帧差法
- 帧差法非常简单，但是会引入噪声和空洞问题

混合高斯模型

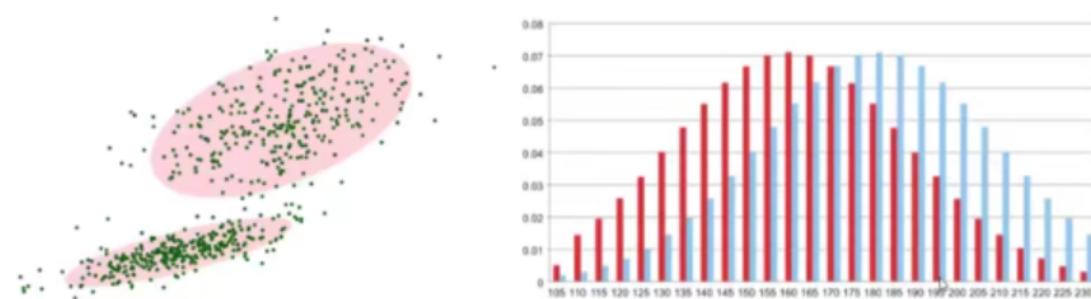
混合高斯模型

在进行前景检测前，先对背景进行训练，对图像中每个背景采用一个混合高斯模型进行模拟。每个背景的混合高斯的个数可以自适应。然后在测试阶段，对新来的像素进行GMM匹配，如果该像素值能够匹配其中一个高斯，则认为是背景，否则认为是前景。由于整个过程GMM模型在不断更新学习中，所以对动态背景有一定的鲁棒性。最后通过对一个有树枝摇摆的动态背景进行前景检测，取得了较好的效果。

在视频中对于像素点的变化情况应当是符合高斯分布



背景的实际分布应当是多个高斯分布混合在一起，每个高斯模型也可以带有权重



混合高斯模型学习方法

- 1.首先初始化每个高斯模型矩阵参数。
- 2.取视频中T帧数据图像用来训练高斯混合模型。来了第一个像素之后用它来当做第一个高斯分布。
- 3.当后面来的像素值时，与前面已有的高斯的均值比较，如果该像素点的值与其模型均值差在3倍的方差内，则属于该分布，并对其进行参数更新。
- 4.如果下一次来的像素不满足当前高斯分布，用它来创建一个新的高斯分布。

- 3σ原则
- 不断的对μ和σ进行更新

• 3-5个高斯分布

混合高斯模型测试方法

在测试阶段，对新采像素点的值与混合高斯模型中的每一个均值进行比较，如果其差值在2倍的方差之间的话，则认为是背景，否则认为是前景。将前景赋值为255，背景赋值为0。这样就形成了一副前景二值图。



```
] import numpy as np
import cv2

#经典的测试视频
cap = cv2.VideoCapture('test.avi')           I
#形态学操作需要使用
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3))
#创建混合高斯模型用于背景建模
fgbg = cv2.createBackgroundSubtractorMOG2()

while(1):
    ret, frame = cap.read()
    fgmask = fgbg.apply(frame)
    #形态学开运算去噪点
    fgmask = cv2.morphologyEx(fgmask, cv2.MORPH_OPEN, kernel)
    #寻找视频中的轮廓
    im, contours, hierarchy = cv2.findContours(fgmask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    for c in contours:
        #计算各轮廓的周长
        perimeter = cv2.arcLength(c, True)
        if perimeter > 188:
            #找到一个直矩形（不会旋转）
            x, y, w, h = cv2.boundingRect(c)

            while(True):
                ret, frame = cap.read()
                fgmask = fgbg.apply(frame)
                #形态学开运算去噪点
                fgmask = cv2.morphologyEx(fgmask, cv2.MORPH_OPEN, kernel)
                #寻找视频中的轮廓
                im, contours, hierarchy = cv2.findContours(fgmask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

                for c in contours:
                    #计算各轮廓的周长
                    perimeter = cv2.arcLength(c, True)
                    if perimeter > 188:
                        #找到一个直矩形（不会旋转）
                        x, y, w, h = cv2.boundingRect(c)
                        #画出这个矩形
                        cv2.rectangle(frame, (x, y), (x+w, y+h), (0,255,0), 2)

                cv2.imshow('frame',frame)
                cv2.imshow('fgmask', fgmask)
                k = cv2.waitKey(100) & 0xff
                if k == 27:
                    break
            cap.release()
```

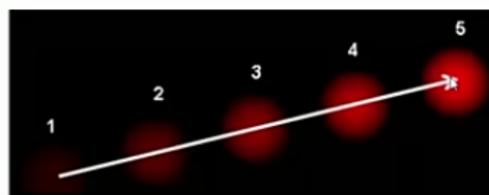
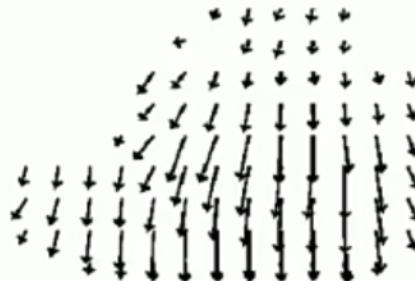
cv2.destroyAllWindows()

光流估计

光流估计

光流是空间运动物体在观测成像平面上的像素运动的“瞬时速度”，根据各个像素点的速度矢量特征，可以对图像进行动态分析，例如目标跟踪。

- 亮度恒定：同一点随着时间的变化，其亮度不会发生改变。
- 小运动：随着时间的变化不会引起位置的剧烈变化，只有小运动情况下才能用前后帧之间单位位置变化引起的灰度变化去近似灰度对位置的偏导数。
- 空间一致：一个场景上邻近的点投影到图像上也是邻近点，且邻近点速度一致。因为光流法基本方程约束只有一个，而要求x, y方向的速度，有两个未知变量。所以需要建立n多个方程求解。



- 亮度恒定
- 空间一致
- 小运动

Lucas-Kanade 算法

约束方程：

$$I(x, y, t) = I(x + dx, y + dy, t + dt)$$

$$= I(x, y, t) + \frac{\partial I}{\partial x} dx + \frac{\partial I}{\partial y} dy + \frac{\partial I}{\partial t} dt$$

$$I_x dx + I_y dy + I_t dt = 0$$

$$\underline{I_x \boxed{u} + I_y \boxed{v} = -I_t} \rightarrow \begin{bmatrix} I_x & I_y \end{bmatrix} \cdot \begin{bmatrix} u \\ v \end{bmatrix} = -I_t$$

如何求解方程组呢？看起来一个像素点根本不够，在物体移动过程中还有哪些特性呢？

$$\begin{bmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ \vdots & \vdots \\ I_{x5} & I_{y5} \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_{t1} \\ I_{t2} \\ \vdots \\ I_{t5} \end{bmatrix}$$

最小二乘法：

$$A\bar{u} = b$$



$$\bar{u} = (A^T A)^{-1} A^T b$$

一定可逆吗？

$$\underbrace{A^T A}_{2 \times 2} \underbrace{\bar{u}}_{2 \times 1} = \underbrace{A^T b}_{2 \times 1}$$

$$\therefore (A^T A)^{-1} A^T b$$

$$A^T A = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}$$

最小二乘法：

$$A\bar{u} = b$$



$$\bar{u} = (A^T A)^{-1} A^T b$$

一定可逆吗？

$$\underbrace{A^T A}_{2 \times 2} \underbrace{\bar{u}}_{2 \times 1} = \underbrace{A^T b}_{2 \times 1}$$

$$\bar{u} = (A^T A)^{-1} A^T b$$

$$A^T A = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}$$

cv2.calcOpticalFlowPyrLK():

参数：

- prevImage 前一帧图像
- nextImage 当前帧图像
- prevPts 待跟踪的特征点向量
- winSize 搜索窗口的大小
- maxLevel 最大的金字塔层数

返回：

- nextPts 输出跟踪特征点向量
- status 特征点是否找到，找到的状态为1，未找到的状态为0

```

In [7]: import numpy as np
import cv2

cap = cv2.VideoCapture('test.avi')

# 角点检测所需参数
feature_params = dict(maxCorners = 100,
                       qualityLevel = 0.3,
                       minDistance = 7)

# lucas kanade参数
lk_params = dict(winSize = (15,15),
                  maxLevel = 2)

# 随机颜色
color = np.random.randint(0,255,(100,3))

# 拿到第一帧图像
ret, old_frame = cap.read()
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
# 返回所有检测特征点，需要输入图像，角点最大数量（效率），品质因子（特征值越大的越好，来筛选）
# 距离相当于这区间有比这个角点强的，就不要这个弱的了
p0 = cv2.goodFeaturesToTrack(old_gray, mask = None, **feature_params)      I

```

```

# 创建一个mask
mask = np.zeros_like(old_frame)

while(True):
    ret, frame = cap.read()
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # 需要传入前一帧和当前图像以及前一帧检测到的角点
    p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None, **lk_params)

    # st=1表示
    good_new = p1[st==1]
    good_old = p0[st==1]

    # 绘制轨迹
    for i,(new,old) in enumerate(zip(good_new,good_old)):
        a,b = new.ravel()
        c,d = old.ravel()
        mask = cv2.line(mask, (a,b),(c,d), color[i].tolist(), 2)
        frame = cv2.circle(frame,(a,b),5,color[i].tolist(),-1)
    img = cv2.add(frame,mask)

```

```

cv2.imshow('frame',img)
k = cv2.waitKey(150) & 0xff
if k == 27:
    break

```

```

# 更新
old_gray = frame_gray.copy()
p0 = good_new.reshape(-1,1,2)  I
cv2.destroyAllWindows()
cap.release()

```

角点的lambda值较大，使得矩阵是可逆的

Opencv的DNN模块

```

9
10 # Caffe所需配置文件
11 net = cv2.dnn.readNetFromCaffe("bvlc_googlenet.prototxt",
12     "bvlc_googlenet.caffemodel")
13

```

cv2.dnn.readNetFromCaffe()

```

4 # 导入模块
5 import utils_paths
6 import numpy as np
7 import cv2
8
9 # 读取文件处理
10 rows = open("synset_words.txt").read().strip().split("\n")
11 classes = [r[r.find(" ") + 1:].split(",")[0] for r in rows]
12
13 # Caffe所需配置文件
14 net = cv2.dnn.readNetFromCaffe("bvlc_googlenet.prototxt",
15     "bvlc_googlenet.caffemodel") I
16
17 # 显示路径
18 imagePaths = sorted(list(utils_paths.list_images("images/")))
19
20 # 附录数据预处理
21 image = cv2.imread(imagePaths[0])
22 resized = cv2.resize(image, (224, 224))
23 # image scaleFactor size mean swapRB
24 blob = cv2.dnn.blobFromImage(resized, 1, (224, 224), (104, 117, 123))
25 print("First Blob: {}".format(blob.shape))

```

- resize
- 减去均值
- 根据实际情况进行二次化定制

```

26 # 向前传播结果
27 net.setInput(blob)
28 preds = net.forward()
29
30 # 排序，取分类可能性最大的
31 idx = np.argsort(preds[0])[:-1][0]
32 text = "Label: {:.2f}%".format(classes[idx],
33     preds[0][idx] * 100)
34 cv2.putText(image, text, (5, 25), cv2.FONT_HERSHEY_SIMPLEX,
35     0.7, (0, 0, 255), 2) I
36 cv2.imshow("Image", image)
37 cv2.waitKey(0)
38
39 # Batch数据制作
40 images = []
41
42 # 读取一片，输出是一个batch
43 for p in imagePaths[1:]:
44     image = cv2.imread(p)
45     image = cv2.resize(image, (224, 224))
46     images.append(image) I
47
48 # blobFromImages函数，切忌有空格
49 blob = cv2.dnn.blobFromImages(images, 1, (224, 224), (104, 117, 123))
50 print("Second Blob: {}".format(blob.shape))
51
52 # 读取数据结果
53 net.setInput(blob)
54 preds = net.forward()
55 for (i, p) in enumerate(imagePaths[1:]):
56     image = cv2.imread(p)
57     idx = np.argsort(preds[i])[:-1][0]
58     text = "Label: {:.2f}%".format(classes[idx],
59         preds[i][idx] * 100)
60     cv2.putText(image, text, (5, 25), cv2.FONT_HERSHEY_SIMPLEX,
61         0.7, (0, 0, 255), 2)
62     cv2.imshow("Image", image)

```

```
52 # 显示预测结果
53 net.setInput(blob)
54 preds = net.forward()
55 for (i, p) in enumerate(image@aths[1:]):
56     image = cv2.imread(p)
57     idx = np.argsort(preds[1])[:-1][0]
58     text = "Label: {}, {:.2f}%".format(classes[idx],
59                                         preds[1][idx] * 100)
60     cv2.putText(image, text, (5, 25), cv2.FONT_HERSHEY_SIMPLEX,
61                 0.7, (0, 0, 255), 2)
62     cv2.imshow("Image", image)
63     cv2.waitKey(0)
```