

# OpenCV-Python 中文教程

OpenCV官方教程中文版 (For Python)

段力辉 译

<http://privategarden.duapp.com/>



# 为什么翻译此书？

段力辉

2014 年 2 月 16 日

## 1 为什么使用 Python

Python 作为一种高效简洁的直译式语言非常适合我们用来解决日常工作的问题。而且它简单易学，初学者几个小时就可以基本入门。再加上 Numpy 和 matplotlib 这两个翅膀，Python 对数据分析的能力不逊于 Matlab。Python 还被称为是胶水语言，有很多软件都提供了 Python 接口。尤其是在 linux 下，可以使用 Python 将不同的软件组成一个工作流，发挥每一个软件自己最大的优势从而完成一个复杂的任务。比如我们可以使用 Mysql 存储数据，使用 R 分析数据，使用 matplotlib 展示数据，使用 OpenGL 进行 3D 建模，使用 Qt 构建漂亮的 GUI。而 Python 可以将他们联合在一起构建一个强大的工作流。

## 2 为什么使用 Python-OpenCV

虽然 python 很强大，而且也有自己的图像处理库 PIL，但是相对于 OpenCV 来讲，它还是弱小很多。跟很多开源软件一样 OpenCV 也提供了完善的 python 接口，非常便于调用。OpenCV 的稳定版是 2.4.8，最新版是 3.0，包含了超过 2500 个算法和函数，几乎任何一个能想到的成熟算法都可以通过调用 OpenCV 的函数来实现，超级方便。

## 3 为什么是这本书

但是非常可惜关于在 Python 下使用 OpenCV 的书，除了这本在线教程之外，仅有一个 100 多页的书 opencv computer vision with python（本

书虽然挺好的，但是不够全面，不能让读者完全了解 opencv 的现状)。而我翻译的这本书是来源于 OpenCV 的官方文档，内容全面，对各种的算法的描述简单易懂，而且不拘泥于长篇大论的数学推导，非常适合想使用 OpenCV 解决实际问题的人，对他们来说具体的数学原理并不重要，重要的是能解决实际问题。

在国内这本书可以说是第一本 Python\_OpenCV 的译作。

## 4 本书的时效性

本书的编写时针对最新的 OpenCV3.0 的，本版本还没有正式发布（但很稳定），其中的内容页非常新，甚至用到了 2012 年才提出的算法。因此本书的时效性上应该是没有问题的。

## 5 本书的目标读者

本书针的读者是高校学生，科研工作者，图像处理爱好者。对于这些人群，他们往往是带着具体的问题，在苦苦寻找解决方案。为了解决一个小问题就让他们去学习 C++ 这么深奥的语言几乎是不可能的。而 Python 的悄然兴起给他们带来的希望，如果说 C++ 是 tex 的话，那 Python 的易用性相当于 word。他们可以很快的看懂本书的所有代码，并可以学着使用它们来解决自己的问题，同时也能拓展自己的视野。别人经常说 Python 不够快，但是对于上面的这些读者，我相信这不是问题，现在我们日常使用的 PC 机已经无比强大了，而且绝大多数情况下不会用到实时处理，更不会在嵌入式设备上使用。因此这不是问题。

# OpenCV-Python

段力辉

2014 年 1 月 30 日

# 目录

<b>I</b>	<b>走进 OpenCV</b>	<b>10</b>
1	关于 OpenCV-Python 教程	10
2	在 Windows 上安装 OpenCV-Python	11
3	在 Fedora 上安装 OpenCV-Python	12
<b>II</b>	<b>OpenCV 中的 Gui 特性</b>	<b>13</b>
4	图片	13
4.1	读入图像 . . . . .	13
4.2	显示图像 . . . . .	14
4.3	保存图像 . . . . .	15
4.4	总结一下 . . . . .	15
5	视频	18
5.1	用摄像头捕获视频 . . . . .	18
5.2	从文件中播放视频 . . . . .	19
5.3	保存视频 . . . . .	21
6	OpenCV 中的绘图函数	24
6.1	画线 . . . . .	24
6.2	画矩形 . . . . .	24
6.3	画圆 . . . . .	25
6.4	画椭圆 . . . . .	25
6.5	画多边形 . . . . .	25
6.6	在图片上添加文字 . . . . .	26
7	把鼠标当画笔	28
7.1	简单演示 . . . . .	28
7.2	高级一点的示例 . . . . .	29
8	用滑动条做调色板	32
8.1	代码示例 . . . . .	32
<b>III</b>	<b>核心操作</b>	<b>36</b>

<b>9 图像的基础操作</b>	<b>36</b>
9.1 获取并修改像素值 . . . . .	36
9.2 获取图像属性 . . . . .	38
9.3 图像 ROI . . . . .	39
9.4 拆分及合并图像通道 . . . . .	40
9.5 为图像扩边（填充） . . . . .	41
<b>10 图像上的算术运算</b>	<b>43</b>
10.1 图像加法 . . . . .	43
10.2 图像混合 . . . . .	43
10.3 按位运算 . . . . .	44
<b>11 程序性能检测及优化</b>	<b>47</b>
11.1 使用 OpenCV 检测程序效率 . . . . .	47
11.2 OpenCV 中的默认优化 . . . . .	48
11.3 在 IPython 中检测程序效率 . . . . .	49
11.4 更多 IPython 的魔法命令 . . . . .	51
11.5 效率优化技术 . . . . .	51
<b>12 OpenCV 中的数学工具</b>	<b>53</b>
<b>IV OpenCV 中的图像处理</b>	<b>54</b>
<b>13 颜色空间转换</b>	<b>54</b>
13.1 转换颜色空间 . . . . .	54
13.2 物体跟踪 . . . . .	55
13.3 怎样找到要跟踪对象的 HSV 值？ . . . . .	57
<b>14 几何变换</b>	<b>59</b>
14.1 扩展缩放 . . . . .	59
14.2 平移 . . . . .	60
14.3 旋转 . . . . .	62
14.4 仿射变换 . . . . .	63
14.5 透视变换 . . . . .	64
<b>15 图像阈值</b>	<b>66</b>
15.1 简单阈值 . . . . .	66
15.2 自适应阈值 . . . . .	68
15.3 Otsu's 二值化 . . . . .	70
15.4 Otsu's 二值化是如何工作的？ . . . . .	72

<b>16 图像平滑</b>	<b>75</b>
16.1 平均	77
16.2 高斯模糊	78
16.3 中值模糊	79
16.4 双边滤波	79
<b>17 形态学转换</b>	<b>81</b>
17.1 腐蚀	81
17.2 膨胀	82
17.3 开运算	83
17.4 闭运算	83
17.5 形态学梯度	83
17.6 礼帽	84
17.7 黑帽	84
17.8 形态学操作之间的关系	84
<b>18 图像梯度</b>	<b>87</b>
18.1 Sobel 算子和 Scharr 算子	87
18.2 Laplacian 算子	87
<b>19 Canny 边缘检测</b>	<b>91</b>
19.1 原理	91
19.1.1 噪声去除	91
19.1.2 计算图像梯度	91
19.1.3 非极大值抑制	91
19.1.4 滞后阈值	92
19.2 OpenCV 中的 Canny 边界检测	92
<b>20 图像金字塔</b>	<b>94</b>
20.1 原理	94
20.2 使用金字塔进行图像融合	96
<b>21 OpenCV 中的轮廓</b>	<b>101</b>
21.1 初识轮廓	101
21.1.1 什么是轮廓	101
21.1.2 怎样绘制轮廓	101
21.1.3 轮廓的近似方法	102
21.2 轮廓特征	104
21.2.1 矩	104
21.2.2 轮廓面积	104
21.2.3 轮廓周长	105

21.2.4轮廓近似	105
21.2.5凸包	106
21.2.6凸性检测	107
21.2.7边界矩形	107
21.2.8最小外接圆	108
21.2.9椭圆拟合	109
21.2.1直线拟合	109
21.3轮廓的性质	111
21.3.1长宽比	111
21.3.2Extent	111
21.3.3Solidity	111
21.3.4Equivalent Diameter	112
21.3.5方向	112
21.3.6掩模和像素点	112
21.3.7最大值和最小值及它们的位置	113
21.3.8平均颜色及平均灰度	113
21.3.9极点	114
21.4轮廓：更多函数	115
21.4.1凸缺陷	115
21.4.2Point Polygon Test	116
21.4.3形状匹配	117
21.5轮廓的层次结构	119
21.5.1什么是层次结构	119
21.5.2OpenCV 中层次结构	120
21.5.3轮廓检索模式	120
<b>22直方图</b>	<b>124</b>
22.1直方图的计算，绘制与分析	124
22.1.1统计直方图	124
22.1.2绘制直方图	126
22.1.3使用掩模	128
22.2直方图均衡化	130
22.2.1OpenCV 中的直方图均衡化	132
22.2.2CLAHE 有限对比适应性直方图均衡化	132
22.32D 直方图	135
22.3.1介绍	135
22.3.2OpenCV 中的 2D 直方图	135
22.3.3Numpy 中 2D 直方图	136
22.3.4绘制 2D 直方图	136
22.4直方图反向投影	141



22.4.1 Numpy 中的算法 . . . . .	141
22.4.2 OpenCV 中的反向投影 . . . . .	143
<b>23 图像变换</b>	<b>146</b>
23.1 傅里叶变换 . . . . .	146
23.1.1 Numpy 中的傅里叶变换 . . . . .	146
23.1.2 OpenCV 中的傅里叶变换 . . . . .	148
23.1.3 DFT 的性能优化 . . . . .	150
23.1.4 为什么拉普拉斯算子是高通滤波器？ . . . . .	152
<b>24 模板匹配</b>	<b>155</b>
24.1 OpenCV 中的模板匹配 . . . . .	155
24.2 多对象的模板匹配 . . . . .	158
<b>25 Hough 直线变换</b>	<b>160</b>
25.1 OpenCV 中的霍夫变换 . . . . .	161
25.2 Probabilistic Hough Transform . . . . .	163
<b>26 Hough 圆环变换</b>	<b>165</b>
<b>27 分水岭算法图像分割</b>	<b>168</b>
27.1 代码 . . . . .	168
<b>28 使用 GrabCut 算法进行交互式前景提取</b>	<b>173</b>
28.1 演示 . . . . .	174
<b>V 图像特征提取与描述</b>	<b>178</b>
<b>29 理解图像特征</b>	<b>178</b>
29.1 解释 . . . . .	178
<b>30 Harris 角点检测</b>	<b>181</b>
30.1 OpenCV 中的 Harris 角点检测 . . . . .	182
30.2 亚像素级精确度的角点 . . . . .	184
<b>31 Shi-Tomasi 角点检测 &amp; 适合于跟踪的图像特征</b>	<b>187</b>
31.1 代码 . . . . .	187
<b>32 介绍 SIFT(Scale-Invariant Feature Transform)</b>	<b>190</b>
<b>33 介绍 SURF(Speeded-Up Robust Features)</b>	<b>195</b>
33.1 OpenCV 中的 SURF . . . . .	197

<b>34 角点检测的 FAST 算法</b>	<b>200</b>
34.1 使用 FAST 算法进行特征提取 . . . . .	200
34.2 机器学习的角点检测器 . . . . .	201
34.3 非极大值抑制 . . . . .	202
34.4 总结 . . . . .	202
34.5 OpenCV 中 FAST 特征检测器 . . . . .	202
<b>35 BRIEF(Binary Robust Independent Elementary Features)</b>	<b>205</b>
35.1 OpenCV 中的 BRIEF . . . . .	205
<b>36 ORB (Oriented FAST and Rotated BRIEF)</b>	<b>207</b>
36.1 OpenCV 中的 ORB 算法 . . . . .	208
<b>37 特征匹配</b>	<b>211</b>
37.1 Brute-Force 匹配的基础 . . . . .	211
37.2 对 ORB 描述符进行蛮力匹配 . . . . .	212
37.3 匹配器对象是什么? . . . . .	213
37.4 对 SIFT 描述符进行蛮力匹配和比值测试 . . . . .	213
37.5 FLANN 匹配器 . . . . .	214
<b>38 使用特征匹配和单应性查找对象</b>	<b>218</b>
38.1 基础 . . . . .	218
38.2 代码 . . . . .	218
<b>VI 视频分析</b>	<b>222</b>
<b>39 Meanshift 和 Camshift</b>	<b>222</b>
39.1 Meanshift . . . . .	222
39.2 OpenCV 中的 Meanshift . . . . .	223
39.3 Camshift . . . . .	225
39.4 OpenCV 中的 Camshift . . . . .	226
<b>40 光流</b>	<b>231</b>
40.1 光流 . . . . .	231
40.2 Lucas-Kanade 法 . . . . .	232
40.3 OpenCV 中的 Lucas-Kanade 光流 . . . . .	232
40.4 OpenCV 中的稠密光流 . . . . .	235
<b>41 背景减除</b>	<b>238</b>
41.1 基础 . . . . .	238
41.2 BackgroundSubtractorMOG . . . . .	238
41.3 BackgroundSubtractorMOG2 . . . . .	239

41.4 BackgroundSubtractorGMG . . . . .	240
41.5 结果 . . . . .	241
<b>VII 摄像机标定和 3D 重构</b>	<b>243</b>
<b>42 摄像机标定</b>	<b>243</b>
42.1 基础 . . . . .	243
42.2 代码 . . . . .	244
42.2.1 设置 . . . . .	245
42.2.2 标定 . . . . .	247
42.2.3 畸变校正 . . . . .	247
42.3 反向投影误差 . . . . .	249
<b>43 姿势估计</b>	<b>250</b>
43.1 基础 . . . . .	250
43.1.1 渲染一个立方体 . . . . .	252
<b>44 对极几何 ( Epipolar Geometry )</b>	<b>254</b>
44.1 基本概念 . . . . .	254
44.2 代码 . . . . .	255
<b>45 立体图像中的深度地图</b>	<b>259</b>
45.1 基础 . . . . .	259
45.2 代码 . . . . .	259
<b>VIII 机器学习</b>	<b>261</b>
<b>46 K 近邻 ( k-Nearest Neighbour )</b>	<b>261</b>
46.1 理解 K 近邻 . . . . .	261
46.1.1 OpenCV 中的 kNN . . . . .	262
46.2 使用 kNN 对手写数字 OCR . . . . .	266
46.2.1 手写数字的 OCR . . . . .	266
46.2.2 英文字母的 OCR . . . . .	268
<b>47 支持向量机</b>	<b>270</b>
47.1 理解 SVM . . . . .	270
47.1.1 线性数据分割 . . . . .	270
47.1.2 非线性数据分割 . . . . .	271
47.2 使用 SVM 进行手写数据 OCR . . . . .	273

<b>48 K 值聚类</b>	<b>277</b>
48.1 理解 K 值聚类 . . . . .	277
48.1.1 T 恤大小问题 . . . . .	277
48.1.2 它是如何工作的？ . . . . .	278
48.2 OpenCV 中的 K 值聚类 . . . . .	281
48.2.1 理解函数的参数 . . . . .	281
48.2.2 仅有一个特征的数据 . . . . .	282
48.2.3 颜色量化 . . . . .	286
 <b>IX 计算摄影学</b>	 <b>288</b>
<b>49 图像去噪</b>	<b>288</b>
49.1 OpenCV 中的图像去噪 . . . . .	289
49.1.1 cv2.fastNlMeansDenoisingColored() . . . . .	290
49.1.2 cv2.fastNlMeansDenoisingMulti() . . . . .	290
 <b>50 图像修补</b>	 <b>294</b>
50.1 基础 . . . . .	294
50.2 代码 . . . . .	295
 <b>X 对象检测</b>	 <b>297</b>
<b>51 使用 Haar 分类器进行面部检测</b>	<b>297</b>
51.1 基础 . . . . .	297
51.2 OpenCV 中的 Haar 级联检测 . . . . .	299

**部分 I**

## **走进 OpenCV**



### **1 关于 OpenCV-Python 教程**

## 2 在 Windows 上安装 OpenCV-Python



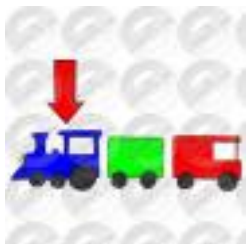
### 3 在 Fedora 上安装 OpenCV-Python



## 部分 II

# OpenCV 中的 Gui 特性

## 4 图片



### 目标

- 在这里你将学会怎样读入一幅图像，怎样显示一幅图像，以及如何保存一幅图像
- 你将要学习如下函数: **cv2.imread()**, **cv2.imshow()**, **cv2.imwrite()**
- 如果你愿意的话，我会叫你如何使用 Matplotlib 显示一幅图片

### 4.1 读入图像

使用函数 **cv2.imread()** 读入图像。这幅图像应该在此程序的工作路径，或者给函数提供完整路径，

第二个参数是要告诉函数应该如何读取这幅图片。

- **cv2.IMREAD\_COLOR**: 读入一副彩色图像。图像的透明度会被忽略，这是默认参数。
- **cv2.IMREAD\_GRAYSCALE**: 以灰度模式读入图像

**cv2.IMREAD\_UNCHANGED**: 读入一幅图像，并且包括图像的 alpha 通道

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Tue Dec 31 19:16:05 2013
4
5 @author: duan
6 """
7
8 import numpy as np
9 import cv2
10
11 # Load an color image in grayscale
12 img = cv2.imread('messi5.jpg',0)
```



**警告：**就算图像的路径是错的，OpenCV 也不会提醒你的，但是当你使用命令 `print img` 时得到的结果是 `None`。

## 4.2 显示图像

使用函数 `cv2.imshow()` 显示图像。窗口会自动调整为图像大小。第一个参数是窗口的名字，其次才是我们的图像。你可以创建多个窗口，只要你喜欢，但是必须给他们不同的名字

```
1 cv2.imshow('image',img)
2 cv2.waitKey(0)
3 cv2.destroyAllWindows()
```

窗口屏幕截图将会像以下的样子 (in Fedora-Gnome machine):



`cv2.waitKey()` 是一个键盘绑定函数。需要指出的是它的时间尺度是毫秒级。函数等待特定的几毫秒，看是否有键盘输入。特定的几毫秒之内，如果按下任意键，这个函数会返回按键的 ASCII 码值，程序将会继续运行。如果没有键盘输入，返回值为 -1，如果我们设置这个函数的参数为 0，那它将会无限期的等待键盘输入。它也可以被用来检测特定键是否被按下，例如按键 a 是否被按下，这个后面我们会接着讨论。

`cv2.destroyAllWindows()` 可以轻易删除任何我们建立的窗口。如果你想删除特定的窗口可以使用 `cv2.destroyWindow()`，在括号内输入你想删除的窗口名。

**建议:** 一种特殊的情况是, 你也可以先创建一个窗口, 之后再加载图像。这种情况下, 你可以决定窗口是否可以调整大小。使用到的函数是 `cv2.namedWindow()`。初始设定函数标签是 `cv2.WINDOW_AUTOSIZE`。但是如果你把标签改成 `cv2.WINDOW_NORMAL`, 你就可以调整窗口大小了。当图像维度太大, 或者要添加轨迹条时, 调整窗口大小将会很有用

代码如下:

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Wed Jan 1 20:49:33 2014
4
5  @author: duan
6  """
7  import numpy as np
8  import cv2
9
10 cv2.namedWindow('image', cv2.WINDOW_NORMAL)
11 cv2.imshow('image',img)
12 cv2.waitKey(0)
13 cv2.destroyAllWindows()
```

### 4.3 保存图像

使用函数 `cv2.imwrite()` 来保存一个图像。首先需要有一个文件名, 之后才是你要保存的图像。

```
1  cv2.imwrite('messigray.png',img)
```

### 4.4 总结一下

下面的程序将会加载一个灰度图, 显示图片, 按下 's' 键保存后退出, 或者按下 ESC 键退出不保存。

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Wed Jan 1 20:49:33 2014
4
5  @author: duan
6  """
7  import numpy as np
8  import cv2
9
10 img = cv2.imread('messi5.jpg',0)
11 cv2.imshow('image',img)
12 k = cv2.waitKey(0)
13 if k == 27: # wait for ESC key to exit
14     cv2.destroyAllWindows()
15 elif k == ord('s'): # wait for 's' key to save and exit
16     cv2.imwrite('messigray.png',img)
17     cv2.destroyAllWindows()

```

**警告：**如果你用的是 64 位系统，你需要将 **k = cv2.waitKey(0)** 这行改成 **k = cv2.waitKey(0)&0xFF**。

## 使用 Matplotlib

Matplotlib 是 python 的一个绘图库，里头有各种各样的绘图方法。之后会陆续了解到。现在，你可以学习怎样用 Matplotlib 显示图像。你可以放大图像，保存它等等。

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Wed Jan 1 20:49:33 2014
4
5  @author: duan
6  """
7  import numpy as np
8  import cv2
9  from matplotlib import pyplot as plt
10
11 img = cv2.imread('messi5.jpg',0)
12 plt.imshow(img, cmap = 'gray', interpolation = 'bicubic')
13 plt.xticks([], plt.yticks([]) # to hide tick values on X and Y axis
14 plt.show()

```

窗口截屏如下：



**参见：**Matplotlib 有多种绘图选择。具体可以参见 Matplotlib docs。我们也会陆续了解一些

**注意：**彩色图像使用 OpenCV 加载时是 BGR 模式。但是 Matplotlib 是 RGB 模式。所以彩色图像如果已经被 OpenCV 读取，那它将不会被 Matplotlib 正确显示。具体细节请看练习

## 附加资源：

[Matplotlib Plotting Styles and Features](#)

## 练习：

1. 当你用 OpenCV 加载一个彩色图像，并用 Matplotlib 显示它时会遇到一些困难。请阅读[this discussion](#)并且尝试理解它。

## 5 视频

### 目标

- 学会读取视频文件，显示视频，保存视频文件
- 学会从摄像头获取并显示视频
- 你将会学习到这些函数：**cv2.VideoCapture()**, **cv2.VideoWrite()**

### 5.1 用摄像头捕获视频

我们经常需要使用摄像头捕获实时图像。OpenCV 为这中应用提供了一个非常简单的接口。让我们使用摄像头来捕获一段视频，并把它转换成灰度视频显示出来。从这个简单的任务开始吧。

为了获取视频，你应该创建一个 **VideoCapture** 对象。他的参数可以是设备的索引号，或者是一个视频文件。设备索引号就是在指定要使用的摄像头。一般的笔记本电脑都有内置摄像头。所以参数就是 0。你可以通过设置成 1 或者其他的来选择别的摄像头。之后，你就可以一帧一帧的捕获视频了。但是最后，别忘了停止捕获视频。

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Fri Jan 3 21:06:22 2014
4
5  @author: duan
6  """
7
8  import numpy as np
9  import cv2
10
11  cap = cv2.VideoCapture(0)
12
13  while(True):
14      # Capture frame-by-frame
15      ret, frame = cap.read()
16
17      # Our operations on the frame come here
18      gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
19
20      # Display the resulting frame
21      cv2.imshow('frame',gray)
22      if cv2.waitKey(1) & 0xFF == ord('q'):
23          break
24
25  # When everything done, release the capture
26  cap.release()
27  cv2.destroyAllWindows()
```

**cap.read()** 返回一个布尔值 ( True/False )。如果帧读取的是正确的, 就是 True。所以最后你可以通过检查他的返回值来查看视频文件是否已经到了结尾。

有时 **cap** 可能不能成功的初始化摄像头设备。这种情况下上面的代码会报错。你可以使用 **cap.isOpened()**, 来检查是否成功初始化了。如果返回值是 True, 那就没有问题。否则就要使用函数 **cap.open()**。

你可以使用函数 **cap.get(propId)** 来获得视频的一些参数信息。这里 **propId** 可以是 0 到 18 之间的任何整数。每一个数代表视频的一个属性, 见下表

其中的一些值可以使用 **cap.set(propId,value)** 来修改, **value** 就是你想要设置成的新值。

例如, 我可以使用 **cap.get(3)** 和 **cap.get(4)** 来查看每一帧的宽和高。默认情况下得到的值是 640X480。但是我可以使用 **ret=cap.set(3,320)** 和 **ret=cap.set(4,240)** 来把宽和高改成 320X240。

**注意:** 当你的程序报错时, 你首先应该检查的是你的摄像头是否能够在其他程序中正常工作 ( 比如 linux 下的 Cheese )。

## 5.2 从文件中播放视频

与从摄像头中捕获一样, 你只需要把设备索引号改成视频文件的名字。在播放每一帧时, 使用 **cv2.waitKey()** 设置适当的持续时间。如果设置的太低视频就会播放的非常快, 如果设置的太高就会播放的很慢 ( 你可以使用这种方法控制视频的播放速度 )。通常情况下 25 毫秒就可以了。

- **CV\_CAP\_PROP\_POS\_MSEC** Current position of the video file in milliseconds.
- **CV\_CAP\_PROP\_POS\_FRAMES** 0-based index of the frame to be decoded/captured next.
- **CV\_CAP\_PROP\_POS\_AVI\_RATIO** Relative position of the video file: 0 - start of the film, 1 - end of the film.
- **CV\_CAP\_PROP\_FRAME\_WIDTH** Width of the frames in the video stream.
- **CV\_CAP\_PROP\_FRAME\_HEIGHT** Height of the frames in the video stream.
- **CV\_CAP\_PROP\_FPS** Frame rate.
- **CV\_CAP\_PROP\_FOURCC** 4-character code of codec.
- **CV\_CAP\_PROP\_FRAME\_COUNT** Number of frames in the video file.
- **CV\_CAP\_PROP\_FORMAT** Format of the Mat objects returned by retrieve() .
- **CV\_CAP\_PROP\_MODE** Backend-specific value indicating the current capture mode.
- **CV\_CAP\_PROP\_BRIGHTNESS** Brightness of the image (only for cameras).
- **CV\_CAP\_PROP\_CONTRAST** Contrast of the image (only for cameras).
- **CV\_CAP\_PROP\_SATURATION** Saturation of the image (only for cameras).
- **CV\_CAP\_PROP\_HUE** Hue of the image (only for cameras).
- **CV\_CAP\_PROP\_GAIN** Gain of the image (only for cameras).
- **CV\_CAP\_PROP\_EXPOSURE** Exposure (only for cameras).
- **CV\_CAP\_PROP\_CONVERT\_RGB** Boolean flags indicating whether images should be converted to RGB.
- **CV\_CAP\_PROP\_WHITE\_BALANCE** Currently unsupported
- **CV\_CAP\_PROP\_RECTIFICATION** Rectification flag for stereo cameras (note: only supported by DC1394 v 2.x backend currently)

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Fri Jan 3 21:06:22 2014
4
5  @author: duan
6  """
7
8  import numpy as np
9  import cv2
10
11  cap = cv2.VideoCapture(0)
12
13  # Define the codec and create VideoWriter object
14  fourcc = cv2.VideoWriter_fourcc(*'XVID')
15  out = cv2.VideoWriter('output.avi',fourcc, 20.0, (640,480))
16
17  while(cap.isOpened()):
18      ret, frame = cap.read()
19      if ret==True:
20          frame = cv2.flip(frame,0)
21
22          # write the flipped frame
23          out.write(frame)
24
25          cv2.imshow('frame',frame)
26          if cv2.waitKey(1) & 0xFF == ord('q'):
27              break
28      else:
29          break
30
31  # Release everything if job is finished
32  cap.release()
33  out.release()
34  cv2.destroyAllWindows()

```

**注意：**你应该确保你已经装了合适版本的 ffmpeg 或者 gstreamer。如果你装错了那就比较头疼了。

### 5.3 保存视频

在我们捕获视频，并对每一帧都进行加工之后我们想要保存这个视频。对于图片来时很简单只需要使用 `cv2.imwrite()`。但对于视频来说就要多做点工作。

这次我们要创建一个 **VideoWriter** 的对象。我们应该确定一个输出文件的名称。接下来指定 FourCC 编码（下面会介绍）。播放频率和帧的大小也都需要确定。最后一个是 `isColor` 标签。如果是 `True`，每一帧就是彩色图，否



则就是灰度图。

FourCC 就是一个 4 字节码，用来确定视频的编码格式。可用的编码列表可以从[fourcc.org](http://fourcc.org)查到。这是平台依赖的。下面这些编码器对我来说是有用的。

- In Fedora: DIVX, XVID, MJPG, X264, WMV1, WMV2. (XVID is more preferable. MJPG results in high size video. X264 gives very small size video)
- In Windows: DIVX (More to be tested and added)
- In OSX : (I don't have access to OSX. Can some one fill this?)

FourCC 码以下的格式传给程序，以 MJPG 为例：

**cv2.cv.FOURCC('M','J','P','G')** 或者 **cv2.cv.FOURCC(\*'MJPG')**。

下面的代码是从摄像头中捕获视频，沿水平方向旋转每一帧并保存它。

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Fri Jan 3 21:06:22 2014
4
5  @author: duan
6  """
7
8  import numpy as np
9  import cv2
10
11  cap = cv2.VideoCapture(0)
12
13  # Define the codec and create VideoWriter object
14  fourcc = cv2.cv.FOURCC(*'XVID')
15  out = cv2.VideoWriter('output.avi',fourcc, 20.0, (640,480))
16
17  while(cap.isOpened()):
18      ret, frame = cap.read()
19      if ret==True:
20          frame = cv2.flip(frame,0)
21
22          # write the flipped frame
23          out.write(frame)
24
25          cv2.imshow('frame',frame)
26          if cv2.waitKey(1) & 0xFF == ord('q'):
27              break
28      else:
29          break
30
31  # Release everything if job is finished
32  cap.release()
33  out.release()
34  cv2.destroyAllWindows()
```

**更多资源**

**练习**

## 6 OpenCV 中的绘图函数

### 目标

- 学习使用 OpenCV 绘制不同几何图形
- 你将会学习到这些函数: `cv2.line()`, `cv2.circle()`, `cv2.rectangle()`, `cv2.ellipse()`, `cv2.putText()` 等。

### 代码

上面所有的这些绘图函数需要设置下面这些参数:

- `img`: 你想要绘制图形的那幅图像。
- `color`: 形状的颜色。以 RGB 为例, 需要传入一个元组, 例如: `(255,0,0)` 代表蓝色。对于灰度图只需要传入灰度值。
- `thickness`: 线条的粗细。如果给一个闭合图形设置为 `-1`, 那么这个图形就会被填充。默认值是 1。
- `linetype`: 线条的类型, 8 连接, 抗锯齿等。默认情况是 8 连接。`cv2.LINE_AA` 为抗锯齿, 这样看起来会非常平滑。

### 6.1 画线

要画一条线, 你只需要告诉函数这条线的起点和终点。我们下面会画一条从左上方到右下角的蓝色线段。

```
1 import numpy as np
2 import cv2
3
4 # Create a black image
5 img=np.zeros((512,512,3), np.uint8)
6
7 # Draw a diagonal blue line with thickness of 5 px
8 cv2.line(img,(0,0),(511,511),(255,0,0),5)
9
```

### 6.2 画矩形

要画一个矩形, 你需要告诉函数的左上角顶点和右下角顶点的坐标。这次我们会在图像的右上角画一个绿色的矩形。

```
1 cv2.rectangle(img, (384,0), (510,128), (0,255,0), 3)
```

## 6.3 画圆

要画圆的话，只需要指定圆形的中心点坐标和半径大小。我们在上面的矩形中画一个圆。

```
1 cv2.circle(img, (447,63), 63, (0,0,255), -1)
```

## 6.4 画椭圆

画椭圆比较复杂，我们要多输入几个参数。一个参数是中心点的位置坐标。下一个参数是长轴和短轴的长度。椭圆沿逆时针方向旋转的角度。椭圆弧演顺时针方向起始的角度和结束角度，如果是 **0** 很 **360**，就是整个椭圆。查看 **cv2.ellipse()** 可以得到更多信息。下面的例子是在图片的中心绘制半个椭圆。

```
1 cv2.ellipse(img, (256,256), (100,50), 0, 0, 180, 255, -1)
```

## 6.5 画多边形

画多边形，需要指点每个顶点的坐标。用这些点的坐标构建一个大小等于行数 **X1X2** 的数组，行数就是点的数目。这个数组的数据类型必须为 **int32**。这里画一个黄色的具有四个顶点的多边形。

```
1 pts=np.array([[10,5],[20,30],[70,20],[50,10]], np.int32)
2 pts=pts.reshape((-1,1,2))
3 # 这里 reshape 的第一个参数为-1，表明这一维的长度是根据后面的维度的计算出来的。
```

**注意：**如果第三个参数是 **False**，我们得到的多边形是不闭合的（首尾不相连）。

**注意：**`cv2.polylines()` 可以被用来画很多条线。只需要把想要画的线放在一个列表中，将这个列表传给函数就可以了。每条线都会被独立绘制。这会比用 `cv2.line()` 一条一条的绘制要快一些。

## 6.6 在图片上添加文字

要在图片上绘制文字，你需要设置下列参数：

- 你要绘制的文字
- 你要绘制的位置
- 字体类型（通过查看 `cv2.putText()` 的文档找到支持的字体）
- 字体的大小
- 文字的一般属性如颜色，粗细，线条的类型等。为了更好看一点推荐使用 **`linetype=cv2.LINE_AA`**。

在图像上绘制白色的 OpenCV。

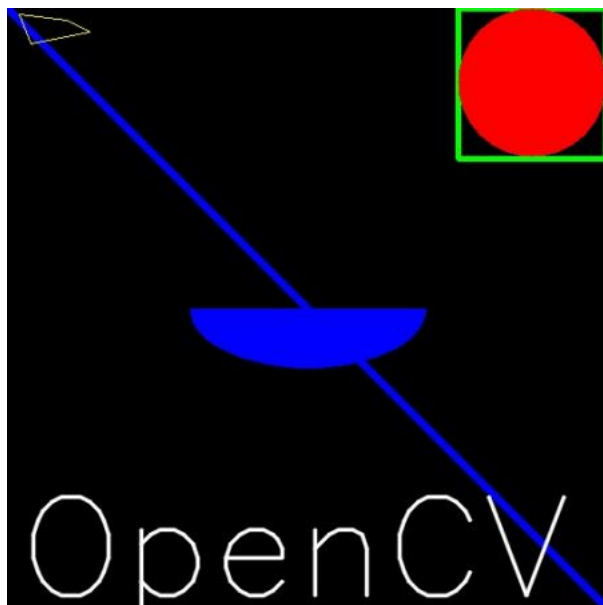
```
1 font=cv2.FONT_HERSHEY_SIMPLEX
2 cv2.putText(img, 'OpenCV', (10, 500), font, 4, (255, 255, 255), 2)
```

**警告：**所有的绘图函数的返回值都是 `None`，所以不能使用 **`img = cv2.line(img, (0,0), (511,511), (255,0,0), 5)`**。

## 结果

下面就是最终结果了，通过你前面几节学到的知识把他显示出来吧。

```
1 winname = 'example'
2 cv2.namedWindow(winname)
3 cv2.imshow(winname, img)
4 cv2.waitKey(0)
5 cv2.destroyAllWindows()
```



## 更多资源

椭圆函数中的角度不是我们的圆形角度。更多细节请查看[讨论](#)

## 练习

尝试使用这些函数挥着 OpenCV 的图标。

## 7 把鼠标当画笔

### 目标

- 学习使用 OpenCV 处理鼠标事件
- 你将要学习的函数是：**cv2.setMouseCallback()**

### 7.1 简单演示

这里我们来创建一个简单的程序，他会在图片上你双击过的位置绘制一个圆圈。

首先我们来创建一个鼠标事件回调函数，但鼠标事件发生是他就会被执行。鼠标事件可以是鼠标上的任何动作，比如左键按下，左键松开，左键双击等。我们可以通过鼠标事件获得与鼠标对应的图片上的坐标。根据这些信息我们可以做任何我们想做的事。你可以通过执行下列代码查看所有被支持的鼠标事件。

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jan 5 11:29:15 2014

@author: duan
"""

import cv2
events=[i for i in dir(cv2) if 'EVENT' in i]
print events
```

所有的鼠标事件回调函数都有一个统一的格式，他们所不同的地方仅仅是被调用后的功能。我们的鼠标事件回调函数只用做一件事：在双击过的地方绘制一个圆圈。下面是代码，不懂的地方可以看看注释。

```

# -*- coding: utf-8 -*-
"""
Created on Sun Jan  5 11:31:53 2014

@author: duan
"""

import cv2
import numpy as np
#mouse callback function

def draw_circle(event,x,y,flags,param):
    if event==cv2.EVENT_LBUTTONDOWN:
        cv2.circle(img,(x,y),100,(255,0,0),-1)

# 创建图像与窗口并将窗口与回调函数绑定
img=np.zeros((512,512,3),np.uint8)
cv2.namedWindow('image')
cv2.setMouseCallback('image',draw_circle)

while(1):
    cv2.imshow('image',img)
    if cv2.waitKey(20)&0xFF==27:
        break
cv2.destroyAllWindows()

```

## 7.2 高级一点的示例

现在我们来创建一个更好的程序。这次我们的程序要完成的任务是根据我们选择的模式在拖动鼠标时绘制矩形或者是圆圈（就像画图程序中一样）。所以我们的回调函数包含两部分，一部分画矩形，一部分画圆圈。这是一个典型的例子他可以帮助我们更好理解与构建人机交互式程序，比如物体跟踪，图像分割等。



```

# -*- coding: utf-8 -*-
"""
Created on Sun Jan 5 11:41:39 2014

@author: duan
"""

import cv2
import numpy as np

# 当鼠标按下时变为 True
drawing=False
# 如果 mode 为 true 绘制矩形。按下'm' 变成绘制曲线。
mode=True
ix,iy=-1,-1

# 创建回调函数
def draw_circle(event,x,y,flags,param):
    global ix,iy,drawing,mode
    # 当按下左键是返回起始位置坐标
    if event==cv2.EVENT_LBUTTONDOWN:
        drawing=True
        ix,iy=x,y
    # 当鼠标左键按下并移动是绘制图形。event 可以查看移动, flag 查看是否按下
    elif event==cv2.EVENT_MOUSEMOVE and flags==cv2.EVENT_FLAG_LBUTTON:
        if drawing==True:
            if mode==True:
                cv2.rectangle(img,(ix,iy),(x,y),(0,255,0),-1)
            else:
                # 绘制圆圈, 小圆点连在一起就成了线, 3 代表了笔画的粗细
                cv2.circle(img,(x,y),3,(0,0,255),-1)
                # 下面注释掉的代码是起始点为圆心, 起点到终点为半径的
                r=int(np.sqrt((x-ix)**2+(y-iy)**2))
                cv2.circle(img,(x,y),r,(0,0,255),-1)
    # 当鼠标松开停止绘画。
    elif event==cv2.EVENT_LBUTTONUP:
        drawing=False
        # if mode==True:
        #     cv2.rectangle(img,(ix,iy),(x,y),(0,255,0),-1)
        # else:
        #     cv2.circle(img,(x,y),5,(0,0,255),-1)

```

下面我们要把这个回调函数与 OpenCV 窗口绑定在一起。在主循环中我们需要将键盘上的“m”键与模式转换绑定在一起。

```
img=np.zeros((512,512,3),np.uint8)
cv2.namedWindow('image')
cv2.setMouseCallback('image',draw_circle)
while(1):
    cv2.imshow('image',img)
    k=cv2.waitKey(1)&0xFF
    if k==ord('m'):
        mode=not mode
    elif k==27:
        break
```

## 更多资源

### 练习

1. 在我们最后的一个练习中，我们绘制的是一个填充的矩形。你可以试着修改代码绘制一个没有填充的矩形。

## 8 用滑动条做调色板

### 目标

- 学会把滑动条绑定到 OpenCV 的窗口
- 你将会学习这些函数: `cv2.getTrackbarPos()`, `cv2.createTrackbar()` 等。

### 8.1 代码示例

现在我们来创建一个简单的程序：通过调节滑动条来设定画板颜色。我们要创建一个窗口来显示显色，还有三个滑动条来设置 B, G, R 的颜色。当我们滑动滚动条是窗口的颜色也会发生相应改变。默认情况下窗口的起始颜色为黑。

`cv2.getTrackbarPos()` 函数的一个参数是滑动条的名字，第二个参数是滑动条被放置窗口的名字，第三个参数是滑动条的默认位置。第四个参数是滑动条的最大值，第五个函数是回调函数，每次滑动条的滑动都会调用回调函数。回调函数通常都会含有一个默认参数，就是滑动条的位置。在本例中这个函数不用做任何事情，我们只需要 **pass** 就可以了。

滑动条的另外一个重要应用就是用作转换按钮。默认情况下 OpenCV 本身不带有按钮函数。所以我们使用滑动条来代替。在我们的程序中，我们要创建一个转换按钮，只有当装换按钮指向 ON 时，滑动条的滑动才有用，否则窗口都是黑的。

```

# -*- coding: utf-8 -*-
"""
Created on Sun Jan  5 13:51:34 2014

@author: duan
"""

import cv2
import numpy as np

def nothing(x):
    pass

# 创建一副黑色图像
img=np.zeros((300,512,3),np.uint8)
cv2.namedWindow('image')

cv2.createTrackbar('R','image',0,255,nothing)
cv2.createTrackbar('G','image',0,255,nothing)
cv2.createTrackbar('B','image',0,255,nothing)

switch='0:OFF\n1:ON'
cv2.createTrackbar(switch,'image',0,1,nothing)

while(1):
    cv2.imshow('image',img)
    k=cv2.waitKey(1)&0xFF
    if k==27:
        break

    r=cv2.getTrackbarPos('R','image')
    g=cv2.getTrackbarPos('G','image')
    b=cv2.getTrackbarPos('B','image')
    s=cv2.getTrackbarPos(switch,'image')

    if s==0:
        img[:]=0
    else:
        img[:]=[b,g,r]

cv2.destroyAllWindows()

```

程序运行效果如下：



## 练习

1. 结合上一节的知识，创建一个画板，可以自选各种颜色的画笔绘画各种图形。

```

# -*- coding: utf-8 -*-
"""
Created on Sun Jan 5 14:59:58 2014

@author: duan
"""

import cv2
import numpy as np

def nothing(x):
    pass

# 当鼠标按下时变为 True
drawing=False
# 如果 mode 为 true 绘制矩形。按下'm' 变成绘制曲线。
mode=True
ix,iy=-1,-1

# 创建回调函数
def draw_circle(event,x,y,flags,param):
    r=cv2.getTrackbarPos('R','image')
    g=cv2.getTrackbarPos('G','image')
    b=cv2.getTrackbarPos('B','image')
    color=(b,g,r)

    global ix,iy,drawing,mode
    # 当按下左键是返回起始位置坐标
    if event==cv2.EVENT_LBUTTONDOWN:
        drawing=True
        ix,iy=x,y
    # 当鼠标左键按下并移动是绘制图形。event 可以查看移动, flag 查看是否按下
    elif event==cv2.EVENT_MOUSEMOVE and flags==cv2.EVENT_FLAG_LBUTTON:
        if drawing==True:
            if mode==True:
                cv2.rectangle(img,(ix,iy),(x,y),color,-1)
            else:
                # 绘制圆圈, 小圆点连在一起就成了线, 3 代表了笔画的粗细
                cv2.circle(img,(x,y),3,color,-1)
                # 下面注释掉的代码是起始点为圆心, 起点到终点为半径的
                r=int(np.sqrt((x-ix)**2+(y-iy)**2))
                cv2.circle(img,(x,y),r,(0,0,255),-1)
        # 当鼠标松开停止绘画。
        elif event==cv2.EVENT_LBUTTONUP:
            drawing=False
            if mode==True:
                cv2.rectangle(img,(ix,iy),(x,y),(0,255,0),-1)
            else:
                cv2.circle(img,(x,y),5,(0,0,255),-1)

img=np.zeros((512,512,3),np.uint8)
cv2.namedWindow('image')
cv2.createTrackbar('R','image',0,255,nothing)
cv2.createTrackbar('G','image',0,255,nothing)
cv2.createTrackbar('B','image',0,255,nothing)
cv2.setMouseCallback('image',draw_circle)
while(1):
    cv2.imshow('image',img)

    k=cv2.waitKey(1)&0xFF
    if k==ord('m'):
        mode=not mode
    elif k==27:
        break

```

## 部分 III

# 核心操作

## 9 图像的基础操作

### 目标

- 获取像素值并修改
- 获取图像的属性（信息）
- 图像的 ROI ()
- 图像通道的拆分及合并

几乎所有这些操作与 Numpy 的关系都比与 OpenCV 的关系更加紧密，因此熟练 Numpy 可以帮助我们写出性能更好的代码。

（示例将会在 Python 终端中展示，因为他们大部分都只有一行代码）

### 9.1 获取并修改像素值

首先我们需要读入一幅图像：

```
import cv2
import numpy as np
img=cv2.imread('/home/duan/workspace/opencv/images/roi.jpg')
```

你可以根据像素的行和列的坐标获取他的像素值。对 BGR 图像而言，返回值为 B, G, R 的值。对灰度图像而言，会返回他的灰度值（亮度？intensity）

```
import cv2
import numpy as np
img=cv2.imread('/home/duan/workspace/opencv/images/roi.jpg')
```

```
px=img[100,100]
print px
blue=img[100,100,0]
print blue

## [57 63 68]
## 57
```

你可以以类似的方式修改像素值。

```
import cv2
import numpy as np
img=cv2.imread('/home/duan/workspace/opencv/images/roi.jpg')
img[100,100]=[255,255,255]
print img[100,100]

## [255 255 255]
```

**警告：**Numpy 是经过优化了的进行快速矩阵运算的软件包。所以我们不推荐逐个获取像素值并修改，这样会很慢，能有矩阵运算就不要用循环。

**注意：**上面提到的方法被用来选取矩阵的一个区域，比如说前 5 行的后 3 列。对于获取每一个像素值，也许使用 Numpy 的 **array.item()** 和 **array.itemset()** 会更好。但是返回值是标量。如果你想获得所有 B, G, R 的值，你需要使用 **array.item()** 分割他们。

获取像素值及修改的更好方法。



```

import cv2
import numpy as np
img=cv2.imread('/home/duan/workspace/opencv/images/roi.jpg')
print img.item(10,10,2)
img.itemset((10,10,2),100)
print img.item(10,10,2)

## 50
## 100

```

## 9.2 获取图像属性

图像的属性包括：行，列，通道，图像数据类型，像素数目等

**img.shape** 可以获取图像的形状。他的返回值是一个包含行数，列数，通道数的元组。

```

import cv2
import numpy as np
img=cv2.imread('/home/duan/workspace/opencv/images/roi.jpg')
print img.shape

## (280, 450, 3)

```

**注意：**如果图像是灰度图，返回值仅有行数和列数。所以通过检查这个返回值就可以知道加载的是灰度图还是彩色图。

**img.size** 可以返回图像的像素数目：

```
import cv2
import numpy as np
img=cv2.imread('/home/duan/workspace/opencv/images/roi.jpg')
print img.size

## 378000
```

**img.dtype** 返回的是图像的数据类型.

```
import cv2
import numpy as np
img=cv2.imread('/home/duan/workspace/opencv/images/roi.jpg')
print img.dtype

## uint8
```

注意：在除虫（debug）时 **img.dtype** 非常重要。因为在 OpenCV-Python 代码中经常出现数据类型的不一致。

### 9.3 图像 ROI

有时你需要对一幅图像的特定区域进行操作。例如我们要检测一副图像中眼睛的位置，我们首先应该在图像中找到脸，再在脸的区域中找眼睛，而不是直接在一幅图像中搜索。这样会提高程序的准确性和性能。

ROI 也是使用 Numpy 索引来获得的。现在我们选择球的部分并把他拷贝到图像的其他区域。

```
import cv2
import numpy as np
img=cv2.imread('/home/duan/workspace/opencv/images/roi.jpg')
```

```
ball=img[280:340,330:390]  
img[273:333,100:160]=ball
```

看看结果吧：



## 9.4 拆分及合并图像通道

有时我们需要对 BGR 三个通道分别进行操作。这是你就需要把 BGR 拆分成单个通道。有时你需要把独立通道的图片合并成一个 BGR 图像。你可以这样做：

```
import cv2  
import numpy as np  
img=cv2.imread('/home/duan/workspace/opencv/images/roi.jpg')  
b,g,r=cv2.split(img)  
img=cv2.merge(b,g,r)
```

或者：

```
import cv2  
import numpy as np
```

```
img=cv2.imread('/home/duan/workspace/opencv/images/roi.jpg')
b=img[:, :, 0]
```

假如你想使所有像素的红色通道值都为 0，你不必先拆分再赋值。你可以直接使用 Numpy 索引，这会更快。

```
import cv2
import numpy as np
img=cv2.imread('/home/duan/workspace/opencv/images/roi.jpg')
img[:, :, 2]=0
```

**警告：** `cv2.split()` 是一个比较耗时的操作。只有真正需要时才用它，能用 Numpy 索引就尽量用。

## 9.5 为图像扩边（填充）

如果你想在图像周围创建一个边，就像相框一样，你可以使用 `cv2.copyMakeBorder()` 函数。这经常在卷积运算或 0 填充时被用到。这个函数包括如下参数：

- **src** 输入图像
- **top, bottom, left, right** 对应边界的像素数目。
- **borderType** 要添加那种类型的边界，类型如下
  - **cv2.BORDER\_CONSTANT** 添加有颜色的常数值边界，还需要下一个参数 (**value**)。
  - **cv2.BORDER\_REFLECT** 边界元素的镜像。比如: fedcba|abcde-fgh|hgfedcb
  - **cv2.BORDER\_REFLECT\_101** or **cv2.BORDER\_DEFAULT** 跟上面一样，但稍作改动。例如: gfedcb|abcdefgh|gfedcba
  - **cv2.BORDER\_REPLICATE** 重复最后一个元素。例如: aaaaaa|abcdefgh|hhhhhhh

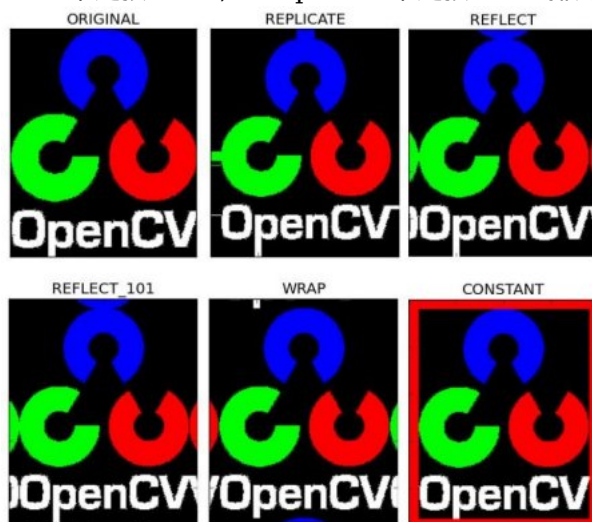
- **cv2.BORDER\_WRAP** 不知道怎么说了, 就像这样: cdefgh|  
abcdefgh|abcdefgh

- **value** 边界颜色, 如果边界的类型是 **cv2.BORDER\_CONSTANT**

为了更好的理解这几种类型请看下面的演示程序。

```
# -*- coding: utf-8 -*-  
"""  
Created on Sun Jan 5 19:03:28 2014  
  
@author: duan  
"""  
  
import cv2  
import numpy as np  
from matplotlib import pyplot as plt  
  
BLUE=[255,0,0]  
  
img1=cv2.imread('opencv_logo.png')  
  
replicate = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_REPLICATE)  
reflect = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_REFLECT)  
reflect101 = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_REFLECT_101)  
wrap = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_WRAP)  
constant= cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_CONSTANT,value=BLUE)  
  
plt.subplot(231),plt.imshow(img1,'gray'),plt.title('ORIGINAL')  
plt.subplot(232),plt.imshow(replicate,'gray'),plt.title('REPLICATE')  
plt.subplot(233),plt.imshow(reflect,'gray'),plt.title('REFLECT')  
plt.subplot(234),plt.imshow(reflect101,'gray'),plt.title('REFLECT_101')  
plt.subplot(235),plt.imshow(wrap,'gray'),plt.title('WRAP')  
plt.subplot(236),plt.imshow(constant,'gray'),plt.title('CONSTANT')  
  
plt.show()
```

结果如下 (由于是使用 matplotlib 绘制, 所以交换 R 和 B 的位置, OpenCV 中是按 BGR, matplotlib 中是按 RGB 排列):



## 10 图像上的算术运算

### 目标

- 学习图像上的算术运算，加法，减法，位运算等。
- 我们将要学习的函数与有：cv2.add(), cv2.addWeighted() 等。

### 10.1 图像加法

你可以使用函数 cv2.add() 将两幅图像进行加法运算，当然也可以直接使用 numpy, res=img1+img2。两幅图像的大小，类型必须一致，或者第二个图像可以使一个简单的标量值。

**注意：**OpenCV 中的加法与 Numpy 的加法是有所不同的。OpenCV 的加法是一种饱和操作，而 Numpy 的加法是一种模操作。

例如下面的两个例子：

```
1 x = np.uint8([250])
2 y = np.uint8([10])
3 print cv2.add(x,y) # 250+10 = 260 => 255
4 [[255]]
5 print x+y          # 250+10 = 260 % 256 = 4
6 [4]
```

这种差别在你对两幅图像进行加法时会更加明显。OpenCV 的结果会更好一点。所以我们尽量使用 OpenCV 中的函数。

### 10.2 图像混合

这其实也是加法，但是不同的是两幅图像的权重不同，这就会给人一种混合或者透明的感觉。图像混合的计算公式如下：

$$g(x) = (1 - \alpha) f_0(x) + \alpha f_1(x)$$

通过修改  $\alpha$  的值 ( $0 \rightarrow 1$ )，可以实现非常酷的混合。

现在我们把两幅图混合在一起。第一幅图的权重是 0.7，第二幅图的权重是 0.3。函数 cv2.addWeighted() 可以按下面的公式对图片进行混合操作。

$$dst = \alpha \cdot img1 + \beta \cdot img2 + \gamma$$

这里  $\gamma$  的取值为 0。

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jan 5 20:24:50 2014

@author: duan
"""

import cv2
import numpy as np

img1=cv2.imread('ml.png')
img2=cv2.imread('opencv_logo.jpg')

dst=cv2.addWeighted(img1,0.7,img2,0.3,0)

cv2.imshow('dst',dst)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

下面就是结果：



### 10.3 按位运算

这里包括的按位操作有：AND，OR，NOT，XOR 等。当我们提取图像的一部分，选择非矩形 ROI 时这些操作会很有用（下一章你就会明白）。下面的例子就是教给我们如何改变一幅图的特定区域。

我想把 OpenCV 的标志放到另一幅图像上。如果我使用加法，颜色会改变，如果使用混合，会得到透明效果，但是我不想要透明。如果他是矩形我可以象上一章那样使用 ROI。但是他不是矩形。但是我们可以通过下面的按位运算实现：

```

# -*- coding: utf-8 -*-
"""
Created on Sun Jan 5 20:34:30 2014

@author: duan
"""

import cv2
import numpy as np

# 加载图像
img1 = cv2.imread('roi.jpg')
img2 = cv2.imread('opencv_logo.png')

# I want to put logo on top-left corner, So I create a ROI
rows,cols,channels = img2.shape
roi = img1[0:rows, 0:cols ]

# Now create a mask of logo and create its inverse mask also
img2gray = cv2.cvtColor(img2,cv2.COLOR_BGR2GRAY)
ret, mask = cv2.threshold(img2gray, 175, 255, cv2.THRESH_BINARY)
mask_inv = cv2.bitwise_not(mask)

# Now black-out the area of logo in ROI
# 取 roi 中不为零的值对应的像素的值, 其他值为 0
# 注意这里必须有 mask=mask 或者 mask=mask_inv, 其中的 mask= 不能忽略
img1_bg = cv2.bitwise_and(roi,roi,mask = mask)
# 取 roi 中与 mask_inv 中不为零的值对应的像素的值, 其他值为 0。
# Take only region of logo from logo image.
img2_fg = cv2.bitwise_and(img2,img2,mask = mask_inv)

# Put logo in ROI and modify the main image
dst = cv2.add(img1_bg,img2_fg)
img1[0:rows, 0:cols ] = dst

cv2.imshow('res',img1)
cv2.waitKey(0)
cv2.destroyAllWindows()

```

结果如下。左面的图像是我们创建的掩码。右边的是最终结果。为了帮助大家理解我把上面程序的中间结果也显示了出来，特别是 `img1_bg` 和 `img2_fg`。





## 更多资源

### 练习

1. 创建一个幻灯片用来演示一幅图如何平滑的转换成另一幅图（使用函数 `cv2.addWeighted`）

## 11 程序性能检测及优化

### 目标

在图像处理中你每秒钟都要做大量的运算，所以你的程序不仅要能给出正确的结果，同时还必须要快。所以这节我们将要学习：

- 检测程序的效率
- 一些能够提高程序效率的技巧
- 你要学到的函数有：**cv2.getTickCount**, **cv2.getTickFrequency** 等

除了 OpenCV，Python 也提供了一个叫 **time** 的模块，你可以用它来测量程序的运行时间。另外一个叫做 **profile** 的模块会帮你得到一份关于你的程序的详细报告，其中包含了代码中每个函数运行需要的时间，以及每个函数被调用的次数。如果你正在使用 IPython 的话，所有这些特点都被以一种用户友好的方式整合在一起了。我们会学习几个重要的，要想学到更加详细的知识就打开更多资源中的链接吧。

### 11.1 使用 OpenCV 检测程序效率

**cv2.getTickCount** 函数返回从参考点到这个函数被执行的时钟数。所以当你在一个函数执行前后都调用它的话，你就会得到这个函数的执行时间（时钟数）。

**cv2.getTickFrequency** 返回时钟频率，或者说每秒钟的时钟数。所以你可以按照下面的方式得到一个函数运行了多少秒：

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 9 21:10:40 2014

@author: duan
"""

import cv2
import numpy as np

e1 = cv2.getTickCount()
# your code execution
e2 = cv2.getTickCount()
time = (e2 - e1) / cv2.getTickFrequency()
```

我们将会用下面的例子演示。下面的例子是用窗口大小不同（5，7，9）的核函数来做中值滤波：

```

# -*- coding: utf-8 -*-
"""
Created on Thu Jan  9 21:06:38 2014

@author: duan
"""
import cv2
import numpy as np

img1 = cv2.imread('roi.jpg')

e1 = cv2.getTickCount()
for i in xrange(5,49,2):
    img1 = cv2.medianBlur(img1,i)
e2 = cv2.getTickCount()
t = (e2 - e1)/cv2.getTickFrequency()
print t

# Result I got is 0.521107655 seconds

```

**注意：**你也可以中 **time** 模块实现上面的功能。但是要用的函数是 **time.time()** 而不是 **cv2.getTickCount**。比较一下这两个结果的差别吧。

## 11.2 OpenCV 中的默认优化

OpenCV 中的很多函数都被优化过（使用 SSE2，AVX 等）。也包含一些没有被优化的代码。如果我们的系统支持优化的话要尽量利用只一点。在编译时优化是被默认开启的。因此 OpenCV 运行的就是优化后的代码，如果你把优化关闭的话就只能执行低效的代码了。你可以使用函数 **cv2.useOptimized()** 来查看优化是否被开启了，使用函数 **cv2.setUseOptimized()** 来开启优化。让我们来看一个简单的例子吧。

```

# -*- coding: utf-8 -*-
"""
Created on Thu Jan  9 21:08:41 2014

@author: duan
"""
import cv2
import numpy as np

# check if optimization is enabled
In [5]: cv2.useOptimized()
Out[5]: True

In [6]: %timeit res = cv2.medianBlur(img,49)
10 loops, best of 3: 34.9 ms per loop

# Disable it
In [7]: cv2.setUseOptimized(False)

In [8]: cv2.useOptimized()
Out[8]: False

In [9]: %timeit res = cv2.medianBlur(img,49)
10 loops, best of 3: 64.1 ms per loop

```

看见了吗，优化后中值滤波的速度是原来的两倍。如果你查看源代码的话，你会发现中值滤波是被 SIMD 优化的。所以你可以在代码的开始处开启优化（你要记住优化是默认开启的）。

### 11.3 在 IPython 中检测程序效率

有时你需要比较两个相似操作的效率，这时你可以使用 IPython 为你提供的魔法命令 **%time**。他会让代码运行好几次从而得到一个准确的（运行）时间。它也可以被用来测试单行代码的。

例如，你知道下面这同一个数学运算用哪种行式的代码会执行的更快吗？

```

x = 5; y = x * 2
x = 5; y = x * x
x = np.uint([5]); y = x * x
y = np.square(x)

```

我们可以在 IPython 的 Shell 中使用魔法命令找到答案。

```

# -*- coding: utf-8 -*-
"""
Created on Thu Jan  9 21:10:40 2014

@author: duan
"""

import cv2
import numpy as np

In [10]: x = 5

In [11]: %timeit y=x**2
10000000 loops, best of 3: 73 ns per loop

In [12]: %timeit y=x*x
10000000 loops, best of 3: 58.3 ns per loop

In [15]: z = np.uint8([5])

In [17]: %timeit y=z*z
1000000 loops, best of 3: 1.25 us per loop

In [19]: %timeit y=np.square(z)
1000000 loops, best of 3: 1.16 us per loop

```

竟然是第一种写法，它居然比 Nump 快了 20 倍。如果考虑到数组构建的话，能达到 100 倍的差。

**注意：**Python 的标量计算比 Nump 的标量计算要快。对于仅包含一两个元素的操作 Python 标量比 Numpy 的数组要快。但是当数组稍微大一点时 Numpy 就会胜出了。

我们来再看几个例子。我们来比较一下 **cv2.countNonZero()** 和 **np.count\_nonzero()**。

```

# -*- coding: utf-8 -*-
"""
Created on Thu Jan  9 21:10:40 2014

@author: duan
"""

import cv2
import numpy as np

In [35]: %timeit z = cv2.countNonZero(img)
100000 loops, best of 3: 15.8 us per loop

In [36]: %timeit z = np.count_nonzero(img)
1000 loops, best of 3: 370 us per loop

```

看见了吧，OpenCV 的函数是 Numpy 函数的 25 倍。

**注意：**一般情况下 OpenCV 的函数要比 Numpy 函数快。所以对于相同的操作最好使用 OpenCV 的函数。当然也有例外，尤其是当使用 Numpy 对视图（而非复制）进行操作时。

## 11.4 更多 IPython 的魔法命令

还有几个魔法命令可以用来检测程序的效率，profiling, line profiling, 内存使用等。他们都有完善的文档。所以这里只提供了超链接。感兴趣的可以自己学习一下。

## 11.5 效率优化技术

有些技术和编程方法可以让我们最大的发挥 Python 和 Numpy 的威力。我们这里仅仅提一下相关的，你可以通过超链接查找更多详细信息。我们要说的最重要的一点是：首先用简单的方式实现你的算法（结果正确最重要），当结果正确后，再使用上面的提到的方法找到程序的瓶颈来优化它。

1. 尽量避免使用循环，尤其双层三层循环，它们天生就是非常慢的。
2. 算法中尽量使用向量操作，因为 Numpy 和 OpenCV 都对向量操作进行了优化。
3. 利用高速缓存一致性。
4. 没有必要的话就不要复制数组。使用视图来代替复制。数组复制是非常浪费资源的。

就算进行了上述优化，如果你的程序还是很慢，或者说大的训诂不可避免的话，你你应该尝试使用其他的包，比如说 Cython，来加速你的程序。

## 更多资源

1. [Python Optimization Techniques](#)
2. [Scipy Lecture Notes - Advanced Numpy](#)
3. [Timing and Profiling in IPython](#)

## 练习

## 12 OpenCV 中的数学工具



## 部分 IV

# OpenCV 中的图像处理

## 13 颜色空间转换

### 目标

- 你将学习如何对图像进行颜色空间转换，比如从 BGR 到灰度图，或者从 BGR 到 HSV 等。
- 我还没要创建一个程序用来从一幅图像中获取某个特定颜色的物体。
- 我们将要学习的函数有：**cv2.cvtColor()**，**cv2.inRange()** 等。

### 13.1 转换颜色空间

在 OpenCV 中有超过 150 中进行颜色空间转换的方法。但是你以后就会发现我们经常用到的也就两种：BGR↔Gray 和 BGR↔HSV。

我们要用到的函数是：**cv2.cvtColor(input\_image, flag)**，其中 flag 就是转换类型。

对于 BGR↔Gray 的转换，我们要使用的 flag 就是 **cv2.COLOR\_BGR2GRAY**。同样对于 BGR↔HSV 的转换，我们用的 flag 就是 **cv2.COLOR\_BGR2HSV**。你还可以通过下面的命令得到所有可用的 flag。

```
# -*- coding: utf-8 -*-
"""
Created on Fri Jan 10 20:23:26 2014

@author: duan
"""

import cv2
flags=[i for i in dir(cv2) if i.startswith('COLOR_')]
print flags
```

**注意：**在 OpenCV 的 HSV 格式中，H（色彩/色度）的取值范围是 [0, 179]，S（饱和度）的取值范围 [0, 255]，V（亮度）的取值范围 [0, 255]。但是不同的软件使用的值可能不同。所以当你需要拿 OpenCV 的 HSV 值与其他软件的 HSV 值进行对比时，一定要记得归一化。

## 13.2 物体跟踪

现在我们知道怎样将一幅图像从 BGR 转换到 HSV 了，我们可以利用这一点来提取带有某个特定颜色的物体。在 HSV 颜色空间中要比在 BGR 空间中更容易表示一个特定颜色。在我们的程序中，我们要提取的是一个蓝色的物体。下面就是我们要做的几步：

- 从视频中获取每一帧图像
- 将图像转换到 HSV 空间
- 设置 HSV 阈值到蓝色范围。
- 获取蓝色物体，当然我们还可以做其他任何我们想做的事，比如：在蓝色物体周围画一个圈。

下面就是我们的代码：

```

# -*- coding: utf-8 -*-
"""
Created on Fri Jan 10 20:25:00 2014

@author: duan
"""

import cv2
import numpy as np

cap=cv2.VideoCapture(0)

while(1):

    # 获取每一帧
    ret,frame=cap.read()

    # 转换到 HSV
    hsv=cv2.cvtColor(frame,cv2.COLOR_BGR2HSV)

    # 设定蓝色的阈值
    lower_blue=np.array([110,50,50])
    upper_blue=np.array([130,255,255])

    # 根据阈值构建掩模
    mask=cv2.inRange(hsv,lower_blue,upper_blue)

    # 对原图像和掩模进行位运算
    res=cv2.bitwise_and(frame,frame,mask=mask)

    # 显示图像
    cv2.imshow('frame',frame)
    cv2.imshow('mask',mask)
    cv2.imshow('res',res)
    k=cv2.waitKey(5)&0xFF
    if k==27:
        break
# 关闭窗口
cv2.destroyAllWindows()

```

下图显示了追踪蓝色物体的结果：



**注意：**图像中仍然有一些噪音，我们会在后面的章节中介绍如何消减噪音。

**注意：**这是物体跟踪中最简单的方法。当你学习了轮廓之后，你就会学到更多相关知识，那是你就可以找到物体的重心，并根据重心来跟踪物体，仅仅在摄像头前挥挥手就可以画出同的图形，或者其他更有趣的事。

### 13.3 怎样找到要跟踪对象的 HSV 值？

这是我在[stackoverflow.com](https://stackoverflow.com)上遇到的最普遍的问题。其实这真的很简单，函数 `cv2.cvtColor()` 也可以用到这里。但是现在你要传入的参数是（你想要的）BGR 值而不是一副图。例如，我们要找到绿色的 HSV 值，我们只需在终端输入以下命令：

```
# -*- coding: utf-8 -*-
"""
Created on Fri Jan 10 20:34:29 2014

@author: duan
"""

import cv2
import numpy as np

green=np.uint8([0,255,0])
hsv_green=cv2.cvtColor(green,cv2.COLOR_BGR2HSV)

error: /builddir/build/BUILD/opencv-2.4.6.1/
modules/imgproc/src/color.cpp:3541:
error: (-215) (scn == 3 || scn == 4) && (depth == CV_8U || depth == CV_32F)
in function cvtColor

#scn (the number of channels of the source),
#i.e. self.img.channels(), is neither 3 nor 4.
#
#depth (of the source),
#i.e. self.img.depth(), is neither CV_8U nor CV_32F.
# 所以不能用 [0,255,0], 而要用 [[[0,255,0]]]
# 这里的三层括号应该分别对应于 cvArray, cvMat, IplImage

green=np.uint8([[[0,255,0]]])
hsv_green=cv2.cvtColor(green,cv2.COLOR_BGR2HSV)
print hsv_green
[[[60 255 255]]]
```

现在你可以分别用 `[H-100, 100, 100]` 和 `[H+100, 255, 255]` 做上下阈值。除了这个方法之外，你可以使用任何其他图像编辑软件（例如 GIMP）或者在线转换软件找到相应的 HSV 值，但是最后别忘了调节 HSV 的范围。

## 更多资源

### 练习

1. 尝试同时提取多个不同的颜色物体，比如同时提取红，蓝，绿三个不同颜色的物体。

## 14 几何变换

### 目标

- 学习对图像进行各种几何变换，例如移动，旋转，仿射变换等。
- 将要学到的函数有：**cv2.getPerspectiveTransform**。

### 变换

OpenCV 提供了两个变换函数，**cv2.warpAffine** 和 **cv2.warpPerspective**，使用这两个函数你可以实现所有类型的变换。**cv2.warpAffine** 接收的参数是  $2 \times 3$  的变换矩阵，而 **cv2.warpPerspective** 接收的参数是  $3 \times 3$  的变换矩阵。

#### 14.1 扩展缩放

扩展缩放只是改变图像的尺寸大小。OpenCV 提供的函数 **cv2.resize()** 可以实现这个功能。图像的尺寸可以自己手动设置，你也可以指定缩放因子。我们可以选择使用不同的插值方法。在缩放时我们推荐使用 **cv2.INTER\_AREA**，在扩展时我们推荐使用 **cv2.INTER\_CUBIC**（慢）和 **cv2.INTER\_LINEAR**。默认情况下所有改变图像尺寸大小的操作使用的插值方法都是 **cv2.INTER\_LINEAR**。你可以使用下面任意一种方法改变图像的尺寸：

```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 11 09:32:51 2014

@author: duan
"""

import cv2
import numpy as np

img=cv2.imread('messi5.jpg')
# 下面的 None 本应该是输出图像的尺寸，但是因为后边我们设置了缩放因子
# 因此这里为 None
res=cv2.resize(img,None,fx=2,fy=2,interpolation=cv2.INTER_CUBIC)

#OR
# 这里呢，我们直接设置输出图像的尺寸，所以不用设置缩放因子
height,width=img.shape[:2]
res=cv2.resize(img,(2*width,2*height),interpolation=cv2.INTER_CUBIC)

while(1):
    cv2.imshow('res',res)
    cv2.imshow('img',img)

    if cv2.waitKey(1) & 0xFF == 27:
        break
cv2.destroyAllWindows()

```

**Resize(src, dst, interpolation=CV\_INTER\_LINEAR)**

## 14.2 平移

平移就是将对象换一个位置。如果你要沿 (x, y) 方向移动，移动的距离是 ( $t_x$ ,  $t_y$ )，你可以以下面的方式构建移动矩阵：

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$$

你可以使用 Numpy 数组构建这个矩阵（数据类型是 np.float32），然后把它传给函数 **cv2.warpAffine()**。看看下面这个例子吧，它被移动了 (100,50) 个像素。

```

# -*- coding: utf-8 -*-
"""
Created on Fri Jan 10 20:25:00 2014

@author: duan
"""

import cv2
import numpy as np

cap=cv2.VideoCapture(0)

while(1):

    # 获取每一帧
    ret,frame=cap.read()

    # 转换到 HSV
    hsv=cv2.cvtColor(frame,cv2.COLOR_BGR2HSV)

    # 设定蓝色的阈值
    lower_blue=np.array([110,50,50])
    upper_blue=np.array([130,255,255])

    # 根据阈值构建掩模
    mask=cv2.inRange(hsv,lower_blue,upper_blue)

    # 对原图像和掩模进行位运算
    res=cv2.bitwise_and(frame,frame,mask=mask)

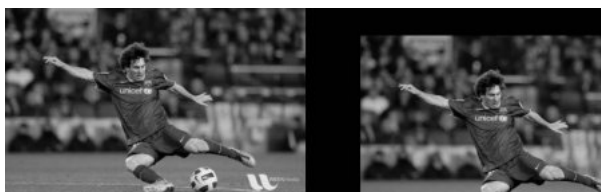
    # 显示图像
    cv2.imshow('frame',frame)
    cv2.imshow('mask',mask)
    cv2.imshow('res',res)
    k=cv2.waitKey(5)&0xFF
    if k==27:
        break
# 关闭窗口
cv2.destroyAllWindows()

```

**警告：**函数 cv2.warpAffine() 的第三个参数的是输出图像的大小，它的格式应该是图像的（宽，高）。应该记住的是图像的宽对应的是列数，高对应的是行数。

下面就是结果：





### 14.3 旋转

对一个图像旋转角度  $\theta$ , 需要使用到下面形式的旋转矩阵。

$$M = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

但是 OpenCV 允许你在任意地方进行旋转, 但是旋转矩阵的形式应该修改为

$$\begin{bmatrix} \alpha & \beta & (1 - \alpha) \cdot center.x - \beta \cdot center.y \\ -\beta & \alpha & \beta \cdot center.x + (1 - \alpha) \cdot center.x \end{bmatrix}$$

其中:

$$\alpha = scale \cdot \cos \theta$$

$$\beta = scale \cdot \sin \theta$$

为了构建这个旋转矩阵, OpenCV 提供了一个函数: **cv2.getRotationMatrix2D**。下面的例子是在不缩放的情况下将图像旋转 90 度。

```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 11 10:06:28 2014

@author: duan
"""

import cv2
import numpy as np

img=cv2.imread('messi5.jpg',0)

rows,cols=img.shape

# 这里的第一个参数为旋转中心，第二个为旋转角度，第三个为旋转后的缩放因子
# 可以通过设置旋转中心，缩放因子，以及窗口大小来防止旋转后超出边界的问题
M=cv2.getRotationMatrix2D((cols/2,rows/2),45,0.6)

# 第三个参数是输出图像的尺寸中心
dst=cv2.warpAffine(img,M,(2*cols,2*rows))
while(1):
    cv2.imshow('img',dst)
    if cv2.waitKey(1)&0xFF==27:
        break
cv2.destroyAllWindows()

```

下面是结果。



## 14.4 仿射变换

在仿射变换中，原图中所有的平行线在结果图像中同样平行。为了创建这个矩阵我们需要从原图像中找到三个点以及他们在输出图像中的位置。然后 **cv2.getAffineTransform** 会创建一个  $2 \times 3$  的矩阵，最后这个矩阵会被传给函数 **cv2.warpAffine**。

来看看下面的例子，以及我选择的点（被标记为绿色的点）

```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 11 10:41:52 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

img=cv2.imread('drawing.png')
rows,cols,ch=img.shape

pts1=np.float32([[50,50],[200,50],[50,200]])
pts2=np.float32([[10,100],[200,50],[100,250]])

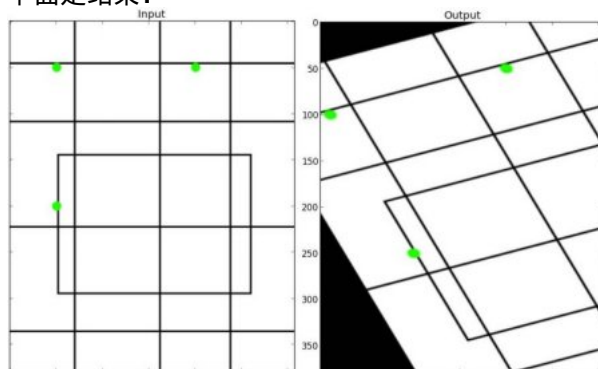
M=cv2.getAffineTransform(pts1,pts2)

dst=cv2.warpAffine(img,M,(cols,rows))

plt.subplot(121,plt.imshow(img),plt.title('Input'))
plt.subplot(121,plt.imshow(dst),plt.title('Output'))
plt.show()

```

下面是结果：



## 14.5 透视变换

对于视角变换，我们需要一个  $3 \times 3$  变换矩阵。在变换前后直线还是直线。要构建这个变换矩阵，你需要在输入图像上找 4 个点，以及他们在输出图像上对应的位置。这四个点中的任意三个都不能共线。这个变换矩阵可以有函数 **cv2.getPerspectiveTransform()** 构建。然后把这个矩阵传给函数 **cv2.warpPerspective**。

代码如下：

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jan 11 10:51:19 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

img=cv2.imread('sudokusmall.png')
rows,cols,ch=img.shape

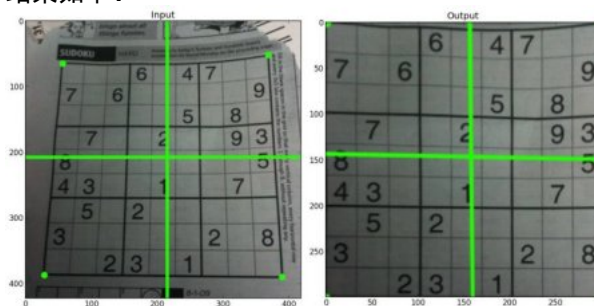
pts1 = np.float32([[56,65],[368,52],[28,387],[389,390]])
pts2 = np.float32([[0,0],[300,0],[0,300],[300,300]])

M=cv2.getPerspectiveTransform(pts1,pts2)

dst=cv2.warpPerspective(img,M,(300,300))

plt.subplot(121,plt.imshow(img),plt.title('Input'))
plt.subplot(121,plt.imshow(dst),plt.title('Output'))
plt.show()
```

结果如下：



## 更多资源

“Computer Vision: Algorithms and Applications”, Richard Szeliski

## 练习

## 15 图像阈值

### 目标

- 本节你将学到简单阈值，自适应阈值，Otsu's 二值化等
- 将要学习的函数有 `cv2.threshold`，`cv2.adaptiveThreshold` 等。

### 15.1 简单阈值

与名字一样，这种方法非常简单。但像素值高于阈值时，我们给这个像素赋予一个新值（可能是白色），否则我们给它赋予另外一种颜色（也许是黑色）。这个函数就是 `cv2.threshold()`。这个函数的第一个参数就是原图像，原图像应该是灰度图。第二个参数就是用来对像素值进行分类的阈值。第三个参数就是当像素值高于（有时是小于）阈值时应该被赋予的新的像素值。OpenCV 提供了多种不同的阈值方法，这是有第四个参数来决定的。这些方法包括：

- `cv2.THRESH_BINARY`
- `cv2.THRESH_BINARY_INV`
- `cv2.THRESH_TRUNC`
- `cv2.THRESH_TOZERO`
- `cv2.THRESH_TOZERO_INV`

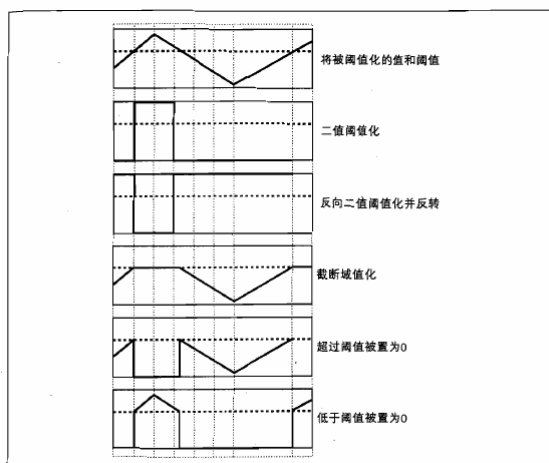


图 5-23：在 `cvThreshold()` 中不同阈值类型的操作结果。每个图表的水平虚线代表应用到最上方图的阈值，5 种阈值类型的操作结果列于其后 【136】

上图摘选自《学习 OpenCV》中文版，其实这些在文档中都有详细介绍了，你也可以直接查看文档。

这个函数有两个返回值，第一个为 `retVal`，我们后面会解释。第二个就是阈值化之后的结果图像了。

代码:

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jan 11 14:12:37 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

img=cv2.imread('gradient.png',0)
ret,thresh1=cv2.threshold(img,127,255,cv2.THRESH_BINARY)
ret,thresh2=cv2.threshold(img,127,255,cv2.THRESH_BINARY_INV)
ret,thresh3=cv2.threshold(img,127,255,cv2.THRESH_TRUNC)
ret,thresh4=cv2.threshold(img,127,255,cv2.THRESH_TOZERO)
ret,thresh5=cv2.threshold(img,127,255,cv2.THRESH_TOZERO_INV)

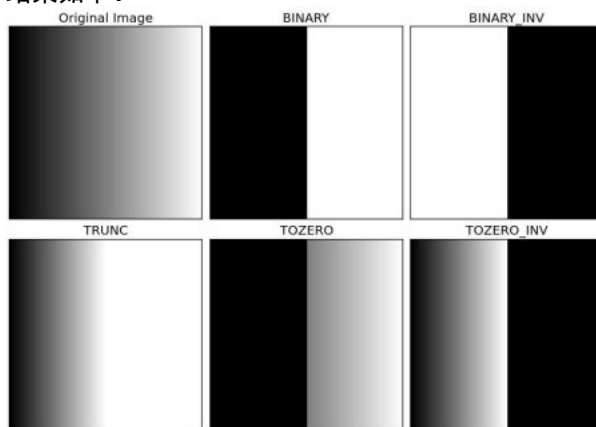
titles = ['Original Image','BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO_INV']
images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]

for i in xrange(6):
    plt.subplot(2,3,i+1),plt.imshow(images[i],'gray')
    plt.title(titles[i])
    plt.xticks([],plt.yticks([]))

plt.show()
```

注意：为了同时在一个窗口中显示多个图像，我们使用函数 `plt.subplot()`。你可以通过查看 Matplotlib 的文档获得更多详细信息。

结果如下：



## 15.2 自适应阈值

在前面的部分我们使用的是全局阈值，整幅图像采用同一个数作为阈值。当时这种方法并不适应与所有情况，尤其是当同一幅图像上的不同部分的具有不同亮度时。这种情况下我们需要采用自适应阈值。此时的阈值是根据图像上的每一个小区域计算与其对应的阈值。因此在同一幅图像上的不同区域采用的是不同的阈值，从而使我们能在亮度不同的情况下得到更好的结果。

这种方法需要我们指定三个参数，返回值只有一个。

- **Adaptive Method**- 指定计算阈值的方法。
  - **cv2.ADPTIVE\_THRESH\_MEAN\_C**: 阈值取自相邻区域的平均值
  - **cv2.ADPTIVE\_THRESH\_GAUSSIAN\_C**: 阈值取值相邻区域的加权和，权重为一个高斯窗口。
- **Block Size** - 邻域大小（用来计算阈值的区域大小）。
- **C** - 这就是是一个常数，阈值就等于的平均值或者加权平均值减去这个常数。

我们使用下面的代码来展示简单阈值与自适应阈值的差别：

```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 11 14:19:29 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('dave.jpg',0)
# 中值滤波
img = cv2.medianBlur(img,5)

ret,th1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
#11 为 Block size, 2 为 C 值
th2 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_MEAN_C,\
                             cv2.THRESH_BINARY,11,2)
th3 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,\
                             cv2.THRESH_BINARY,11,2)

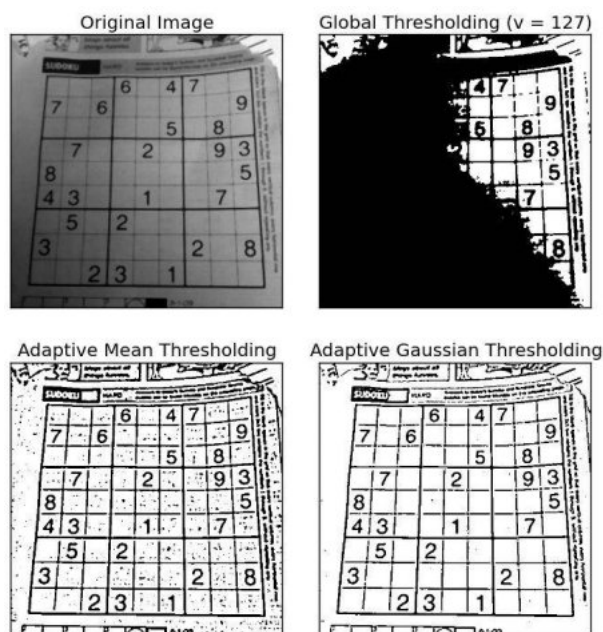
titles = ['Original Image', 'Global Thresholding (v = 127)',
          'Adaptive Mean Thresholding', 'Adaptive Gaussian Thresholding']
images = [img, th1, th2, th3]

for i in xrange(4):
    plt.subplot(2,2,i+1),plt.imshow(images[i],'gray')
    plt.title(titles[i])
    plt.xticks([],plt.yticks([]))
plt.show()

```

结果：





### 15.3 Otsu's 二值化

在第一部分中我们提到过 **retVal**，当我们使用 Otsu 二值化时会用到它。那么它到底是什么呢？

在使用全局阈值时，我们就是随便给了一个数来做阈值，那我们怎么知道我们选取的这个数的好坏呢？答案就是不停的尝试。如果是一副双峰图像（简单来说双峰图像是指图像直方图中存在两个峰）呢？我们岂不是应该在两个峰之间的峰谷选一个值作为阈值？这就是 Otsu 二值化要做的。简单来说就是对一副双峰图像自动根据其直方图计算出一个阈值。（对于非双峰图像，这种方法得到的结果可能会不理想）。

这里用到到的函数还是 **cv2.threshold()**，但是需要多传入一个参数（flag）：**cv2.THRESH\_OTSU**。这时要把阈值设为 0。然后算法会找到最优阈值，这个最优阈值就是返回值 **retVal**。如果不使用 Otsu 二值化，返回的 **retVal** 值与设定的阈值相等。

下面的例子中，输入图像是一副带有噪声的图像。第一种方法，我们设 127 为全局阈值。第二种方法，我们直接使用 Otsu 二值化。第三种方法，我们首先使用一个 5x5 的高斯核除去噪音，然后再使用 Otsu 二值化。看看噪音去除对结果的影响有多大吧。

```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 11 14:25:38 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('noisy2.png',0)

# global thresholding
ret1,th1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)

# Otsu's thresholding
ret2,th2 = cv2.threshold(img,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)

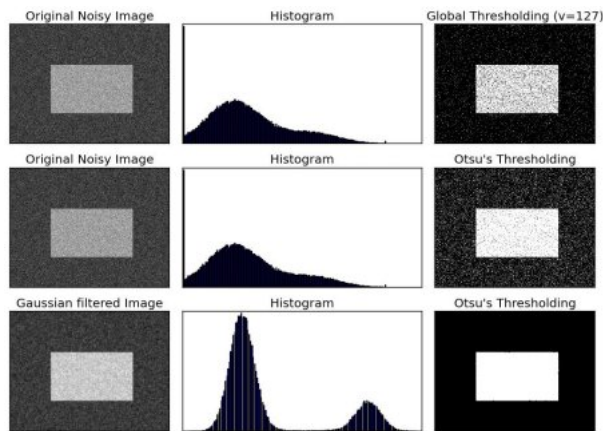
# Otsu's thresholding after Gaussian filtering
# (5,5) 为高斯核的大小, 0 为标准差
blur = cv2.GaussianBlur(img,(5,5),0)
# 阈值一定要设为 0 !
ret3,th3 = cv2.threshold(blur,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)

# plot all the images and their histograms
images = [img, 0, th1,
          img, 0, th2,
          blur, 0, th3]
titles = ['Original Noisy Image','Histogram','Global Thresholding (v=127)',
          'Original Noisy Image','Histogram',"Otsu's Thresholding",
          'Gaussian filtered Image','Histogram',"Otsu's Thresholding"]

# 这里使用了 pyplot 中画直方图的方法, plt.hist, 要注意的是它的参数是一维数组
# 所以这里使用了 (numpy) ravel 方法, 将多维数组转换成一维, 也可以使用 flatten 方法
# ndarray.flat 1-D iterator over an array.
# ndarray.flatten 1-D array copy of the elements of an array in row-major order.
for i in xrange(3):
    plt.subplot(3,3,i*3+1),plt.imshow(images[i*3],'gray')
    plt.title(titles[i*3]), plt.xticks([]), plt.yticks([])
    plt.subplot(3,3,i*3+2),plt.hist(images[i*3].ravel(),256)
    plt.title(titles[i*3+1]), plt.xticks([]), plt.yticks([])
    plt.subplot(3,3,i*3+3),plt.imshow(images[i*3+2],'gray')
    plt.title(titles[i*3+2]), plt.xticks([]), plt.yticks([])
plt.show()

```

结果:



## 15.4 Otsu's 二值化是如何工作的？

在这一部分我们会演示怎样使用 Python 来实现 Otsu 二值化算法，从而告诉大家它是如何工作的。如果你不感兴趣的话可以跳过这一节。

因为是双峰图，Otsu 算法就是要找到一个阈值（ $t$ ），使得同一类加权方差最小，需要满足下列关系式：

$$\sigma_w^2(t) = q_1(t)\sigma_1^2(t) + q_2(t)\sigma_2^2(t)$$

其中：

$$\begin{aligned} q_1(t) &= \sum_{i=1}^t P(i) \quad \& \quad q_2(t) = \sum_{i=t+1}^I P(i) \\ \mu_1(t) &= \sum_{i=1}^t \frac{iP(i)}{q_1(t)} \quad \& \quad \mu_2(t) = \sum_{i=t+1}^I \frac{iP(i)}{q_2(t)} \\ \sigma_1^2(t) &= \sum_{i=1}^t [i - \mu_1(t)]^2 \frac{P(i)}{q_1(t)} \quad \& \quad \sigma_2^2(t) = \sum_{i=t+1}^I [i - \mu_2(t)]^2 \frac{P(i)}{q_2(t)} \end{aligned}$$

其实就是在两个峰之间找到一个阈值  $t$ ，将这两个峰分开，并且使每一个峰内的方差最小。实现这个算法的 Python 代码如下：

```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 11 14:46:12 2014

@author: duan
"""

import cv2
import numpy as np

img = cv2.imread('noisy2.png',0)
blur = cv2.GaussianBlur(img,(5,5),0)

# find normalized_histogram, and its cumulative distribution function
# 计算归一化直方图
#CalcHist(image, accumulate=0, mask=NULL)
hist = cv2.calcHist([blur],[0],None,[256],[0,256])
hist_norm = hist.ravel()/hist.max()
Q = hist_norm.cumsum()

bins = np.arange(256)

fn_min = np.inf
thresh = -1

for i in xrange(1,256):
    p1,p2 = np.hsplit(hist_norm,[i]) # probabilities
    q1,q2 = Q[i],Q[255]-Q[i] # cum sum of classes
    b1,b2 = np.hsplit(bins,[i]) # weights

    # finding means and variances
    m1,m2 = np.sum(p1*b1)/q1, np.sum(p2*b2)/q2
    v1,v2 = np.sum(((b1-m1)**2)*p1)/q1,np.sum(((b2-m2)**2)*p2)/q2

    # calculates the minimization function
    fn = v1*q1 + v2*q2
    if fn < fn_min:
        fn_min = fn
        thresh = i

# find otsu's threshold value with OpenCV function
ret, otsu = cv2.threshold(blur,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
print thresh,ret

```

(这里有些新的函数，我们会在后面的章节中讲到他们)

## 更多资源

1. Digital Image Processing, Rafael C. Gonzalez

## 练习

## 16 图像平滑

### 目标

- 学习使用不同的低通滤波器对图像进行模糊
- 使用自定义的滤波器对图像进行卷积（2D 卷积）

### 2D 卷积

与以为信号一样，我们也可以对 2D 图像实施低通滤波（LPF），高通滤波（HPF）等。LPF 帮助我们去除噪音，模糊图像。HPF 帮助我们找到图像的边缘

OpenCV 提供的函数 **cv.filter2D()** 可以让我们对一幅图像进行卷积操作。下面我们将对一幅图像使用平均滤波器。下面是一个 5x5 的平均滤波器核：

$$K = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

操作如下：将核放在图像的一个像素 A 上，求与核对应的图像上 25（5x5）个像素的和，在取平均数，用这个平均数替代像素 A 的值。重复以上操作直到将图像的每一个像素值都更新一边。代码如下，运行一下吧。

```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 11 19:43:04 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('opencv_logo.png')

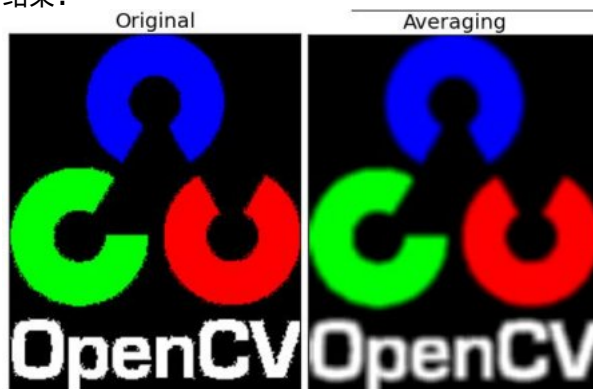
kernel = np.ones((5,5),np.float32)/25

#cv.Filter2D(src, dst, kernel, anchor=(-1, -1))
#ddepth -desired depth of the destination image;
#if it is negative, it will be the same as src.depth();
#the following combinations of src.depth() and ddepth are supported:
#src.depth() = CV_8U, ddepth = -1/CV_16S/CV_32F/CV_64F
#src.depth() = CV_16U/CV_16S, ddepth = -1/CV_32F/CV_64F
#src.depth() = CV_32F, ddepth = -1/CV_32F/CV_64F
#src.depth() = CV_64F, ddepth = -1/CV_64F
#when ddepth=-1, the output image will have the same depth as the source.
dst = cv2.filter2D(img,-1,kernel)

plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([], plt.yticks([]))
plt.subplot(122),plt.imshow(dst),plt.title('Averaging')
plt.xticks([], plt.yticks([]))
plt.show()

```

结果:



## 图像模糊（图像平滑）

使用低通滤波器可以达到图像模糊的目的。这对与去除噪音很有帮助。其实就是去除图像中的高频成分（比如：噪音，边界）。所以边界也会被模糊一点。（当然，也有一些模糊技术不会模糊掉边界）。OpenCV 提供了四种模糊技术。

### 16.1 平均

这是由一个归一化卷积框完成的。他只是用卷积框覆盖区域所有像素的平均值来代替中心元素。可以使用函数 **cv2.blur()** 和 **cv2.boxFilter()** 来完成这个任务。可以同看查看文档了解更多卷积框的细节。我们需要设定卷积框的宽和高。下面是一个 3x3 的归一化卷积框：

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

**注意：**如果你不想使用归一化卷积框，你应该使用 **cv2.boxFilter()**，这时要传入参数 **normalize=False**。

下面与第一部分一样的一个例子：

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jan 11 19:54:08 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

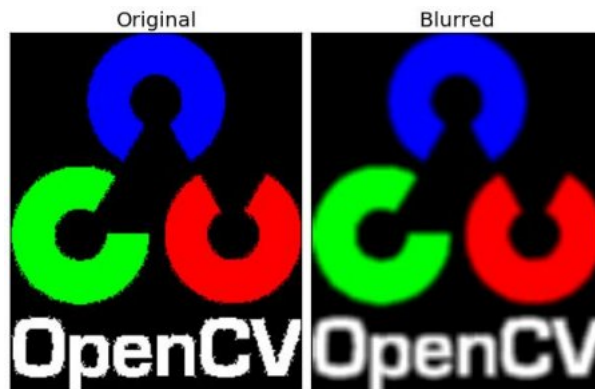
img = cv2.imread('opencv_logo.png')

blur = cv2.blur(img,(5,5))

plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(blur),plt.title('Blurred')
plt.xticks([]), plt.yticks([])
plt.show()
```

结果：





## 16.2 高斯模糊

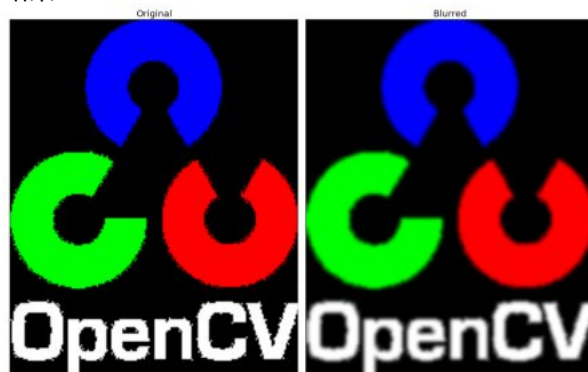
现在把卷积核换成高斯核（简单来说，方框不变，将原来每个方框的值是相等的，现在里面的值是符合高斯分布的，方框中心的值最大，其余方框根据距离中心元素的距离递减，构成一个高斯小山包。原来的求平均数现在变成求加权平均数，全就是方框里的值）。实现的函数是 **cv2.GaussianBlur()**。我们需要指定高斯核的宽和高（必须是奇数）。以及高斯函数沿 X，Y 方向的标准差。如果我们只指定了 X 方向的的标准差，Y 方向也会取相同值。如果两个标准差都是 0，那么函数会根据核函数的大小自己计算。高斯滤波可以有效的从图像中去除高斯噪音。

如果你愿意的话，你也可以使用函数 **cv2.getGaussianKernel()** 自己构建一个高斯核。

如果要使用高斯模糊的话，上边的代码应该写成：

```
#0 是指根据窗口大小 (5,5) 来计算高斯函数标准差
blur = cv2.GaussianBlur(img,(5,5),0)
```

结果：



## 16.3 中值模糊

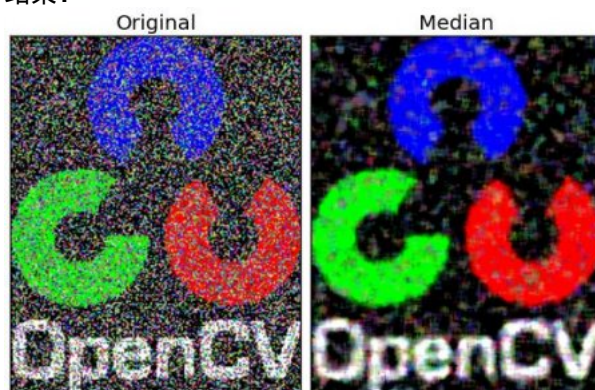
顾名思义就是用与卷积框对应像素的中值来替代中心像素的值。这个滤波器经常用来去除椒盐噪声。前面的滤波器都是用计算得到的一个新值来取代中心像素的值，而中值滤波是用中心像素周围（也可以使他本身）的值来取代他。他能有效的去除噪声。卷积核的大小也应该是一个奇数。

在这个例子中，我们给原始图像加上 50% 的噪声然后再使用中值模糊。

代码：

```
median = cv2.medianBlur(img,5)
```

结果：



## 16.4 双边滤波

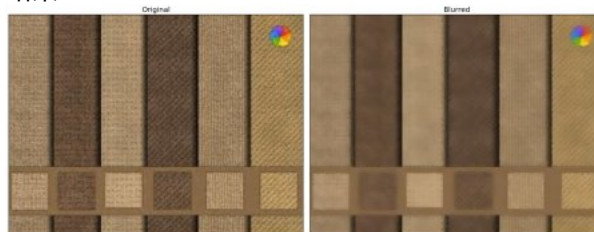
函数 **cv2.bilateralFilter()** 能在保持边界清晰的情况下有效的去除噪音。但是这种操作与其他滤波器相比会比较慢。我们已经知道高斯滤波器是求中心点邻近区域像素的高斯加权平均值。这种高斯滤波器只考虑像素之间的空间关系，而不会考虑像素值之间的关系（像素的相似度）。所以这种方法不会考虑一个像素是否位于边界。因此边界也会别模糊掉，而这正不是我们想要。

双边滤波在同时使用空间高斯权重和灰度值相似性高斯权重。空间高斯函数确保只有邻近区域的像素对中心点有影响，灰度值相似性高斯函数确保只有与中心像素灰度值相近的才会被用来做模糊运算。所以这种方法会确保边界不会被模糊掉，因为边界处的灰度值变化比较大。

进行双边滤波的代码如下：

```
#cv2.bilateralFilter(src, d, sigmaColor, sigmaSpace)
#d - Diameter of each pixel neighborhood that is used during filtering.
# If it is non-positive, it is computed from sigmaSpace
#9 邻域直径, 两个 75 分别是空间高斯函数标准差, 灰度值相似性高斯函数标准差
blur = cv2.bilateralFilter(img,9,75,75)
```

结果:



看见了把, 上图中的纹理被模糊掉了, 但是边界还在。

## 更多资源

1. Details about the [bilateral filtering](#)

## 练习

## 17 形态学转换

### 目标

- 学习不同的形态学操作，例如腐蚀，膨胀，开运算，闭运算等
- 我们要学习的函数有：`cv2.erode()`，`cv2.dilate()`，`cv2.morphologyEx()`等

### 原理

形态学操作是根据图像形状进行的简单操作。一般情况下对二值化图像进行的操作。需要输入两个参数，一个是原始图像，第二个被称为结构化元素或核，它是用来决定操作的性质的。两个基本的形态学操作是腐蚀和膨胀。他们的变体构成了开运算，闭运算，梯度等。我们会以下图为例逐一介绍它们。



### 17.1 腐蚀

就像土壤侵蚀一样，这个操作会把前景物体的边界腐蚀掉（但是前景仍然是白色）。这是怎么做到的呢？卷积核沿着图像滑动，如果与卷积核对应的原图像的所有像素值都是 1，那么中心元素就保持原来的像素值，否则就变为零。

这回产生什么影响呢？根据卷积核的大小靠近前景的所有像素都会被腐蚀掉（变为 0），所以前景物体会变小，整幅图像的白色区域会减少。这对于去除白噪声很有用，也可以用来断开两个连在一块的物体等。

这里我们有一个例子，使用一个 5x5 的卷积核，其中所有的值都是以。让我们看看他是如何工作的：

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jan 11 21:46:11 2014

@author: duan
"""

import cv2
import numpy as np

img = cv2.imread('j.png',0)
kernel = np.ones((5,5),np.uint8)
erosion = cv2.erode(img,kernel,iterations = 1)
```

结果:



## 17.2 膨胀

与腐蚀相反，与卷积核对应的原图像的像素值中只要有一个是 1，中心元素的像素值就是 1。所以这个操作会增加图像中的白色区域（前景）。一般在去噪声时先用腐蚀再用膨胀。因为腐蚀在去掉白噪声的同时，也会使前景对象变小。所以我们再对他进行膨胀。这时噪声已经被去除了，不会再回来了，但是前景还在并会增加。膨胀也可以用来连接两个分开的物体。

```
dilation = cv2.dilate(img,kernel,iterations = 1)
```

结果:



### 17.3 开运算

先进性腐蚀再进行膨胀就叫做开运算。就像我们上面介绍的那样，它被用来去除噪声。这里我们用到的函数是 `cv2.morphologyEx()`。

```
opening = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
```

结果：



### 17.4 闭运算

先膨胀再腐蚀。它经常被用来填充前景物体中的小洞，或者前景物体上的小黑点。

```
closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
```

结果：



### 17.5 形态学梯度

其实就是一幅图像膨胀与腐蚀的差别。  
结果看上去就像前景物体的轮廓。

```
gradient = cv2.morphologyEx(img, cv2.MORPH_GRADIENT, kernel)
```

结果：



## 17.6 礼帽

原始图像与进行开运算之后得到的图像的差。下面的例子是用一个 9x9 的核进行礼帽操作的结果。

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

结果：



## 17.7 黑帽

进行闭运算之后得到的图像与原始图像的差。

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

结果：



## 17.8 形态学操作之间的关系

我们把以上集中形态学操作之间的关系列出来以供大家参考：

Opening:

$$dst = open(src, element) = dilate(erode(src, element), element)$$

Closing:

$$dst = close(src, element) = erode(dilate(src, element), element)$$

Morphological gradient:

$$dst = morph\_grad(src, element) = dilate(src, element) - erode(src, element)$$

“Top hat”:

$$dst = tophat(src, element) = src - open(src, element)$$

“Black hat”:

$$dst = blackhat(src, element) = close(src, element) - src$$

## 结构化元素

在前面的例子中我们使用 Numpy 构建了结构化元素，它是正方形的。但有时我们需要构建一个椭圆形/圆形的核。为了实现这种要求，提供了 OpenCV 函数 **cv2.getStructuringElement()**。你只需要告诉他你需要的核的形状和大小。

```
# Rectangular Kernel
>>> cv2.getStructuringElement(cv2.MORPH_RECT, (5,5))
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]], dtype=uint8)

# Elliptical Kernel
>>> cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
array([[0, 0, 1, 0, 0],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [0, 0, 1, 0, 0]], dtype=uint8)

# Cross-shaped Kernel
>>> cv2.getStructuringElement(cv2.MORPH_CROSS, (5,5))
array([[0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0],
       [1, 1, 1, 1, 1],
       [0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0]], dtype=uint8)
```



## 更多资源

1. [Morphological Operations](#) at HIPR2

## 练习

## 18 图像梯度

### 目标

- 图像梯度，图像边界等
- 使用到的函数有：`cv2.Sobel()`，`cv2.Scharr()`，`cv2.Laplacian()` 等

### 原理

梯度简单来说就是求导。

OpenCV 提供了三种不同的梯度滤波器，或者说高通滤波器：Sobel, Scharr 和 Laplacian。我们会意义介绍他们。

Sobel, Scharr 其实就是求一阶或二阶导数。Scharr 是对 Sobel (使用小的卷积核求解求解梯度角度时) 的优化。Laplacian 是求二阶导数。

### 18.1 Sobel 算子和 Scharr 算子

Sobel 算子是高斯平滑与微分操作的结合体，所以它的抗噪声能力很好。你可以设定求导的方向 (xorder 或 yorder)。还可以设定使用的卷积核的大小 (ksize)。如果 ksize=-1, 会使用 3x3 的 Scharr 滤波器，它的的效果要比 3x3 的 Sobel 滤波器好 (而且速度相同，所以在使用 3x3 滤波器时应该尽量使用 Scharr 滤波器)。3x3 的 Scharr 滤波器卷积核如下：

	-3	0	3
x 方向	-10	0	10
	-3	0	3

, y 方向

-3	-10	-3
0	0	0
3	10	3

### 18.2 Laplacian 算子

拉普拉斯算子可以使用二阶导数的形式定义，可假设其离散实现类似于二阶 Sobel 导数，事实上，OpenCV 在计算拉普拉斯算子时直接调用 Sobel 算子。计算公式如下：

$$\Delta src = \frac{\partial^2 src}{\partial x^2} + \frac{\partial^2 src}{\partial y^2}$$

拉普拉斯滤波器使用的卷积核：

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

## 代码

下面的代码分别使用以上三种滤波器对同一幅图进行操作。使用的卷积核都是 5x5 的。

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jan 12 11:01:40 2014

@author: duan
"""
import cv2
import numpy as np
from matplotlib import pyplot as plt

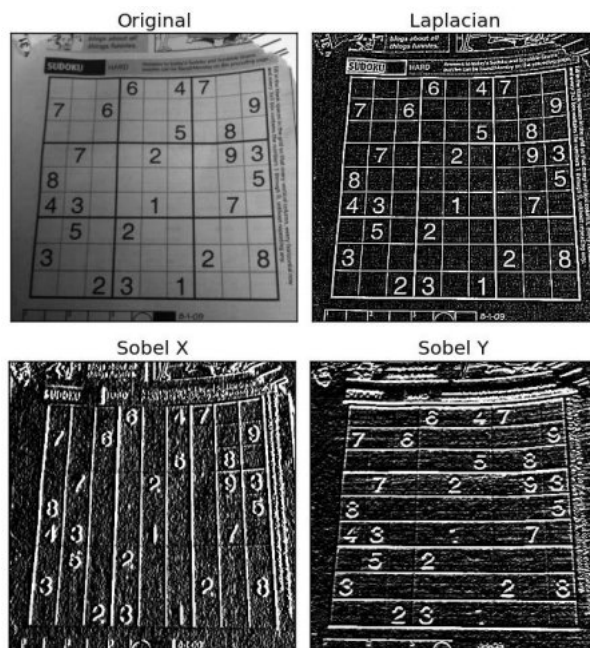
img=cv2.imread('dave.jpg',0)

#cv2.CV_64F 输出图像的深度 (数据类型), 可以使用-1, 与原图像保持一致 np.uint8
laplacian=cv2.Laplacian(img,cv2.CV_64F)
# 参数 1,0 为只在 x 方向求一阶导数, 最大可以求 2 阶导数。
sobelx=cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)
# 参数 0,1 为只在 y 方向求一阶导数, 最大可以求 2 阶导数。
sobely=cv2.Sobel(img,cv2.CV_64F,0,1,ksize=5)

plt.subplot(2,2,1),plt.imshow(img,cmap = 'gray')
plt.title('Original'), plt.xticks([], plt.yticks([]))
plt.subplot(2,2,2),plt.imshow(laplacian,cmap = 'gray')
plt.title('Laplacian'), plt.xticks([], plt.yticks([]))
plt.subplot(2,2,3),plt.imshow(sobelx,cmap = 'gray')
plt.title('Sobel X'), plt.xticks([], plt.yticks([]))
plt.subplot(2,2,4),plt.imshow(sobely,cmap = 'gray')
plt.title('Sobel Y'), plt.xticks([], plt.yticks([]))

plt.show()
```

结果:



## 一个重要的事！

在查看上面这个例子的注释时不知道你有没有注意到：当我们可以通过参数 `-1` 来设定输出图像的深度（数据类型）与原图像保持一致，但是我们在代码中使用的却是 `cv2.CV_64F`。这是为什么呢？想象一下一个从黑到白的边界的导数是整数，而一个从白到黑的边界点导数却是负数。如果原图像的深度是 `np.int8` 时，所有的负值都会被截断变成 `0`，换句话说就是把边界丢失掉。

所以如果这两种边界你都检测到，最好的办法就是将输出的数据类型设置的更高，比如 `cv2.CV_16S`, `cv2.CV_64F` 等。取绝对值然后再把它转回到 `cv2.CV_8U`。下面的示例演示了输出图片的深度不同造成的不同效果。

```

# -*- coding: utf-8 -*-
"""
Created on Sun Jan 12 11:11:02 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('boxs.png',0)

# Output dtype = cv2.CV_8U
sobelx8u = cv2.Sobel(img,cv2.CV_8U,1,0,ksize=5)
# 也可以将参数设为-1
#sobelx8u = cv2.Sobel(img,-1,1,0,ksize=5)

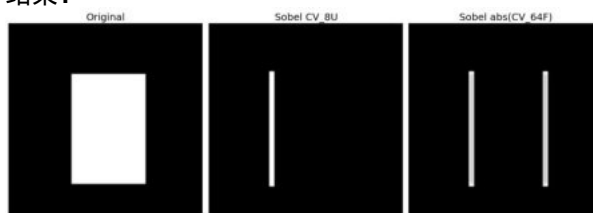
# Output dtype = cv2.CV_64F. Then take its absolute and convert to cv2.CV_8U
sobelx64f = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)
abs_sobel64f = np.absolute(sobelx64f)
sobel_8u = np.uint8(abs_sobel64f)

plt.subplot(1,3,1),plt.imshow(img,cmap = 'gray')
plt.title('Original'), plt.xticks([], plt.yticks([]))
plt.subplot(1,3,2),plt.imshow(sobelx8u,cmap = 'gray')
plt.title('Sobel CV_8U'), plt.xticks([], plt.yticks([]))
plt.subplot(1,3,3),plt.imshow(sobel_8u,cmap = 'gray')
plt.title('Sobel abs(CV_64F)'), plt.xticks([], plt.yticks([]))

plt.show()

```

结果：



更多资源

练习

## 19 Canny 边缘检测

### 目标

OpenCV 中的 Canny 边缘检测

- 了解 Canny 边缘检测的概念
- 学习函数 `cv2.Canny()`

### 19.1 原理

Canny 边缘检测是一种非常流行的边缘检测算法，是 John F.Canny 在 1986 年提出的。它是一个有很多步构成的算法，我们接下来会逐步介绍。

#### 19.1.1 噪声去除

由于边缘检测很容易受到噪声影响，所以第一步是使用 5x5 的高斯滤波器去除噪声，这个前面我们已经学过了。

#### 19.1.2 计算图像梯度

对平滑后的图像使用 Sobel 算子计算水平方向和竖直方向的一阶导数（图像梯度）（ $G_x$  和  $G_y$ ）。根据得到的这两幅梯度图（ $G_x$  和  $G_y$ ）找到边界的梯度和方向，公式如下：

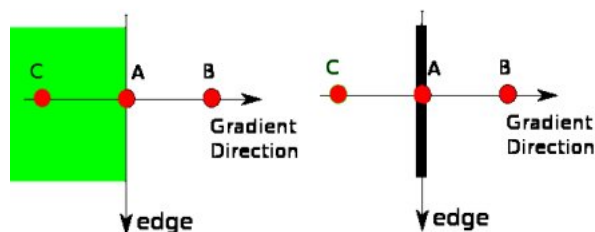
$$Edge\_Gradient(G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle(\theta) = \tan^{-1} \left( \frac{G_x}{G_y} \right)$$

梯度的方向一般总是与边界垂直。梯度方向被归为四类：垂直，水平，和两个对角线。

#### 19.1.3 非极大值抑制

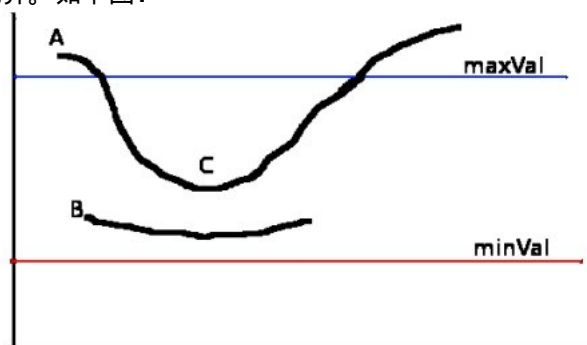
在获得梯度的方向和大小之后，应该对整幅图像做一个扫描，去除那些非边界上的点。对每一个像素进行检查，看这个点的梯度是不是周围具有相同梯度方向的点中最大的。如下图所示：



现在你得到的是一个包含“窄边界”的二值图像。

#### 19.1.4 滞后阈值

现在要确定那些边界才是真正的边界。这时我们需要设置两个阈值： $\text{minVal}$  和  $\text{maxVal}$ 。当图像的灰度梯度高于  $\text{maxVal}$  时被认为是真的边界，那些低于  $\text{minVal}$  的边界会被抛弃。如果介于两者之间的话，就要看这个点是否与某个被确定为真正的边界点相连，如果是就认为它也是边界点，如果不是就抛弃。如下图：



A 高于阈值  $\text{maxVal}$  所以是真正的边界点，C 虽然低于  $\text{maxVal}$  但高于  $\text{minVal}$  并且与 A 相连，所以也被认为是真正的边界点。而 B 就会被抛弃，因为他不仅低于  $\text{maxVal}$  而且不与真正的边界点相连。所以选择合适的  $\text{maxVal}$  和  $\text{minVal}$  对于能否得到好的结果非常重要。

在这一步一些小的噪声点也会被除去，因为我们假设边界都是一些长的线段。

## 19.2 OpenCV 中的 Canny 边界检测

在 OpenCV 中只需要一个函数：`cv2.Canny()`，就可以完成以上几步。让我们看如何使用这个函数。这个函数的第一个参数是输入图像。第二和第三个分别是 **minVal** 和 **maxVal**。第三个参数设置用来计算图像梯度的 Sobel 卷积核的大小，默认值为 3。最后一个参数是 **L2gradient**，它可以用来设定求梯度大小的方程。如果设为 **True**，就会使用我们上面提到过的方程，否则使用方程： $\text{Edge\_Gradient}(G) = |G_x^2| + |G_y^2|$  代替，默认值为 **False**。

```

# -*- coding: utf-8 -*-
"""
Created on Sun Jan 12 14:37:56 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

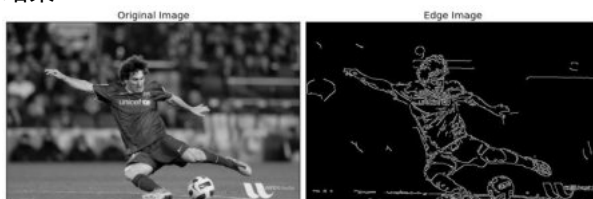
img = cv2.imread('messi5.jpg',0)
edges = cv2.Canny(img,100,200)

plt.subplot(121),plt.imshow(img,cmap = 'gray')
plt.title('Original Image'), plt.xticks([], plt.yticks([]))
plt.subplot(122),plt.imshow(edges,cmap = 'gray')
plt.title('Edge Image'), plt.xticks([], plt.yticks([]))

plt.show()

```

结果：



## 更多资源

1. Canny edge detector at [Wikipedia](#)
2. [Canny Edge Detection Tutorial](#) by Bill Green, 2002.

## 练习

1. 写一个小程序，可以通过调节滑动条来设置阈值 minVal 和 maxVal 进而来进行 Canny 边界检测。这样你就会理解阈值的重要性了。



## 20 图像金字塔

### 目标

- 学习图像金字塔
- 使用图像创建一个新水果：“橘子苹果”
- 将要学习的函数有：cv2.pyrUp(), cv2.pyrDown()。

### 20.1 原理

一般情况下，我们要处理是一副具有固定分辨率的图像。但是有些情况下，我们需要对同一图像的不同分辨率的子图像进行处理。比如，我们要在一幅图像中查找某个目标，比如脸，我们不知道目标在图像中的尺寸大小。这种情况下，我们需要创建一组图像，这些图像是具有不同分辨率的原始图像。我们把这组图像叫做图像金字塔（简单来说就是同一图像的不同分辨率的子图集合）。如果我们把最大的图像放在底部，最小的放在顶部，看起来像一座金字塔，故而得名图像金字塔。

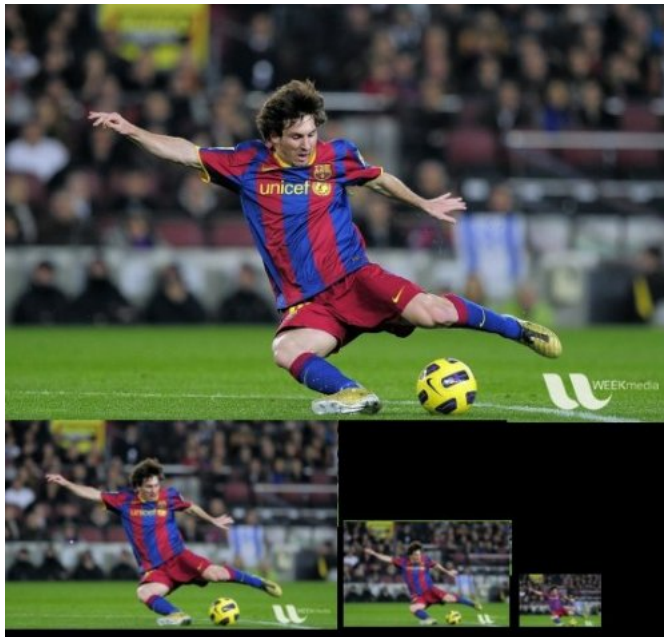
有两类图像金字塔：高斯金字塔和拉普拉斯金字塔。

高斯金字塔的顶部是通过将底部图像中的连续的行和列去除得到的。顶部图像中的每个像素值等于下一层图像中 5 个像素的高斯加权平均值。这样操作一次一个  $M \times N$  的图像就变成了一个  $M/2 \times N/2$  的图像。所以这幅图像的面积就变为原来图像面积的四分之一。这被称为 Octave。连续进行这样的操作我们就会得到一个分辨率不断下降的图像金字塔。我们可以使用函数 cv2.pyrDown() 和 cv2.pyrUp() 构建图像金字塔。

函数 cv2.pyrDown() 从一个高分辨率大尺寸的图像向上构建一个金字塔（尺寸变小，分辨率降低）。

```
img = cv2.imread('messi5.jpg')
lower_reso = cv2.pyrDown(higher_reso)
```

下图是一个四层的图像金字塔。



函数 `cv2.pyrUp()` 从一个低分辨率小尺寸的图像向下构建一个金子塔（尺寸变大，但分辨率不会增加）。

```
higher_reso2 = cv2.pyrUp(lower_reso)
```

你要记住的是 `higher_reso2` 和 `higher_reso` 是不同的。因为一旦使用 `cv2.pyrDown()`，图像的分辨率就会降低，信息就会被丢失。下图就是从 `cv2.pyrDown()` 产生的图像金字塔的（由下到上）第三层图像使用函数 `cv2.pyrUp()` 得到的图像，与原图像相比分辨率差了很多。



拉普拉斯金字塔可以有高斯金字塔计算得来，公式如下：

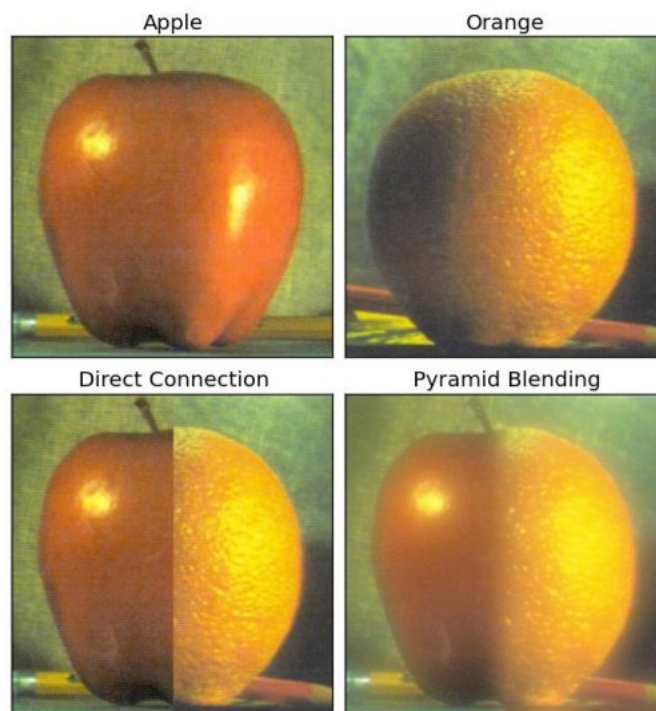
$$L_i = G_i - PyrUp(G_{i+1})$$

拉普拉金字塔的图像看起来就像边界图，其中很多像素都是 0。他们经常被用在图像压缩中。下图就是一个三层的拉普拉斯金字塔：



## 20.2 使用金字塔进行图像融合

图像金字塔的一个应用是图像融合。例如，在图像缝合中，你需要将两幅图叠在一起，但是由于连接区域图像像素的不连续性，整幅图的效果看起来会很差。这时图像金字塔就可以排上用场了，他可以帮你实现无缝连接。这里的一个经典案例就是将两个水果融合成一个，看看下图也许你就明白我在讲什么了。



你可以通过阅读后边的更多资源来了解更多关于图像融合，拉普拉斯金字塔的细节。

实现上述效果的步骤如下：

1. 读入两幅图像，苹果和橘子
2. 构建苹果和橘子的高斯金字塔（6 层）
3. 根据高斯金字塔计算拉普拉斯金字塔
4. 在拉普拉斯的每一层进行图像融合（苹果的左边与橘子的右边融合）
5. 根据融合后的图像金字塔重建原始图像。

下图是摘自《学习 OpenCV》展示了金子塔的构建，以及如何从金字塔重建原始图像的过程。

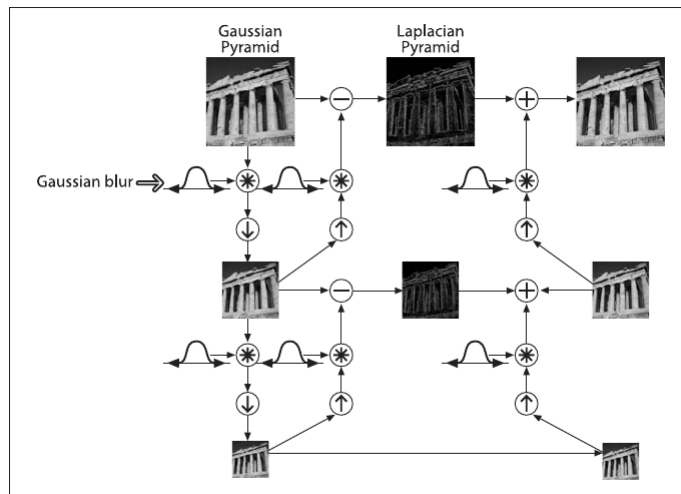


Figure 5-21. The Gaussian pyramid and its inverse, the Laplacian pyramid

整个过程的代码如下。(为了简单，每一步都是独立完成的，这回消耗更多的内存，如果你愿意的话可以对他进行优化)

```

import cv2
import numpy as np, sys

A = cv2.imread('apple.jpg')
B = cv2.imread('orange.jpg')

# generate Gaussian pyramid for A
G = A.copy()
gpA = [G]
for i in xrange(6):
    G = cv2.pyrDown(G)
    gpA.append(G)

# generate Gaussian pyramid for B
G = B.copy()
gpB = [G]
for i in xrange(6):
    G = cv2.pyrDown(G)
    gpB.append(G)

# generate Laplacian Pyramid for A
lpA = [gpA[5]]
for i in xrange(5,0,-1):
    GE = cv2.pyrUp(gpA[i])
    L = cv2.subtract(gpA[i-1],GE)
    lpA.append(L)

# generate Laplacian Pyramid for B
lpB = [gpB[5]]
for i in xrange(5,0,-1):
    GE = cv2.pyrUp(gpB[i])
    L = cv2.subtract(gpB[i-1],GE)
    lpB.append(L)

# Now add left and right halves of images in each level
#numpy.hstack(tup)
#Take a sequence of arrays and stack them horizontally
#to make a single array.
LS = []
for la,lb in zip(lpA,lpB):
    rows,cols,dpt = la.shape
    ls = np.hstack((la[:,0:cols/2], lb[:,cols/2:]))
    LS.append(ls)

# now reconstruct
ls_ = LS[0]
for i in xrange(1,6):
    ls_ = cv2.pyrUp(ls_)
    ls_ = cv2.add(ls_, LS[i])

# image with direct connecting each half
real = np.hstack((A[:,0:cols/2],B[:,cols/2:]))

cv2.imwrite('Pyramid_blending2.jpg',ls_)
cv2.imwrite('Direct_blending.jpg',real)

```

## 更多资源

1. [Image Blending](#)

## 练习

## 21 OpenCV 中的轮廓

### 21.1 初识轮廓

#### 目标

- 理解什么是轮廓
- 学习找轮廓，绘制轮廓等
- 函数：`cv2.findContours()`，`cv2.drawContours()`

#### 21.1.1 什么是轮廓

轮廓可以简单认为成将连续的点（连着边界）连在一起的曲线，具有相同的颜色或者灰度。轮廓在形状分析和物体的检测和识别中很有用。

- 为了更加准确，要使用二值化图像。在寻找轮廓之前，要进行阈值化处理或者 Canny 边界检测。
- 查找轮廓的函数会修改原始图像。如果你在找到轮廓之后还想使用原始图像的话，你应该将原始图像存储到其他变量中。
- 在 OpenCV 中，查找轮廓就像在黑色背景中超白色物体。你应该记住，要找的物体应该是白色而背景应该是黑色。

让我们看看如何在一个二值图像中查找轮廓：

函数 `cv2.findContours()` 有三个参数，第一个是输入图像，第二个是轮廓检索模式，第三个是轮廓近似方法。返回值有三个，第一个是图像，第二个是轮廓，第三个是（轮廓的）层析结构。轮廓（第二个返回值）是一个 Python 列表，其中存储这图像中的所有轮廓。每一个轮廓都是一个 Numpy 数组，包含对象边界点（`x`，`y`）的坐标。

**注意：**我们后边会对第二和第三个参数，以及层次结构进行详细介绍。在那之前，例子中使用的参数值对所有图像都是适用的。

#### 21.1.2 怎样绘制轮廓

函数 `cv2.drawContours()` 可以被用来绘制轮廓。它可以根据你提供的边界点绘制任何形状。它的第一个参数是原始图像，第二个参数是轮廓，一个 Python 列表。第三个参数是轮廓的索引（在绘制独立轮廓是很有用，当设置为 -1 时绘制所有轮廓）。接下来的参数是轮廓的颜色和厚度等。

在一幅图像上绘制所有的轮廓：



```
# -*- coding: utf-8 -*-
"""
Created on Sun Jan 12 18:05:52 2014

@author: duan
"""

import numpy as np
import cv2

im = cv2.imread('test.jpg')
imgray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
ret,thresh = cv2.threshold(imgray,127,255,0)
image, contours, hierarchy = cv2.findContours(thresh,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
```

绘制独立轮廓，如第四个轮廓：

```
img = cv2.drawContour(img, contours, -1, (0,255,0), 3)
```

但是大多数时候，下面的方法更有用：

```
img = cv2.drawContours(img, contours, 3, (0,255,0), 3)
```

**注意：**最后这两种方法结果是一样的，但是后边的知识会告诉你最后一种方法更有用。

### 21.1.3 轮廓的近似方法

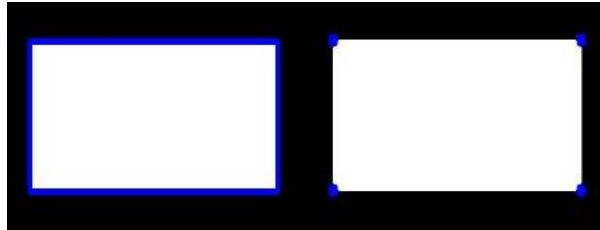
这是函数 **cv2.findContours()** 的第三个参数。它到底代表什么意思呢？

上边我们已经提到轮廓是一个形状具有相同灰度值的边界。它会存储形状边界上所有的 (x, y) 坐标。但是需要将所有的这些边界点都存储吗？这就是这个参数要告诉函数 **cv2.findContours** 的。

这个参数如果被设置为 **cv2.CHAIN\_APPROX\_NONE**，所有的边界点都会被存储。但是我们真的需要这么多点吗？例如，当我们找的边界是一条直线时。你用需要直线上所有的点来表示直线吗？不是的，我们只需要这条直线的两个端点而已。这就是 **cv2.CHAIN\_APPROX\_SIMPLE** 要做的。它会

将轮廓上的冗余点都去掉，压缩轮廓，从而节省内存开支。

我们用下图中的矩形来演示这个技术。在轮廓列表中的每一个坐标上画一个蓝色圆圈。第一个图显示使用 `cv2.CHAIN_APPROX_NONE` 的效果，一共 734 个点。第二个图是使用 `cv2.CHAIN_APPROX_SIMPLE` 的结果，只有 4 个点。看到他的威力了吧！



**更多资源**

**练习**

## 21.2 轮廓特征

### 目标

- 查找轮廓的不同特征，例如面积，周长，重心，边界框等。
- 你会学到很多轮廓相关函数

### 21.2.1 矩

图像的矩可以帮助我们计算图像的质心，面积等。详细信息请查看维基百科[Image Moments](#)。

函数 `cv2.moments()` 会将计算得到的矩以一个字典的形式返回。如下：

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jan 12 18:30:17 2014

@author: duan
"""

import cv2
import numpy as np

img = cv2.imread('star.jpg',0)
ret,thresh = cv2.threshold(img,127,255,0)
contours,hierarchy = cv2.findContours(thresh, 1, 2)

cnt = contours[0]
M = cv2.moments(cnt)
print M
```

根据这些矩的值，我们可以计算出对象的重心： $C_x = \frac{M_{10}}{M_{00}}$ ， $C_y = \frac{M_{01}}{M_{00}}$ 。

```
cx = int(M['m10']/M['m00'])
cy = int(M['m01']/M['m00'])
```

### 21.2.2 轮廓面积

轮廓的面积可以使用函数 `cv2.contourArea()` 计算得到，也可以使用矩 (0 阶矩)，`M['m00']`。

```
area = cv2.contourArea(cnt)
```

### 21.2.3 轮廓周长

也被称为弧长。可以使用函数 **cv2.arcLength()** 计算得到。这个函数的第二参数可以用来指定对象的形状是闭合的 (**True**)，还是打开的 (一条曲线)。

```
perimeter = cv2.arcLength(cnt,True)
```

### 21.2.4 轮廓近似

将轮廓形状近似到另外一种由更少点组成的轮廓形状，新轮廓的点的数目由我们设定的准确度来决定。使用的 **Douglas-Peucker** 算法，你可以到维基百科获得更多此算法的细节。

为了帮助理解，假设我们要在一幅图像中查找一个矩形，但是由于图像的种种原因，我们不能得到一个完美的矩形，而是一个“坏形状”(如下图所示)。现在你就可以使用这个函数来近似这个形状 () 了。这个函数的第二个参数叫 **epsilon**，它是从原始轮廓到近似轮廓的最大距离。它是一个准确度参数。选择一个好的 **epsilon** 对于得到满意结果非常重要。

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

下边，第二幅图中的绿线是当  $\epsilon = 10\%$  时得到的近似轮廓，第三幅图是当  $\epsilon = 1\%$  时得到的近似轮廓。第三个参数设定弧线是否闭合。



### 21.2.5 凸包

凸包与轮廓近似相似，但不同，虽然有些情况下它们给出的结果是一样的。函数 `cv2.convexHull()` 可以用来检测一个曲线是否具有凸性缺陷，并能纠正缺陷。一般来说，凸性曲线总是凸出来的，至少是平的。如果有地方凹进去了就被叫做凸性缺陷。例如下图中的手。红色曲线显示了手的凸包，凸性缺陷被双箭头标出来了。



关于他的语法还有一些需要交代：

```
hull = cv2.convexHull(points[, hull[, clockwise[, returnPoints]])
```

参数：

- **points** 我们要传入的轮廓
- **hull** 输出，通常不需要
- **clockwise** 方向标志。如果设置为 **True**，输出的凸包是顺时针方向的。否则为逆时针方向。
- **returnPoints** 默认值为 **True**。它会返回凸包上点的坐标。如果设置为 **False**，就会返回与凸包点对应的轮廓上的点。

要获得上图的凸包，下面的命令就够了：

```
hull = cv2.convexHull(cnt)
```

但是如果你想获得凸性缺陷，需要把 **returnPoints** 设置为 **False**。以上面的矩形为例，首先我们找到他的轮廓 `cnt`。现在我把 **returnPoints** 设置为 **True** 查找凸包，我得到下列值：

[[[234 202]], [[ 51 202]], [[ 51 79]], [[234 79]]], 其实就是矩形的四个角点。

现在把 **returnPoints** 设置为 **False**, 我得到的结果是

[[129],[ 67],[ 0],[142]]

他们是轮廓点的索引。例如:  $cnt[129] = [[234, 202]]$ , 这与前面我们得到结果的第一个值是一样的。

在凸检验中你我们还会遇到这些。

### 21.2.6 凸性检测

函数 **cv2.isContourConvex()** 可以用来检测一个曲线是不是凸的。它只能返回 **True** 或 **False**。没什么大不了的。

```
k = cv2.isContourConvex(cnt)
```

### 21.2.7 边界矩形

有两类边界矩形。

**直边界矩形** 一个直矩形 ( 就是没有旋转的矩形 )。它不会考虑对象是否旋转。所以边界矩形的面积不是最小的。可以使用函数 **cv2.boundingRect()** 查找得到。

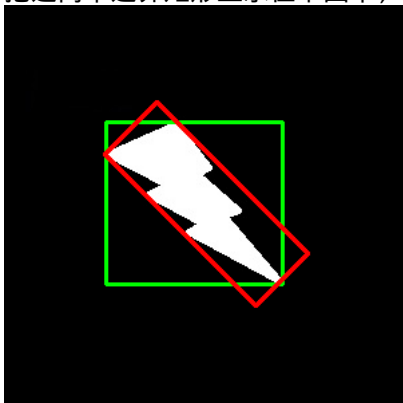
(  $x, y$  ) 为矩形左上角的坐标, (  $w, h$  ) 是矩形的宽和高。

```
x,y,w,h = cv2.boundingRect(cnt)
img = cv2.rectangle(img,(x,y),(x+w,y+h),(0,255,0),2)
```

**旋转的边界矩形** 这个边界矩形是面积最小的, 因为它考虑了对象的旋转。用到的函数为 **cv2.minAreaRect()**。返回的是一个 **Box2D** 结构, 其中包含矩形左上角角点的坐标 (  $x, y$  ), 矩形的宽和高 (  $w, h$  ), 以及旋转角度。但是要绘制这个矩形需要矩形的 4 个角点, 可以通过函数 **cv2.boxPoints()** 获得。

```
x,y,w,h = cv2.boundingRect(cnt)
img = cv2.rectangle(img,(x,y),(x+w,y+h),(0,255,0),2)
```

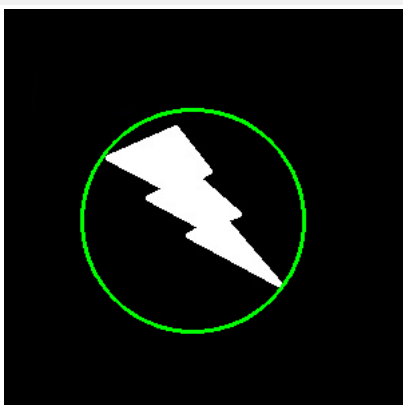
把这两中边界矩形显示在下图中，其中绿色的为直矩形，红的为旋转矩形。



### 21.2.8 最小外接圆

函数 **cv2.minEnclosingCircle()** 可以帮我们找到一个对象的外切圆。它是所有能够包括对象的圆中面积最小的一个。

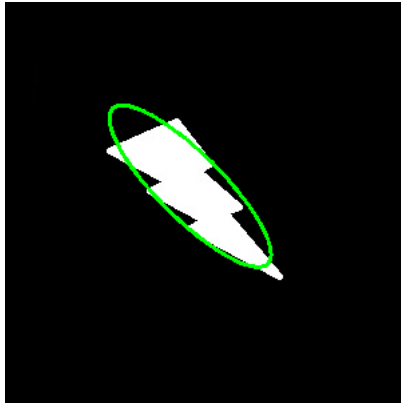
```
(x,y),radius = cv2.minEnclosingCircle(cnt)
center = (int(x),int(y))
radius = int(radius)
img = cv2.circle(img,center,radius,(0,255,0),2)
```



### 21.2.9 椭圆拟合

使用的函数为 **cv2.ellipse()**，返回值其实就是旋转边界矩形的内切圆。

```
ellipse = cv2.fitEllipse(cnt)
im = cv2.ellipse(im,ellipse,(0,255,0),2)
```



### 21.2.10 直线拟合

我们可以根据一组点拟合出一条直线，同样我们也可以为图像中的白色点拟合出一条直线。



```

rows,cols = img.shape[:2]

#cv2.fitLine(points, distType, param, reps, aeps[, line ]) → line

#points – Input vector of 2D or 3D points, stored in std::vector<> or Mat.

#line – Output line parameters. In case of 2D fitting, it should be a vector of
#4 elements (like Vec4f) - (vx, vy, x0, y0), where (vx, vy) is a normalized
#vector collinear to the line and (x0, y0) is a point on the line. In case of
#3D fitting, it should be a vector of 6 elements (like Vec6f) - (vx, vy, vz,
#x0, y0, z0), where (vx, vy, vz) is a normalized vector collinear to the line
#and (x0, y0, z0) is a point on the line.

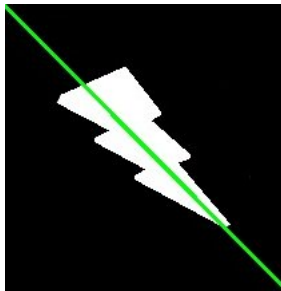
#distType – Distance used by the M-estimator
#distType=CV_DIST_L2
#p(r) = r2 / 2 (the simplest and the fastest least-squares method)

#param – Numerical parameter ( C ) for some types of distances. If it is 0, an optimal value
#is chosen.

#reps – Sufficient accuracy for the radius (distance between the coordinate origin and the
#line).

#aeps – Sufficient accuracy for the angle. 0.01 would be a good default value for reps and
#aeps.
[vx,vy,x,y] = cv2.fitLine(cnt, cv2.DIST_L2,0,0.01,0.01)
lefty = int((-x*vy/vx) + y)
righty = int(((cols-x)*vy/vx)+y)
img = cv2.line(img,(cols-1,righty),(0,lefty),(0,255,0),2)

```



**更多资源**

**练习**

## 21.3 轮廓的性质

本小节我们将要学习提取一些经常使用的对象特征。你可以在[Matlab regionprops documentation](#) 更多的图像特征。

### 21.3.1 长宽比

边界矩形的宽高比

$$Aspect\ Ratio = \frac{Width}{Height}$$

```
x,y,w,h = cv2.boundingRect(cnt)
aspect_ratio = float(w)/h
```

### 21.3.2 Extent

轮廓面积与边界矩形面积的比。

$$Extent = \frac{Object\ Area}{Bounding\ Rectangle\ Area}$$

```
area = cv2.contourArea(cnt)
x,y,w,h = cv2.boundingRect(cnt)
rect_area = w*h
extent = float(area)/rect_area
```

### 21.3.3 Solidity

轮廓面积与凸包面积的比。

$$Solidity = \frac{Contour\ Area}{Convex\ Hull\ Area}$$

```
area = cv2.contourArea(cnt)
hull = cv2.convexHull(cnt)
hull_area = cv2.contourArea(hull)
solidity = float(area)/hull_area
```

#### 21.3.4 Equivalent Diameter

与轮廓面积相等的圆形的直径

$$\text{Equivalent Diameter} = \sqrt{\frac{4 \times \text{Contour Area}}{\pi}}$$

```
area = cv2.contourArea(cnt)
equi_diameter = np.sqrt(4*area/np.pi)
```

#### 21.3.5 方向

对象的方向，下面的方法还会返回长轴和短轴的长度

```
(x,y),(MA,ma),angle = cv2.fitEllipse(cnt)
```

#### 21.3.6 掩模和像素点

有时我们需要构成对象的所有像素点，我们可以这样做：

```

mask = np.zeros(imgray.shape,np.uint8)

# 这里一定要使用参数-1, 绘制填充的的轮廓
cv2.drawContours(mask,[cnt],0,255,-1)

#Returns a tuple of arrays, one for each dimension of a,
#containing the indices of the non-zero elements in that dimension.
#The result of this is always a 2-D array, with a row for
#each non-zero element.
#To group the indices by element, rather than dimension, use:
#transpose(nonzero(a))
#>>> x = np.eye(3)
#>>> x
#array([[ 1.,  0.,  0.],
#       [ 0.,  1.,  0.],
#       [ 0.,  0.,  1.]])
#>>> np.nonzero(x)
#(array([0, 1, 2]), array([0, 1, 2]))
#>>> x[np.nonzero(x)]
#array([ 1.,  1.,  1.])
#>>> np.transpose(np.nonzero(x))
#array([[0, 0],
#       [1, 1],
#       [2, 2]])

pixelpoints = np.transpose(np.nonzero(mask))
#pixelpoints = cv2.findNonZero(mask)

```

这里我们是用两种方法，第一种方法使用了 Numpy 函数，第二种使用了 OpenCV 函数。结果相同，但还是有点不同。Numpy 给出的坐标是 **(row, colum)**

形式的。而 OpenCV 给出的格式是 **(x, y)** 形式的。所以这两个结果基本是可以互换的。row=x, column=y。

### 21.3.7 最大值和最小值及它们的位置

我们可以使用掩模图像得到这些参数。

```

min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(imgray,mask = mask)

```

### 21.3.8 平均颜色及平均灰度

我们也可以使用相同的掩模求一个对象的平均颜色或平均灰度

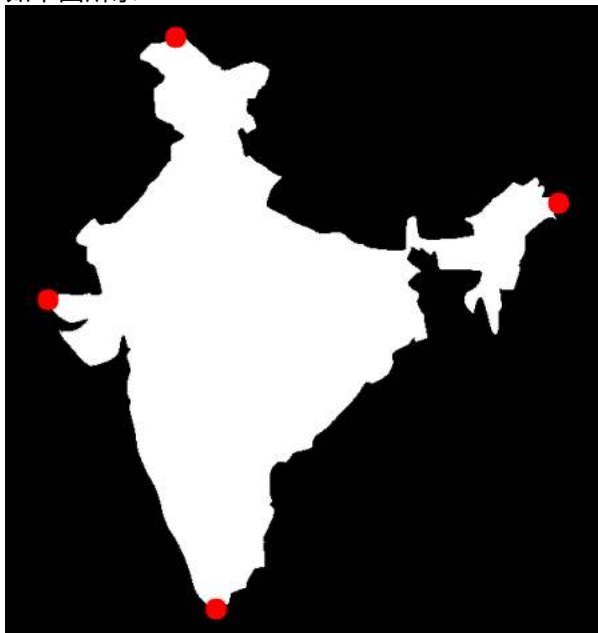
```
mean_val = cv2.mean(im,mask = mask)
```

### 21.3.9 极点

一个对象最上面，最下面，最左边，最右边的点。

```
leftmost = tuple(cnt[cnt[:, :, 0].argmin()][0])  
rightmost = tuple(cnt[cnt[:, :, 0].argmax()][0])  
topmost = tuple(cnt[cnt[:, :, 1].argmin()][0])  
bottommost = tuple(cnt[cnt[:, :, 1].argmax()][0])
```

如下图所示：



#### 更多资源

#### 练习

1. Matlab regionprops 文档中还有一些图像特征我们在这里没有讲到，你可以尝试着使用 Python 和 OpenCV 来实现他们。

## 21.4 轮廓：更多函数

### 目标

我们要学习

- 凸缺陷，以及如何找凸缺陷
- 找某一点到一个多边形的最短距离
- 不同形状的匹配

### 原理与代码

#### 21.4.1 凸缺陷

前面我们已经学习了轮廓的凸包，对象上的任何凹陷都被成为凸缺陷。

OpenCV 中有一个函数 **cv.convexityDefect()** 可以帮助我们找到凸缺陷。函数调用如下：

```
hull = cv2.convexHull(cnt,returnPoints = False)
defects = cv2.convexityDefects(cnt,hull)
```

**注意：**如果要查找凸缺陷，在使用函数 **cv2.convexHull** 找凸包时，参数 **returnPoints** 一定要是 **False**。

它会返回一个数组，其中每一行包含的值是 [起点，终点，最远的点，到最远点的近似距离]。我们可以在一张图上显示它。我们将起点和终点用一条绿线连接，在最远点画一个圆圈，要记住的是返回结果的前三个值是轮廓点的索引。所以我们还要到轮廓点中去找它们。

```

import cv2
import numpy as np

img = cv2.imread('star.jpg')
img_gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(img_gray, 127, 255,0)
contours,hierarchy = cv2.findContours(thresh,2,1)
cnt = contours[0]

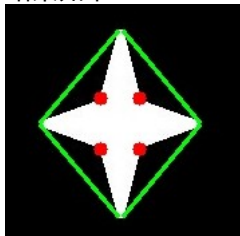
hull = cv2.convexHull(cnt,returnPoints = False)
defects = cv2.convexityDefects(cnt,hull)

for i in range(defects.shape[0]):
    s,e,f,d = defects[i,0]
    start = tuple(cnt[s][0])
    end = tuple(cnt[e][0])
    far = tuple(cnt[f][0])
    cv2.line(img,start,end,[0,255,0],2)
    cv2.circle(img,far,5,[0,0,255],-1)

cv2.imshow('img',img)
cv2.waitKey(0)
cv2.destroyAllWindows()

```

结果如下：



#### 21.4.2 Point Polygon Test

求解图像中的一个点到一个对象轮廓的最短距离。如果点在轮廓的外部，返回值为负。如果在轮廓上，返回值为 0。如果在轮廓内部，返回值为正。

下面我们以点 ( 50, 50 ) 为例：

```

dist = cv2.pointPolygonTest(cnt,(50,50),True)

```

此函数的第三个参数是 **measureDist**。如果设置为 **True**，就会计算最短距离。如果是 **False**，只会判断这个点与轮廓之间的位置关系（返回值为 +1, -1, 0）。

**注意：**如果你不需要知道具体距离，建议你将第三个参数设为 **False**，这样速度会提高 2 到 3 倍。

### 21.4.3 形状匹配

函数 **cv2.matchShape()** 可以帮我们比较两个形状或轮廓的相似度。如果返回值越小，匹配越好。它是根据 **Hu 矩**来计算的。文档中对不同的方法都有解释。

我们试着将下面的图形进行比较：

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jan 13 20:57:38 2014

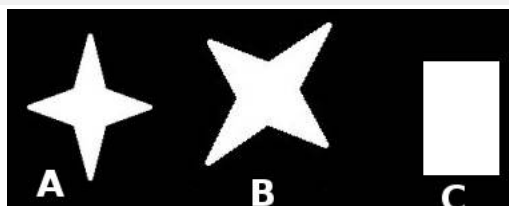
@author: duan
"""

import cv2
import numpy as np

img1 = cv2.imread('star.jpg',0)
img2 = cv2.imread('star2.jpg',0)

ret, thresh = cv2.threshold(img1, 127, 255,0)
ret, thresh2 = cv2.threshold(img2, 127, 255,0)
contours,hierarchy = cv2.findContours(thresh,2,1)
cnt1 = contours[0]
contours,hierarchy = cv2.findContours(thresh2,2,1)
cnt2 = contours[0]

ret = cv2.matchShapes(cnt1,cnt2,1,0.0)
print ret
```



我得到的结果是：

- A 与自己匹配 0.0
- A 与 B 匹配 0.001946
- A 与 C 匹配 0.326911



看见了吗，及时发生了旋转对匹配的结果影响也不是非常大。

**注意：****Hu 矩**是归一化中心矩的线性组合，之所以这样做是为了能够获取代表图像的某个特征的矩函数，这些矩函数对某些变化如缩放，旋转，镜像映射（除了  $h1$ ）具有不变形。此段摘自《学习 OpenCV》中文版。

## 更多资源

### 练习

1. 创建一个小程序，可以将图片上的点绘制成不同的颜色，颜色是根据这个点到轮廓的距离来决定的。要使用的函数：**cv2.pointPolygonTest()**。
2. 使用函数 **cv2.matchShapes()** 匹配带有字母或者数字的图片。

## 21.5 轮廓的层次结构

### 目标

现在我们要学习轮廓的层次结构了，比如轮廓之间的父子关系。

### 更多资源

### 原理

在前面的内容中我们使用函数 `cv2.findContours` 来查找轮廓，我们需要传入一个参数：轮廓提取模式（`Contour_Retrieval_Mode`）。我们总是把它设置为 `cv2.RETR_LIST` 或者是 `cv2.RETR_TREE`，效果还可以。但是它们到底代表什么呢？

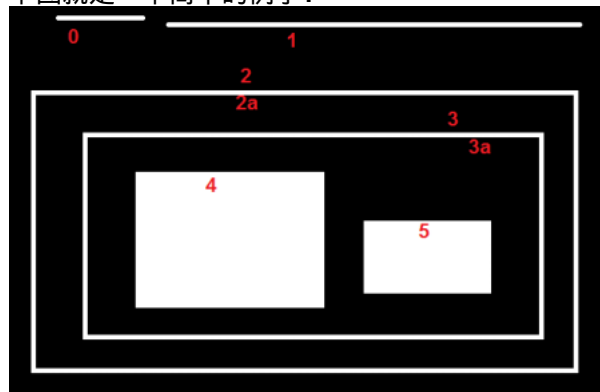
同时，我们得到的结果包含 3 个数组，第一个图像，第二个是轮廓，第三个是层次结构。但是我们从来没有用过层次结构。层次结构是用来干嘛的呢？层次结构与轮廓提取模式有什么关系呢？

这就是我们本节要讲的。

### 21.5.1 什么是层次结构

通常我们使用函数 `cv2.findContours` 在图片中查找一个对象。有时对象可能位于不同的位置。还有些情况，一个形状在另外一个形状的内部。这种情况下我们称外部的形状为父，内部的形状为子。按照这种方式分类，一幅图像中的所有轮廓之间就建立父子关系。这样我们就可以确定一个轮廓与其他轮廓是怎样连接的，比如它是不是某个轮廓的子轮廓，或者是父轮廓。这种关系就成为组织结构

下图就是一个简单的例子：



在这幅图像中，我给这几个形状编号为 0-5。2 和 2a 分别代表最外边矩形的外轮廓和内轮廓。

在这里边轮廓 0，1，2 在外部或最外边。我们可以称他们为（组织结构）0 级，简单来说就是他们属于同一级。

接下来轮廓 2a。我们把它当成轮廓 2 的子轮廓。它就成为（组织结构）第 1 级。同样轮廓 3 是轮廓 2 的子轮廓，成为（组织结构）第 3 级。最后轮廓 4,5 是轮廓 3a 的子轮廓，成为（组织结构）4 级（最后一级）。按照这种方式给这些形状编号，我们可以说轮廓 4 是轮廓 3a 的子轮廓（当然轮廓 5 也是）。

我说这么多就是为了解释层次结构，外轮廓，子轮廓，父轮廓，子轮廓等。现在让我们进入 OpenCV 吧。

### 21.5.2 OpenCV 中层次结构

不管层次结构是什么样的，每一个轮廓都包含自己的信息：谁是父，谁是子等。OpenCV 使用一个含有四个元素的数组表示。**[Next, Previous, First\_Child, Parent]**。

**Next** 表示同一级组织结构中的下一个轮廓。

以上图中的轮廓 0 为例，轮廓 1 就是他的 **Next**。同样，轮廓 1 的 **Next** 是 2，**Next=2**。

那轮廓 2 呢？在同一级没有 **Next**。这时 **Next=-1**。而轮廓 4 的 **Next** 为 5，所以它的 **Next=5**。

**Previous** 表示同一级结构中的前一个轮廓。

与前面一样，轮廓 1 的 **Previous** 为轮廓 0，轮廓 2 的 **Previous** 为轮廓 1。轮廓 0 没有 **Previous**，所以 **Previous=-1**。

**First\_Child** 表示它的第一个子轮廓。

没有必要再解释了，轮廓 2 的子轮廓为 2a。所以它的 **First\_Child** 为 2a。那轮廓 3a 呢？它有两个子轮廓。但是我们只要第一个子轮廓，所以是轮廓 4（按照从上往下，从左往右的顺序排序）。

**Parent** 表示它的父轮廓。

与 **First\_Child** 刚好相反。轮廓 4 和 5 的父轮廓是轮廓 3a。而轮廓 3a 的父轮廓是 3。

**注意：**如果没有父或子，就为 -1。

现在我么了解了 OpenCV 中的轮廓组织结构。我们还是根据上边的图片再学习一下 OpenCV 中的轮廓检索模式。

**cv2.RETR\_LIST, cv2.RETR\_TREE, cv2.RETR\_CCOMP, cv2.RETR\_EXTERNAL** 到底代表什么意思？

### 21.5.3 轮廓检索模式

**RETR\_LIST** 从解释的角度来看，这中应是最简单的。它只是提取所有的轮廓，而不去创建任何父子关系。换句话说就是“人人平等”，它们属于同一级组织轮廓。

所以在这种情况下，组织结构数组的第三和第四个数都是 -1。但是，很明显，Next 和 Previous 要有对应的值，你可以自己试着看看。

下面就是我得到的结果，每一行是对应轮廓的组织结构细节。例如，第一行对应的是轮廓 0。下一个轮廓为 1，所以 Next=1。前面没有其他轮廓，所以 Previous=0。接下来的两个参数就是 -1，与刚才我们说的一样。

```
>>> hierarchy
array([[[ 1, -1, -1, -1],
        [ 2,  0, -1, -1],
        [ 3,  1, -1, -1],
        [ 4,  2, -1, -1],
        [ 5,  3, -1, -1],
        [ 6,  4, -1, -1],
        [ 7,  5, -1, -1],
        [-1,  6, -1, -1]]])
```

如果你不关心轮廓之间的关系，这是一个非常好的选择。

**RETR\_EXTERNAL** 如果你选择这种模式的话，只会返回最外边的的轮廓，所有的子轮廓都会被忽略掉。

所以在上图中使用这种模式的话只会返回最外边的轮廓（第 0 级）：轮廓 0，1，2。下面是我选择这种模式得到的结果：

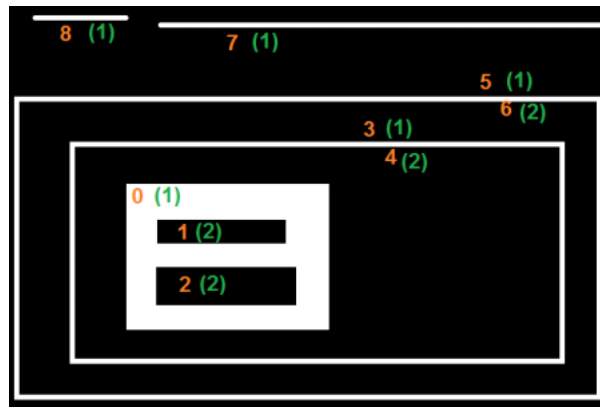
```
>>> hierarchy
array([[[ 1, -1, -1, -1],
        [ 2,  0, -1, -1],
        [-1,  1, -1, -1]])
```

当你只想得到最外边的轮廓时，你可以选择这种模式。这在有些情况下很有用。

**RETR\_CCOMP** 在这种模式下会返回所有的轮廓并将轮廓分为两级组织结构。例如，一个对象的外轮廓为第 1 级组织结构。而对象内部中空洞的轮廓为第 2 级组织结构，空洞中的任何对象的轮廓又是第 1 级组织结构。空洞的组织结构为第 2 级。

想象一下一副黑底白字的图像，图像中是数字 0。0 的外边界属于第一级组织结构，0 的内部属于第 2 级组织结构。

我们可以以下图为例简单介绍一下。我们已经用红色数字为这些轮廓编号，并用绿色数字代表它们的组织结构。顺序与 OpenCV 检测轮廓的顺序一直。



现在我们考虑轮廓 0，它的组织结构为第 1 级。其中有两个空洞 1 和 2，它们属于第 2 级组织结构。所以对于轮廓 0 来说跟他属于同一级组织结构的下一个 (Next) 是轮廓 3，并且没有 Previous。它的 First\_Child 为轮廓 1，组织结构为 2。由于它是第 1 级，所以没有父轮廓。因此它的组织结构数组为 [3, -1, 1, -1]。

现在是轮廓 1，它是第 2 级。处于同一级的下一个轮廓为 2。没有 Previous，也没有 Child，(因为是第 2 级所以有父轮廓) 父轮廓是 0。所以数组是 [2, -1, -1, 0]。

轮廓 2：它是第 2 级。在同一级的组织结构中没有 Next。Previous 为轮廓 1。没有子，父轮廓为 0，所以数组是 [-1, 1, -1, 0]

轮廓 3：它是第 1 级。在同一级的组织结构中 Next 为 5。Previous 为轮廓 0。子为 4，没有父轮廓，所以数组是 [5, 0, 4, -1]

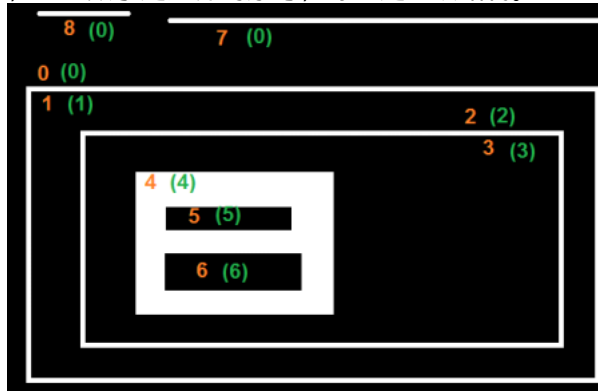
轮廓 4：它是第 2 级。在同一级的组织结构中没有 Next。没有 Previous，没有子，父轮廓为 3，所以数组是 [-1, -1, -1, 3]

下面是我得到的答案：

```
>>> hierarchy
array([[[ 3, -1,  1, -1],
        [ 2, -1, -1,  0],
        [-1,  1, -1,  0],
        [ 5,  0,  4, -1],
        [-1, -1, -1,  3],
        [ 7,  3,  6, -1],
        [-1, -1, -1,  5],
        [ 8,  5, -1, -1],
        [-1,  7, -1, -1]]])
```

**RETR\_TREE** 终于到最后一个了，也是最完美的一个。这种模式下会返回所有轮廓，并且创建一个完整的组织结构列表。它甚至会告诉你谁是爷爷，爸爸，儿子，孙子等。

还是以上图为例，使用这种模式，对 OpenCV 返回的结果重新排序并分析它，红色数字是边界的序号，绿色是组织结构。



轮廓 0 的组织结构为 0，同一级中 Next 为 7，没有 Previous。子轮廓是 1，没有父轮廓。所以数组是 [7, -1, 1, -1]。

轮廓 1 的组织结构为 1，同一级中没有其他，没有 Previous。子轮廓是 2，父轮廓为 0。所以数组是 [-1, -1, 2, 0]。

剩下的自己试试计算一下吧。下面是结果：

```
>>> hierarchy
array([[[ 7, -1,  1, -1],
        [-1, -1,  2,  0],
        [-1, -1,  3,  1],
        [-1, -1,  4,  2],
        [-1, -1,  5,  3],
        [ 6, -1, -1,  4],
        [-1,  5, -1,  4],
        [ 8,  0, -1, -1],
        [-1,  7, -1, -1]]])
```

**更多资源**

**练习**

## 22 直方图

### 22.1 直方图的计算，绘制与分析

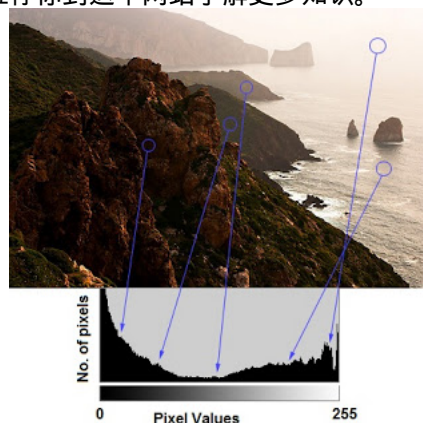
#### 目标

- 使用 OpenCV 或 Numpy 函数计算直方图
- 使用 Opencv 或者 Matplotlib 函数绘制直方图
- 将要学习的函数有：`cv2.calcHist()`，`np.histogram()`

#### 原理

什么是直方图呢？通过直方图你可以对整幅图像的灰度分布有一个整体的了解。直方图的 x 轴是灰度值（0 到 255），y 轴是图片中具有同一个灰度值的点的数目。

直方图其实就是对图像的另一种解释。一下图为例，通过直方图我们可以对图像的对比度，亮度，灰度分布等有一个直观的认识。几乎所有的图像处理软件都提供了直方图分析功能。下图来自[Cambridge in Color website](#)，强烈推荐你到这个网站了解更多知识。



让我们来一起看看这幅图片和它的直方图吧。（要记住，直方图是根据灰度图像绘制的，而不是彩色图像）。直方图的左边区域像是暗一点的像素数量，右侧显示了亮一点的像素的数量。从这幅图上你可以看到灰暗的区域比亮的区域要大，而处于中间部分的像素点很少。

#### 22.1.1 统计直方图

现在我们知道什么是直方图了，那怎样获得一副图像的直方图呢？OpenCV 和 Numpy 都有内置函数做这件事。在使用这些函数之前我们有必要了解一下直方图相关的术语。

**BINS:** 上面的直方图显示了每个灰度值对应的像素数。如果像素值为 0 到 255, 你就需要 256 个数来显示上面的直方图。但是, 如果你不需要知道每一个像素值的像素点数目, 而只希望知道两个像素值之间的像素点数目怎么办呢? 举例来说, 我们想知道像素值在 0 到 15 之间的像素点的数目, 接着是 16 到 31,..., 240 到 255。我们只需要 16 个值来绘制直方图。[OpenCV Tutorials on histograms](#)中例子所演示的内容。

那到底怎么做呢? 你只需要把原来的 256 个值等分成 16 小组, 取每组的总和。而这里的每一个小组就被成为 BIN。第一个例子中有 256 个 BIN, 第二个例子中有 16 个 BIN。在 OpenCV 的文档中用 histSize 表示 BINS。

**DIMS:** 表示我们收集数据的参数数目。在本例中, 我们对收集到的数据只考虑一件事: 灰度值。所以这里就是 1。

**RANGE:** 就是要统计的灰度值范围, 一般来说为 [0, 256], 也就是说所有的灰度值

**使用 OpenCV 统计直方图** 函数 `cv2.calcHist` 可以帮助我们统计一幅图像的直方图。我们一起来熟悉一下这个函数和它的参数:

```
cv2.calcHist(images, channels, mask, histSize, ranges[, hist[, accumulate]])
```

1. **images:** 原图像 (图像格式为 uint8 或 float32)。当传入函数时应该用中括号 [] 括起来, 例如: [img]。
2. **channels:** 同样需要用中括号括起来, 它会告诉函数我们要统计那幅图像的直方图。如果输入图像是灰度图, 它的值就是 [0]; 如果是彩色图像的话, 传入的参数可以是 [0], [1], [2] 它们分别对应着通道 B, G, R。
3. **mask:** 掩模图像。要统计整幅图像的直方图就把它设为 None。但是如果你想统计图像某一部分的直方图的话, 你就需要制作一个掩模图像, 并使用它。(后边有例子)
4. **histSize:** BIN 的数目。也应该用中括号括起来, 例如: [256]。
5. **ranges:** 像素值范围, 通常为 [0, 256]

让我们从一副简单图像开始吧。以灰度格式加载一幅图像并统计图像的直方图。

```
img = cv2.imread('home.jpg',0)
# 别忘了中括号 [img],[0],None,[256],[0,256], 只有 mask 没有中括号
hist = cv2.calcHist([img],[0],None,[256],[0,256])
```

hist 是一个 256x1 的数组, 每一个值代表了与灰度值对应的像素点数目。



**使用 Numpy 统计直方图** Numpy 中的函数 **np.histogram()** 也可以帮我们统计直方图。你也可以尝试一下下面的代码：

```
#img.ravel() 将图像转成一维数组，这里没有中括号。
hist,bins = np.histogram(img.ravel(),256,[0,256])
```

hist 与上面计算的一样。但是这里的 bins 是 257，因为 Numpy 计算 bins 的方式为：0-0.99,1-1.99,2-2.99 等。所以最后一个范围是 255-255.99。为了表示它，所以在 bins 的结尾加上了 256。但是我们不需要 256，到 255 就够了。

**其他：**Numpy 还有一个函数 **np.bincount()**，它的运行速度是 **np.histogram** 的十倍。所以对于一维直方图，我们最好使用这个函数。使用 **np.bincount** 时别忘了设置 **minlength=256**。例如，**hist=np.bincount(img.ravel(), minlength=256)**

**注意：**OpenCV 的函数要比 **np.histogram()** 快 40 倍。所以坚持使用 OpenCV 函数。

现在是时候学习绘制直方图了。

### 22.1.2 绘制直方图

有两种方法来绘制直方图：

1. Short Way (简单方法)：使用 Matplotlib 中的绘图函数。
2. Long Way (复杂方法)：使用 OpenCV 绘图函数

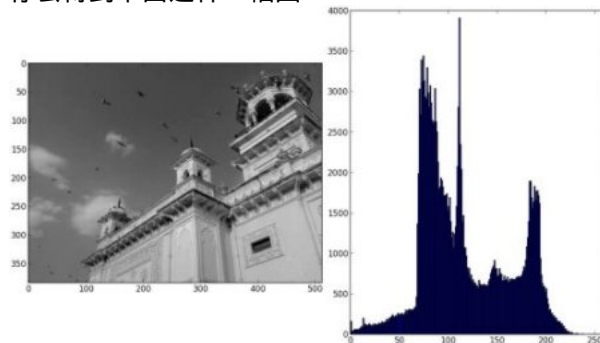
**使用 Matplotlib** Matplotlib 中有直方图绘制函数：**matplotlib.pyplot.hist()**

它可以直接统计并绘制直方图。你应该使用函数 **calcHist()** 或 **np.histogram()** 统计直方图。代码如下：

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('home.jpg',0)
plt.hist(img.ravel(),256,[0,256]);
plt.show()
```

你会得到下面这样一幅图：



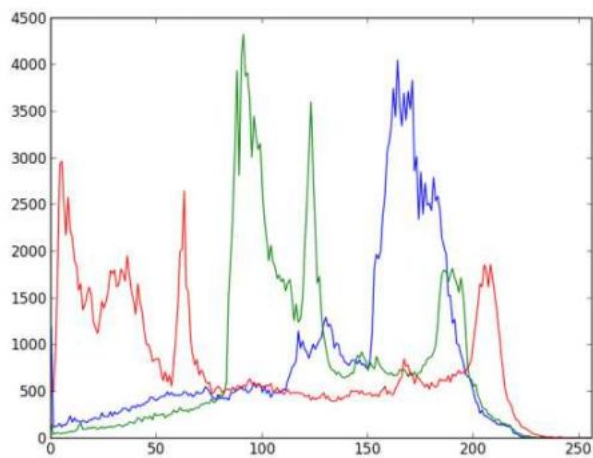
或者你可以只使用 matplotlib 的绘图功能，这在同时绘制多通道（BGR）的直方图，很有用。但是你首先要告诉绘图函数你的直方图数据在哪里。运行一下下面的代码：

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('home.jpg')
color = ('b','g','r')

# 对一个列表或数组既要遍历索引又要遍历元素时
# 使用内置 enumerate 函数会有更加直接，优美的做法
# enumerate 会将数组或列表组成一个索引序列。
# 使我们再获取索引和索引内容的时候更加方便
for i,col in enumerate(color):
    histr = cv2.calcHist([img],[i],None,[256],[0,256])
    plt.plot(histr,color = col)
    plt.xlim([0,256])
plt.show()
```

结果：



从上边的直方图你可以推断出蓝色曲线靠右侧的最多（很明显这些就是天空）

**使用 OpenCV** 使用 OpenCV 自带函数绘制直方图比较麻烦，这里不作介绍，有兴趣可以自己研究。可以参考 OpenCV-Python2 的[官方示例](#)。

### 22.1.3 使用掩模

要统计图像某个局部区域的直方图只需要构建一副掩模图像。将要统计的部分设置成白色，其余部分为黑色，就构成了一副掩模图像。然后把这个掩模图像传给函数就可以了。

```
img = cv2.imread('home.jpg',0)

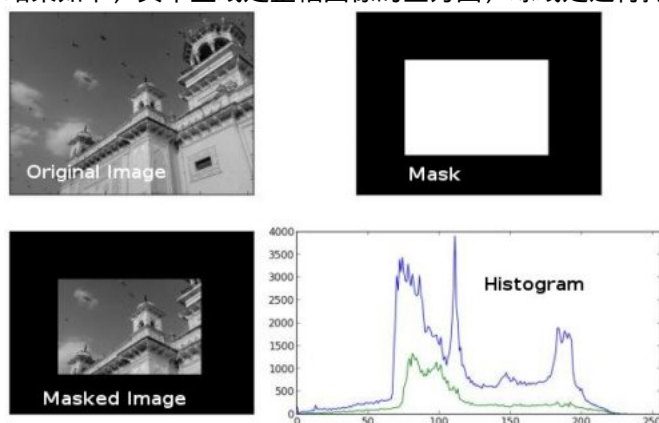
# create a mask
mask = np.zeros(img.shape[:2], np.uint8)
mask[100:300, 100:400] = 255
masked_img = cv2.bitwise_and(img,img,mask = mask)

# Calculate histogram with mask and without mask
# Check third argument for mask
hist_full = cv2.calcHist([img],[0],None,[256],[0,256])
hist_mask = cv2.calcHist([img],[0],mask,[256],[0,256])

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()
```

结果如下，其中蓝线是整幅图像的直方图，绿线是进行掩模之后的直方图。



## 更多资源

1. [Cambridge in Color website](#)

## 练习

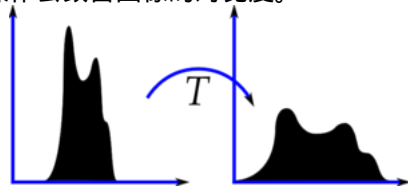
## 22.2 直方图均衡化

### 目标

- 本小节我们要学习直方图均衡化的概念，以及如何使用它来改善图片的对比。

### 原理

想象一下如果一副图像中的大多是像素点的像素值都集中在一个像素值范围之内会怎样呢？例如，如果一幅图片整体很亮，那所有的像素值应该都会很高。但是一副高质量的图像的像素值分布应该很广泛。所以你应该把它的直方图做一个横向拉伸（如下图），这就是直方图均衡化要做的事情。通常情况下这种操作会改善图像的对比度。



推荐你去读读维基百科中关于[直方图均衡化](#)的条目。其中的解释非常给力，读完之后相信你就会对整个过程有一个详细的了解了。我们先看看怎样使用 Numpy 来进行直方图均衡化，然后再学习使用 OpenCV 进行直方图均衡化。

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 16 10:15:23 2014

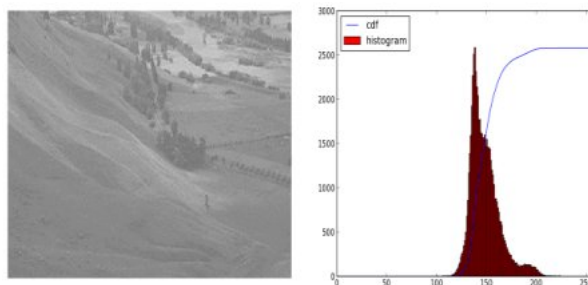
@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

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

#flatten() 将数组变成一维
hist,bins = np.histogram(img.flatten(),256,[0,256])
# 计算累积分布图
cdf = hist.cumsum()
cdf_normalized = cdf * hist.max()/ cdf.max()

plt.plot(cdf_normalized, color = 'b')
plt.hist(img.flatten(),256,[0,256], color = 'r')
plt.xlim([0,256])
plt.legend(('cdf','histogram'), loc = 'upper left')
plt.show()
```



我们可以看出来直方图大部分在灰度值较高的部分，而且分布很集中。而我希望直方图的分布比较分散，能够涵盖整个 x 轴。所以，我们就需要一个变换函数帮助我们把现在的直方图映射到一个广泛分布的直方图中。这就是直方图均衡化要做的事情。

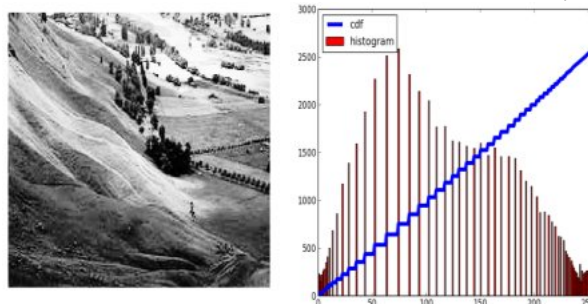
我们现在要找到直方图中的最小值（除了 0），并把它用于 wiki 中的直方图均衡化公式。但是我在这里使用了 Numpy 的掩模数组。对于掩模数组的所有操作都只对 non-masked 元素有效。你可以到 Numpy 文档中获取更多掩模数组的信息。

```
# 构建 Numpy 掩模数组, cdf 为原数组, 当数组元素为 0 时, 掩盖 (计算时被忽略)。  
cdf_m = np.ma.masked_equal(cdf,0)  
cdf_m = (cdf_m - cdf_m.min())*255/(cdf_m.max()-cdf_m.min())  
# 对被掩盖的元素赋值, 这里赋值为 0  
cdf = np.ma.filled(cdf_m,0).astype('uint8')
```

现在就获得了一个表，我们可以通过查表得知与输入像素对应的输出像素的值。我们只需要把这种变换应用到图像上就可以了。

```
img2 = cdf[img]
```

我们再根据前面的方法绘制直方图和累积分布图，结果如下：



另一个重要的特点是，即使我们的输入图片是一个比较暗的图片（不象上

边我们用到到的整体都很亮的图片), 在经过直方图均衡化之后也能得到相同的结果。因此, 直方图均衡化经常用来使所有的图片具有相同的亮度条件的参考工具。这在很多情况下都很有用。例如, 脸部识别, 在训练分类器前, 训练集的所有图片都要先进行直方图均衡化从而使它们达到相同的亮度条件。

### 22.2.1 OpenCV 中的直方图均衡化

OpenCV 中的直方图均衡化函数为 `cv2.equalizeHist()`。这个函数的输入图片仅仅是一副灰度图像, 输出结果是直方图均衡化之后的图像。

下边的代码还是对上边的那幅图像进行直方图均衡化:

```
img = cv2.imread('wiki.jpg',0)
equ = cv2.equalizeHist(img)
res = np.hstack((img,equ))
#stacking images side-by-side
cv2.imwrite('res.png',res)
```

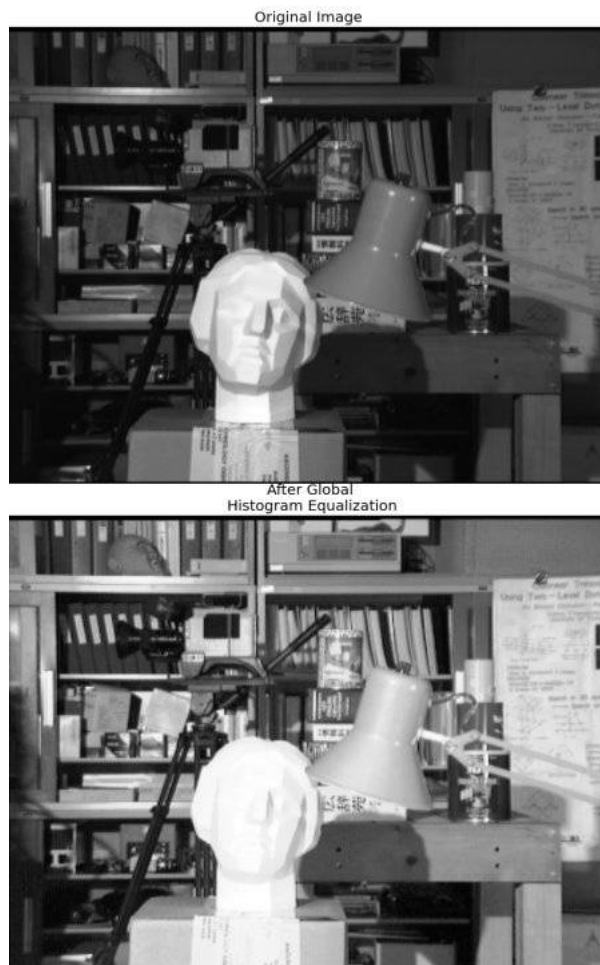


现在你可以拿一些不同亮度的照片自己来试一下了。

当直方图中的数据集中在某一个灰度值范围内时, 直方图均衡化很有用。但是如果像素的变化很大, 而且占据的灰度范围非常广时, 例如: 既有很亮的像素点又有很暗的像素点时。请查看更多资源中的 [SOF](#) 链接。

### 22.2.2 CLAHE 有限对比适应性直方图均衡化

我们上边做的直方图均衡化会改变整个图像的对比度, 但是在很多情况下, 这样做的效果并不好。例如, 下图分别是输入图像和进行直方图均衡化之后的输出图像。



的确在进行完直方图均衡化之后，图片背景的对比度被改变了。但是你再对比一下两幅图像中雕像的面部，由于太亮我们丢失了很多信息。造成这种结果的根本原因在于这幅图像的直方图并不是集中在某一个区域（试着画出它的直方图，你就明白了）。

为了解决这个问题，我们需要使用自适应的直方图均衡化。这种情况下，整幅图像会被分成很多小块，这些小块被称为“tiles”（在 OpenCV 中 tiles 的大小默认是  $8 \times 8$ ），然后再对每一个小块分别进行直方图均衡化（跟前面类似）。所以在每一个的区域中，直方图会集中在某一个小的区域中（除非有噪声干扰）。如果有噪声的话，噪声会被放大。为了避免这种情况的出现要使用对比度限制。对于每个小块来说，如果直方图中的 bin 超过对比度的上限的话，就把其中的像素点均匀分散到其他 bins 中，然后在进行直方图均衡化。最后，为了去除每一个小块之间“人造的”（由于算法造成）边界，再使用双线性差值，对小块进行缝合。

下面的代码显示了如何使用 OpenCV 中的 CLAHE。



```
import numpy as np
import cv2

img = cv2.imread('tsukuba_1.png',0)

# create a CLAHE object (Arguments are optional).
# 不知道为什么我没好到 createCLAHE 这个模块
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
cl1 = clahe.apply(img)

cv2.imwrite('clahe_2.jpg',cl1)
```

下面就是结果了，与前面的结果对比一下，尤其是雕像区域：



### 更多资源

1. 维基百科中的[直方图均衡化](#)。
2. [Masked Arrays in Numpy](#)

关于调整图片对比度 SOF 问题：

1. 在 C 语言中怎样使用 OpenCV 调整图像对比度？
2. 怎样使用 OpenCV 调整图像的对比度和亮度？

### 练习

## 22.3 2D 直方图

### 目标

本节我们会学习如何绘制 2D 直方图，我们会在下一节中使用到它。

### 22.3.1 介绍

在前面的部分我们介绍了如何绘制一维直方图，之所以称为一维，是因为我们只考虑了图像的一个特征：灰度值。但是在 2D 直方图中我们就要考虑两个图像特征。对于彩色图像的直方图通常情况下我们需要考虑每个的颜色（**Hue**）和饱和度（**Saturation**）。根据这两个特征绘制 2D 直方图。

OpenCV 的官方文档中包含一个创建彩色直方图的例子。本节就是要和大家一起来学习如何绘制颜色直方图，这会对我们下一节学习直方图投影有所帮助。

### 22.3.2 OpenCV 中的 2D 直方图

使用函数 `cv2.calcHist()` 来计算直方图既简单又方便。如果要绘制颜色直方图的话，我们首先需要将图像的颜色空间从 BGR 转换到 HSV。（记住，计算一维直方图，要从 BGR 转换到 HSV）。计算 2D 直方图，函数的参数要做如下修改：

- **channels=[0, 1]** 因为我们需要同时处理 H 和 S 两个通道。
- **bins=[180, 256]** H 通道为 180，S 通道为 256。
- **range=[0, 180, 0, 256]** H 的取值范围在 0 到 180，S 的取值范围在 0 到 256。

代码如下：

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 16 19:37:21 2014

@author: duan
"""

import cv2
import numpy as np

img = cv2.imread('home.jpg')
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

hist = cv2.calcHist([hsv], [0, 1], None, [180, 256], [0, 180, 0, 256])
```

这就搞定了，简单吧！

### 22.3.3 Numpy 中 2D 直方图

Numpy 同样提供了绘制 2D 直方图的函数：**np.histogram2d()**。（还记得吗，绘制 1D 直方图时我们使用的是 **np.histogram()**）。

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 16 19:37:21 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('home.jpg')
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

hist, xbins, ybins = np.histogram2d(h.ravel(), s.ravel(), [180, 256], [[0, 180], [0, 256]])
```

第一个参数是 H 通道，第二个参数是 S 通道，第三个参数是 bins 的数目，第四个参数是数值范围。

现在我们要看看如何绘制颜色直方图。

### 22.3.4 绘制 2D 直方图

**方法 1：使用 cv2.imshow()** 我们得到结果是一个 180x256 的两维数组。所以我们可以使用函数 **cv2.imshow()** 来显示它。但是这是一个灰度图，除非我们知道不同颜色 H 通道的值，否则我们根本就不知道那到底代表什么颜色。

**方法 2：使用 Matplotlib()** 我们还可以使用函数 **matplotlib.pyplot.imshow()** 来绘制 2D 直方图，再搭配上不同的颜色图（color\_map）。这样我们会对每个点所代表的数值大小有一个更直观的认识。但是跟前面的问题一样，你还是不知道那个数代表的颜色到底是什么。虽然如此，我还是更喜欢这个方法，它既简单又好用。

**注意：**在使用这个函数时，要记住设置插值参数为 **nearest**。

代码如下：

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 16 19:37:21 2014

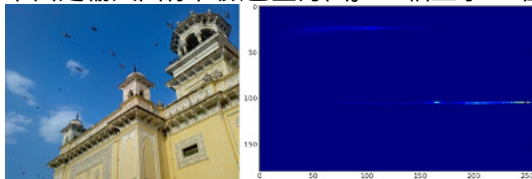
@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('home.jpg')
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
hist = cv2.calcHist( [hsv], [0, 1], None, [180, 256], [0, 180, 0, 256] )

plt.imshow(hist, interpolation = 'nearest')
plt.show()
```

下面是输入图像和颜色直方图。X 轴显示 S 值，Y 轴显示 H 值。

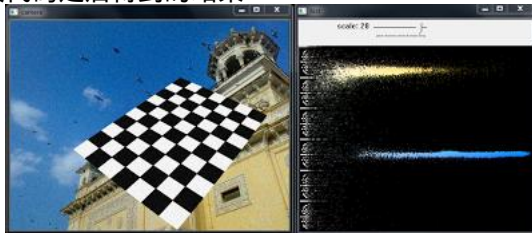


在直方图中，你可以看到在  $H=100$ ， $S=100$  附近有比较高的值。这部分与天的蓝色相对应。同样另一个峰值在  $H=25$  和  $S=100$  附近。这一宫殿的黄色相对应。你可用通过使用图像编辑软件（GIMP）修改图像，然后在绘制直方图看看我说的对不对。

**方法 3: OpenCV 风格** 在官方文档中有一个关于颜色直方图的例子。运行一下这个代码，你看到的颜色直方图也显示了对应的颜色。简单来说就是：输出结果是一副由颜色编码的直方图。效果非常好（虽然要添加很多代码）。

在那个代码中，作者首先创建了一个 HSV 格式的颜色地图，然后把它转换成 BGR 格式。再将得到的直方图与颜色直方图相乘。作者还用了几步来去除小的孤立的点，从而得到了一个好的直方图。

我把对代码的分析留给你们了，自己去玩一下把。下边是对上边的图运行这段代码之后得到的结果：



从直方图中我们可以很清楚的看出它们代表的颜色，蓝色，换色，还有棋盘带来的白色，漂亮 !!!

## 更多资源

## 练习

```
# -*- coding: utf-8 -*-
#!/usr/bin/env python

import numpy as np
import cv2
from time import clock
import sys

import video
#video 模块也是 opencv 官方文档中自带的
if __name__ == '__main__':

    # 构建 HSV 颜色地图
    hsv_map = np.zeros((180, 256, 3), np.uint8)
    # np.indices 可以返回由数组索引构建的新数组。
    # 例如: np.indices((3,2)); 其中(3,2)为原来数组的维度: 行和列。
    # 返回值首先看输入的参数有几维:(3,2)有 2 维, 所以从输出的结果应该是
    # [[a],[b]], 其中包含两个 3 行, 2 列数组。
    # 第二看每一维的大小, 第一维为 3, 所以 a 中的值就 0 到 2 (最大索引数),
    # a 中的每一个值就是它的行索引; 同样的方法得到 b (列索引)
    # 结果就是
    # array([[0, 0],
    #        [1, 1],
    #        [2, 2]],
    #       [[0, 1],
    #        [0, 1],
    #        [0, 1]])

    h, s = np.indices(hsv_map.shape[:2])
    hsv_map[:, :, 0] = h
    hsv_map[:, :, 1] = s
    hsv_map[:, :, 2] = 255
    hsv_map = cv2.cvtColor(hsv_map, cv2.COLOR_HSV2BGR)
    cv2.imshow('hsv_map', hsv_map)

    cv2.namedWindow('hist', 0)
    hist_scale = 10
    def set_scale(val):
        global hist_scale
        hist_scale = val
    cv2.createTrackbar('scale', 'hist', hist_scale, 32, set_scale)

    try: fn = sys.argv[1]
    except: fn = 0
    cam = video.create_capture(fn, fallback='synth:bg=./cpp/baboon.jpg:class=chess:noise=0.05')
```

```

while True:
    flag, frame = cam.read()
    cv2.imshow('camera', frame)
    # 图像金字塔
    # 通过图像金字塔降低分辨率，但不会对直方图有太大影响。
    # 但这种低分辨率，可以很好抑制噪声，从而去除孤立的小点对直方图的影响。
    small = cv2.pyrDown(frame)

    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    # 取 v 通道（亮度）的值。
    # 没有用过这种写法，还是改用最常见的用法。
    #dark = hsv[...,2] < 32
    # 此步操作得到的是一个布尔矩阵，小于 32 的为真，大于 32 的为假。
    dark = hsv[:, :, 2] < 32

    hsv[dark] = 0
    h = cv2.calcHist( [hsv], [0, 1], None, [180, 256], [0, 180, 0, 256] )

#numpy.clip(a, a_min, a_max, out=None)[source]
#Given an interval, values outside the interval are clipped to the interval edges.
#For example, if an interval of [0, 1] is specified, values smaller
#than 0 become 0, and values larger than 1 become 1.
#>>> a = np.arange(10)
#>>> np.clip(a, 1, 8)
#array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
    h = np.clip(h*0.005*hist_scale, 0, 1)

#In numpy one can use the 'newaxis' object in the slicing syntax to create an
#axis of length one. one can also use None instead of newaxis,
#the effect is exactly the same
    #h 从一维变成 3 维
    vis = hsv_map*h[:, :, np.newaxis] / 255.0
    cv2.imshow('hist', vis)

    ch = 0xFF & cv2.waitKey(1)
    if ch == 27:
        break
cv2.destroyAllWindows()

```

## 22.4 直方图反向投影

### 目标

本节我们将要学习直方图反向投影

### 原理

直方图反向投影是由 Michael J. Swain 和 Dana H. Ballard 在他们的文章“Indexing via color histograms”中提出。

那它到底是什么呢？它可以用来做图像分割，或者在图像中找寻我们感兴趣的部分。简单来说，它会输出与输入图像（待搜索）同样大小的图像，其中的每一个像素值代表了输入图像上对应点属于目标对象的概率。用更简单的话来解释，输出图像中像素值越高（越白）的点就越可能代表我们要搜索的目标（在输入图像所在的位置）。这是一个直观的解释。直方图投影经常与 camshift 算法等一起使用。

我们应该怎样来实现这个算法呢？首先我们要为一张包含我们要查找目标的图像创建直方图（在我们的示例中，我们要查找的是草地，其他的都不要）。我们要查找的对象要尽量占满这张图像（换句话说，这张图像上最好是有且仅有我们要查找的对象）。最好使用颜色直方图，因为一个物体的颜色要比它的灰度能更好的被用来进行图像分割与对象识别。接着我们再把这个颜色直方图投影到输入图像中寻找我们的目标，也就是找到输入图像中的每一个像素点的像素值在直方图中对应的概率，这样我们就得到一个概率图像，最后设置适当的阈值对概率图像进行二值化，就这么简单。

### 22.4.1 Numpy 中的算法

此处的算法与上边介绍的算法稍有不同。

首先，我们要创建两幅颜色直方图，目标图像的直方图（'M'），（待搜索）输入图像的直方图（'I'）。



```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 18 10:52:34 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

#roi is the object or region of object we need to find
roi = cv2.imread('rose_red.png')
hsv = cv2.cvtColor(roi,cv2.COLOR_BGR2HSV)

#target is the image we search in
target = cv2.imread('rose.png')
hsvt = cv2.cvtColor(target,cv2.COLOR_BGR2HSV)

# Find the histograms using calcHist. Can be done with np.histogram2d also
M = cv2.calcHist([hsv],[0, 1], None, [180, 256], [0, 180, 0, 256] )
I = cv2.calcHist([hsvt],[0, 1], None, [180, 256], [0, 180, 0, 256] )

```

计算比值:  $R = \frac{M}{I}$ 。反向投影 R，也就是根据 R 这个“调色板”创建一副新的图像，其中的每一个像素代表这个点就是目标的概率。例如  $B(x, y) = R[h(x, y), s(x, y)]$ ，其中 h 为点 (x, y) 处的 hue 值，s 为点 (x, y) 处的 saturation 值。最后加入再一个条件  $B(x, y) = \min[B(x, y), 1]$ 。

```

h,s,v = cv2.split(hsvt)
B = R[h.ravel(),s.ravel()]
B = np.minimum(B,1)
B = B.reshape(hsvt.shape[:2])

```

现在使用一个圆盘算子做卷积， $B = D \times B$ ，其中 D 为卷积核。

```

disc = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
B=cv2.filter2D(B,-1,disc)
B = np.uint8(B)
cv2.normalize(B,B,0,255,cv2.NORM_MINMAX)

```

现在输出图像中灰度值最大的地方就是我们要查找的目标的位置了。如果我们要找的是一个区域，我们就可以使用一个阈值对图像进行二值化，这样就可以得到一个很好的结果了。

```
ret,thresh = cv2.threshold(B,50,255,0)
```

就是这样。

#### 22.4.2 OpenCV 中的反向投影

OpenCV 提供的函数 **cv2.calcBackProject()** 可以用来做直方图反向投影。它的参数与函数 **cv2.calcHist** 的参数基本相同。其中的一个参数是我们要查找目标的直方图。同样再使用目标的直方图做反向投影之前我们应该先对其做归一化处理。返回的结果是一个概率图像，我们再使用一个圆盘形卷积核对其做卷积操作，最后使用阈值进行二值化。下面就是代码和结果：

```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 18 10:08:25 2014

@author: duan
"""

import cv2
import numpy as np

roi = cv2.imread('tar.jpg')
hsv = cv2.cvtColor(roi,cv2.COLOR_BGR2HSV)

target = cv2.imread('roi.jpg')
hsvt = cv2.cvtColor(target,cv2.COLOR_BGR2HSV)

# calculating object histogram
roihist = cv2.calcHist([hsv],[0, 1], None, [180, 256], [0, 180, 0, 256] )

# normalize histogram and apply backprojection
# 归一化: 原始图像, 结果图像, 映射到结果图像中的最小值, 最大值, 归一化类型
#cv2.NORM_MINMAX 对数组的所有值进行转化, 使它们线性映射到最小值和最大值之间
# 归一化之后的直方图便于显示, 归一化之后就成为了 0 到 255 之间的数了。
cv2.normalize(roihist,roihist,0,255,cv2.NORM_MINMAX)
dst = cv2.calcBackProject([hsvt],[0,1],roihist,[0,180,0,256],1)

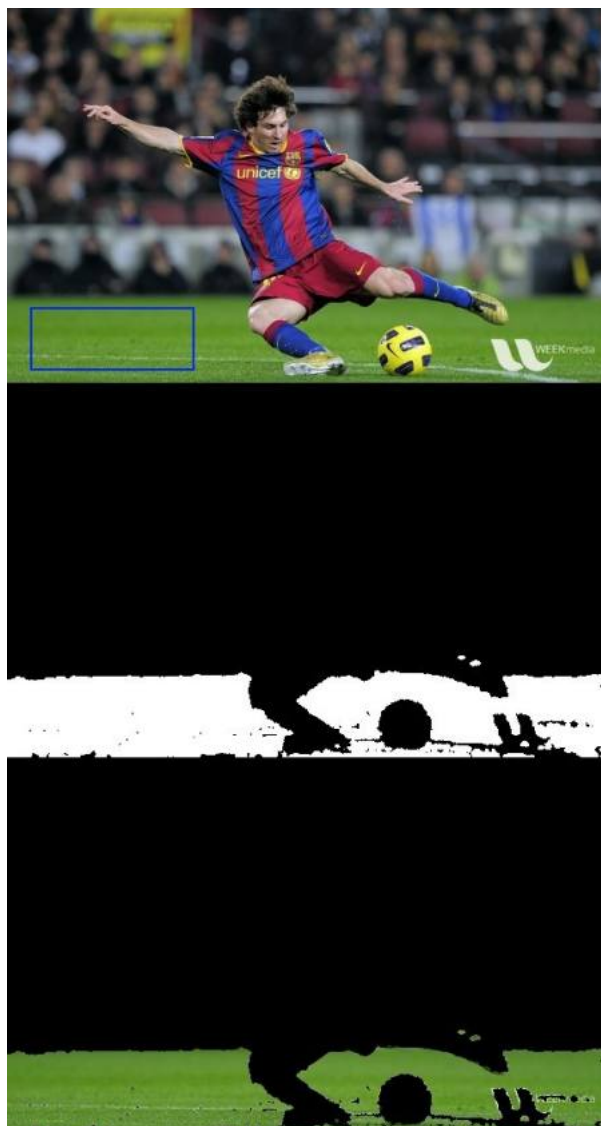
# Now convolute with circular disc
# 此处卷积可以把分散的点连在一起
disc = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
dst=cv2.filter2D(dst,-1,disc)

# threshold and binary AND
ret,thresh = cv2.threshold(dst,50,255,0)
# 别忘了是三通道图像, 因此这里使用 merge 变成 3 通道
thresh = cv2.merge((thresh,thresh,thresh))
# 按位操作
res = cv2.bitwise_and(target,thresh)

res = np.hstack((target,thresh,res))
cv2.imwrite('res.jpg',res)
# 显示图像
cv2.imshow('1',res)
cv2.waitKey(0)

```

下面是我使用的一幅图像。我使用图中蓝色矩形中的区域作为取样对象，再根据这个样本搜索图中所有的类似区域（草地）。



### 更多资源

1. "Indexing via color histograms", Swain, Michael J. , Third international conference on computer vision, 1990.

### 练习

## 23 图像变换

### 23.1 傅里叶变换

#### 目标

本小节我们将要学习：

- 使用 OpenCV 对图像进行傅里叶变换
- 使用 Numpy 中 FFT (快速傅里叶变换) 函数
- 傅里叶变换的一些用处
- 我们将要学习的函数有：`cv2.dft()`，`cv2.idft()` 等

#### 原理

傅里叶变换经常被用来分析不同滤波器的频率特性。我们可以使用 2D 离散傅里叶变换 (DFT) 分析图像的频域特性。实现 DFT 的一个快速算法被称为快速傅里叶变换 (FFT)。关于傅里叶变换的细节知识可以在任意一本图像处理或信号处理的书中找到。请查看本小节中更多资源部分。

对于一个正弦信号： $x(t) = A \sin(2\pi ft)$ ，它的频率为  $f$ ，如果把这个信号转到它的频域表示，我们会在频率  $f$  中看到一个峰值。如果我们的信号是由采样产生的离散信号好组成，我们会得到类似的频谱图，只不过前面是连续的，现在是离散。你可以把图像想象成沿着两个方向采集的信号。所以对图像同时进行 X 方向和 Y 方向的傅里叶变换，我们就会得到这幅图像的频域表示（频谱图）。

更直观一点，对于一个正弦信号，如果它的幅度变化非常快，我们可以说他是高频信号，如果变化非常慢，我们称之为低频信号。你可以把这种想法应用到图像中，图像那里的幅度变化非常大呢？边界点或者噪声。所以我们说边界和噪声是图像中的高频分量（注意这里的高频是指变化非常快，而非出现的次数多）。如果没有如此大的幅度变化我们称之为低频分量。

现在我们看看怎样进行傅里叶变换。

#### 23.1.1 Numpy 中的傅里叶变换

首先我们看看如何使用 Numpy 进行傅里叶变换。Numpy 中的 FFT 包可以帮助我们实现快速傅里叶变换。函数 `np.fft.fft2()` 可以对信号进行频率转换，输出结果是一个复杂的数组。本函数的第一个参数是输入图像，要求是灰度格式。第二个参数是可选的，决定输出数组的大小。输出数组的大小和输入图像大小一样。如果输出结果比输入图像大，输入图像就需要在进行 FFT 前补 0。如果输出结果比输入图像小的话，输入图像就会被切割。

现在我们得到了结果，频率为 0 的部分（直流分量）在输出图像的左上角。如果想让它（直流分量）在输出图像的中心，我们还需要将结果沿两个方向平移  $\frac{N}{2}$ 。函数 `np.fft.fftshift()` 可以帮助我们实现这一步。（这样更容易分析）。进行完频率变换之后，我们就可以构建振幅谱了。

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jan 18 15:30:10 2014

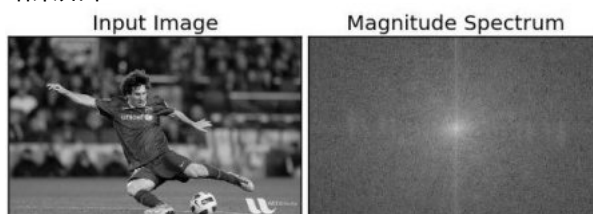
@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('messi5.jpg',0)
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
# 这里构建振幅图的公式没学过
magnitude_spectrum = 20*np.log(np.abs(fshift))

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()
```

结果如下：



我们可以看到输出结果的中心部分更白（亮），这说明低频分量更多。

现在我们可以进行频域变换了，我们就可以在频域对图像进行一些操作了，例如高通滤波和重建图像（DFT 的逆变换）。比如我们可以使用一个 60x60 的矩形窗口对图像进行掩模操作从而去除低频分量。然后再使用函数 `np.fft.ifftshift()` 进行逆平移操作，所以现在直流分量又回到左上角了，左后使用函数 `np.ifft2()` 进行 FFT 逆变换。同样又得到一堆复杂的数字，我们可以对他们取绝对值：

```

rows, cols = img.shape
crow,ccol = rows/2 , cols/2
fshift[crow-30:crow+30, ccol-30:ccol+30] = 0
f_ishift = np.fft.ifftshift(fshift)
img_back = np.fft.ifft2(f_ishift)
# 取绝对值
img_back = np.abs(img_back)

plt.subplot(131),plt.imshow(img, cmap = 'gray')
plt.title('Input Image'), plt.xticks([], plt.yticks([]))
plt.subplot(132),plt.imshow(img_back, cmap = 'gray')
plt.title('Image after HPF'), plt.xticks([], plt.yticks([]))
plt.subplot(133),plt.imshow(img_back)
plt.title('Result in JET'), plt.xticks([], plt.yticks([]))
plt.show()

```

结果如下：



上图的结果显示高通滤波其实是一种边界检测操作。这就是我们在前面图像梯度那一章看到的。同时我们还发现图像中的大部分数据集中在频谱图的低频区域。我们现在已经知道如何使用 Numpy 进行 DFT 和 IDFT 了，接着我们来看看如何使用 OpenCV 进行这些操作。

如果你观察仔细的话，尤其是最后一章 JET 颜色的图像，你会看到一些不自然的东西（如我用红色箭头标出的区域）。看上图那里有些条带装的结构，这被成为振铃效应。这是由于我们使用矩形窗口做掩模造成的。这个掩模被转换成正弦形状时就会出现这个问题。所以一般我们不适用矩形窗口滤波。最好的选择是高斯窗口。

### 23.1.2 OpenCV 中的傅里叶变换

OpenCV 中相应的函数是 `cv2.dft()` 和 `cv2.idft()`。和前面输出的结果一样，但是是双通道的。第一个通道是结果的实数部分，第二个通道是结果的虚数部分。输入图像要首先转换成 `np.float32` 格式。我们来看看如何操作。

```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 18 15:33:18 2014

@author: duan
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('messi5.jpg',0)

dft = cv2.dft(np.float32(img),flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)

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()

```

**注意：**你可以使用函数 `cv2.cartToPolar()`，它会同时返回幅度和相位。

现在我们来逆 DFT。在前面的部分我们实现了一个 HPF（高通滤波），现在我们来逆 LPF（低通滤波）将高频部分去除。其实就是对图像进行模糊操作。首先我们需要构建一个掩模，与低频区域对应的地方设置为 1，与高频区域对应的地方设置为 0。



```

rows, cols = img.shape
crow,ccol = rows/2 , cols/2

# create a mask first, center square is 1, remaining all zeros
mask = np.zeros((rows,cols,2),np.uint8)
mask[crow-30:crow+30, ccol-30:ccol+30] = 1

# apply mask and inverse DFT
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('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])
plt.show()

```

结果如下：



**注意：**OpenCV 中的函数 **cv2.dft()** 和 **cv2.idft()** 要比 Numpy 快。但是 Numpy 函数更加用户友好。关于性能的描述，请看下面的章节。

### 23.1.3 DFT 的性能优化

当数组的大小为某些值时 DFT 的性能会更好。当数组的大小是 2 的指数时 DFT 效率最高。当数组的大小是 2, 3, 5 的倍数时效率也会很高。所以如果你想提高代码的运行效率时，你可以修改输入图像的大小（补 0）。对于 OpenCV 你必须自己手动补 0。但是 Numpy，你只需要指定 FFT 运算的大小，它会自动补 0。

那我们怎样确定最佳大小呢？OpenCV 提供了一个函数：**cv2.getOptimalDFTSize()**。它可以同时被 **cv2.dft()** 和 **np.fft.fft2()** 使用。让我们一起使用 IPython 的魔法命令 **%timeit** 来测试一下吧。

```

In [16]: img = cv2.imread('messi5.jpg',0)
In [17]: rows,cols = img.shape
In [18]: print rows,cols
342 548

In [19]: nrows = cv2.getOptimalDFTSize(rows)
In [20]: ncols = cv2.getOptimalDFTSize(cols)
In [21]: print nrows, ncols
360 57

```

看到了吧，数组的大小从（342，548）变成了（360，576）。现在我们为它补 0，然后看看性能有没有提升。你可以创建一个大的 0 数组，然后把我们的数据拷贝过去，或者使用函数 **cv2.copyMakeBorder()**。

```

nimg = np.zeros((nrows,ncols))
nimg[:rows,:cols] = img

```

或者：

```

right = ncols - cols
bottom = nrows - rows
#just to avoid line breakup in PDF file
bordertype = cv2.BORDER_CONSTANT
nimg = cv2.copyMakeBorder(img,0,bottom,0,right,bordertype, value = 0)

```

现在我们看看 Numpy 的表现：

```

In [22]: %timeit fft1 = np.fft.fft2(img)
10 loops, best of 3: 40.9 ms per loop
In [23]: %timeit fft2 = np.fft.fft2(img,[nrows,ncols])
100 loops, best of 3: 10.4 ms per loop

```

速度提高了 4 倍。我们再看看 OpenCV 的表现：

```
In [24]: %timeit dft1= cv2.dft(np.float32(img),flags=cv2.DFT_COMPLEX_OUTPUT)
100 loops, best of 3: 13.5 ms per loop
In [27]: %timeit dft2= cv2.dft(np.float32(nimg),flags=cv2.DFT_COMPLEX_OUTPUT)
100 loops, best of 3: 3.11 ms per loop
```

也提高了 4 倍，同时我们也会发现 OpenCV 的速度是 Numpy 的 3 倍。你也可以测试一下逆 FFT 的表现。

#### 23.1.4 为什么拉普拉斯算子是高通滤波器？

我在论坛中遇到了一个类似的问题。为什么拉普拉斯算子是高通滤波器？为什么 Sobel 是 HPF？等等。对于第一个问题的答案我们以傅里叶变换的形式给出。我们一起来对不同的算子进行傅里叶变换并分析它们：

```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 18 15:37:28 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

# simple averaging filter without scaling parameter
mean_filter = np.ones((3,3))

# creating a gaussian filter
x = cv2.getGaussianKernel(5,10)
#x.T 为矩阵转置
gaussian = x*x.T

# different edge detecting filters
# scharr in x-direction
scharr = np.array([[-3, 0, 3],
                  [-10,0,10],
                  [-3, 0, 3]])
# sobel in x direction
sobel_x= np.array([[-1, 0, 1],
                  [-2, 0, 2],
                  [-1, 0, 1]])
# sobel in y direction
sobel_y= np.array([[-1,-2,-1],
                  [0, 0, 0],
                  [1, 2, 1]])
# laplacian
laplacian=np.array([[0, 1, 0],
                  [1,-4, 1],
                  [0, 1, 0]])

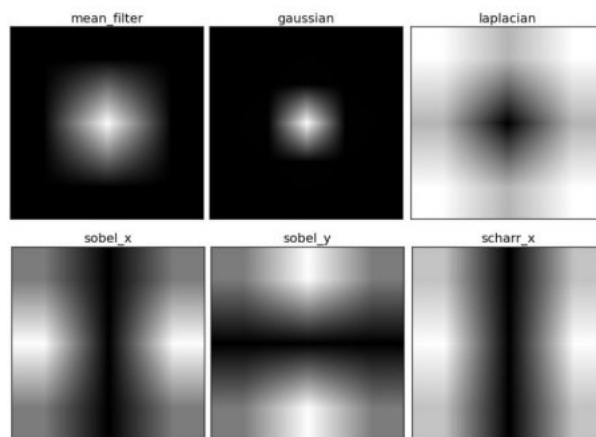
filters = [mean_filter, gaussian, laplacian, sobel_x, sobel_y, scharr]
filter_name = ['mean_filter', 'gaussian','laplacian', 'sobel_x', \
              'sobel_y', 'scharr_x']
fft_filters = [np.fft.fft2(x) for x in filters]
fft_shift = [np.fft.fftshift(y) for y in fft_filters]
mag_spectrum = [np.log(np.abs(z)+1) for z in fft_shift]

for i in xrange(6):
    plt.subplot(2,3,i+1),plt.imshow(mag_spectrum[i],cmap = 'gray')
    plt.title(filter_name[i]), plt.xticks([]), plt.yticks([])

plt.show()

```

结果：



从图像中我们就可以看出每一个算子允许通过那些信号。从这些信息中我们就可以知道那些是 HPF 那是 LPF。

#### 更多资源

1. [An Intuitive Explanation of Fourier Theory](#) by Steven Lehar
2. [Fourier Transform](#) at HIPR
3. [What does frequency domain denote in case of images?](#)

#### 练习

## 24 模板匹配

### 目标

在本节我们要学习：

1. 使用模板匹配在一幅图像中查找目标
2. 函数：`cv2.matchTemplate()`，`cv2.minMaxLoc()`

### 原理

模板匹配是用来在一副大图中搜寻查找模版图像位置的方法。OpenCV 为我们提供了函数：`cv2.matchTemplate()`。和 2D 卷积一样，它也是用模板图像在输入图像（大图）上滑动，并在每一个位置对模板图像和与其对应的输入图像的子区域进行比较。OpenCV 提供了几种不同的比较方法（细节请看文档）。返回的结果是一个灰度图像，每一个像素值表示了此区域与模板的匹配程度。

如果输入图像的大小是  $(W \times H)$ ，模板的大小是  $(w \times h)$ ，输出的结果的大小就是  $(W-w+1, H-h+1)$ 。当你得到这幅图之后，就可以使用函数 `cv2.minMaxLoc()` 来找到其中的最小值和最大值的位置了。第一个值为矩形左上角的点（位置）， $(w, h)$  为模板矩形的宽和高。这个矩形就是找到的模板区域了。

**注意：**如果你使用的比较方法是 `cv2.TM_SQDIFF`，最小值对应的位置才是匹配的区域。

### 24.1 OpenCV 中的模板匹配

我们这里有一个例子：我们在梅西的照片中搜索梅西的面部。所以我们要制作下面这样一个模板：



我们会尝试使用不同的比较方法，这样我们就可以比较一下它们的效果了。

```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 18 16:10:33 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('messi5.jpg',0)
img2 = img.copy()
template = cv2.imread('messi_face.jpg',0)
w, h = template.shape[::-1]

# All the 6 methods for comparison in a list
methods = ['cv2.TM_CCOEFF', 'cv2.TM_CCOEFF_NORMED', 'cv2.TM_CCORR',
           'cv2.TM_CCORR_NORMED', 'cv2.TM_SQDIFF', 'cv2.TM_SQDIFF_NORMED']

for meth in methods:
    img = img2.copy()

    #exec 语句用来执行储存在字符串或文件中的 Python 语句。
    #例如，我们可以在运行时生成一个包含 Python 代码的字符串，然后使用 exec 语句执行这些语句。
    #eval 语句用来计算储存在字符串中的有效 Python 表达式
    method = eval(meth)

    # Apply template Matching
    res = cv2.matchTemplate(img,template,method)
    min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)

    # 使用不同的比较方法，对结果的解释不同
    # If the method is TM_SQDIFF or TM_SQDIFF_NORMED, take minimum
    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)

    cv2.rectangle(img,top_left, bottom_right, 255, 2)

    plt.subplot(121),plt.imshow(res,cmap = 'gray')
    plt.title('Matching Result'), plt.xticks([], plt.yticks([]))
    plt.subplot(122),plt.imshow(img,cmap = 'gray')
    plt.title('Detected Point'), plt.xticks([], plt.yticks([]))
    plt.suptitle(meth)

plt.show()

```

结果如下：

**cv2.TM\_CCOEFF**

Matching Result



Detected Point



**cv2.TM\_CCOEFF\_NORMED**

Matching Result



Detected Point



**cv2.TM\_CCORR**

Matching Result



Detected Point



**cv2.TM\_CCORR\_NORMED**

Matching Result



Detected Point



**cv2.TM\_SQDIFF**





**cv2.TM\_SQDIFF\_NORMED**



我们看到 **cv2.TM\_CCORR** 的效果不想我们想的那么好。

## 24.2 多对象的模板匹配

在前面的部分，我们在图片中搜索梅西的脸，而且梅西只在图片中出现了一次。假如你的目标对象只在图像中出现了很多次怎么办呢？函数 **cv.imMaxLoc()** 只会给出最大值和最小值。此时，我们就要使用阈值了。在下面的例子中我们要经典游戏 Mario 的一张截屏图片中找到其中的硬币。

```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 18 16:33:15 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

img_rgb = cv2.imread('mario.png')
img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_BGR2GRAY)
template = cv2.imread('mario_coin.png',0)
w, h = template.shape[::-1]

res = cv2.matchTemplate(img_gray,template,cv2.TM_CCOEFF_NORMED)
threshold = 0.8

#umpy.where(condition[, x, y])
#Return elements, either from x or y, depending on condition.
#If only condition is given, return condition.nonzero().
loc = np.where( res >= threshold)
for pt in zip(*loc[::-1]):
    cv2.rectangle(img_rgb, pt, (pt[0] + w, pt[1] + h), (0,0,255), 2)

cv2.imwrite('res.png',img_rgb)

```

结果:



更多资源

练习

## 25 Hough 直线变换

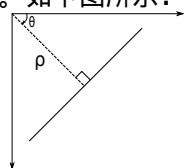
### 目标

- 理解霍夫变换的概念
- 学习如何在一张图片中检测直线
- 学习函数：`cv2.HoughLines()`，`cv2.HoughLinesP()`

### 原理

霍夫变换在检测各种形状的的技术中非常流行，如果你要检测的形状可以用数学表达式写出，你就可以是使用霍夫变换检测它。及时要检测的形状存在一点破坏或者扭曲也可以使用。我们下面就看看如何使用霍夫变换检测直线。

一条直线可以用数学表达式  $y = mx + c$  或者  $\rho = x \cos \theta + y \sin \theta$  表示。 $\rho$  是从原点到直线的垂直距离， $\theta$  是直线的垂线与横轴顺时针方向的夹角（如果你使用的坐标系不同方向也可能不同，我是按 OpenCV 使用的坐标系描述的）。如下图所示：



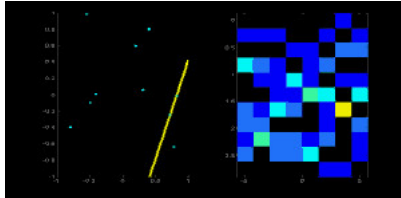
所以如果一条线在原点下方经过， $\rho$  的值就应该大于 0，角度小于 180。但是如果从原点上方经过的话，角度不是大于 180，也是小于 180，但  $\rho$  的值小于 0。垂直的线角度为 0 度，水平线的角度为 90 度。

让我们来看看霍夫变换是如何工作的。每一条直线都可以用  $(\rho, \theta)$  表示。所以首先创建一个 2D 数组（累加器），初始化累加器，所有的值都为 0。行表示  $\rho$ ，列表示  $\theta$ 。这个数组的大小决定了最后结果的准确性。如果你希望角度精确到 1 度，你就需要 180 列。对于  $\rho$ ，最大值为图片对角线的距离。所以如果精确度要达到一个像素的级别，行数就应该与图像对角线的距离相等。

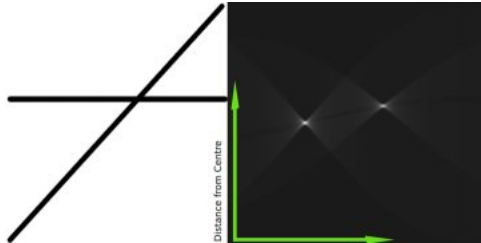
想象一下我们有一个大小为 100x100 的直线位于图像的中央。取直线上的第一个点，我们知道此处的  $(x, y)$  值。把  $x$  和  $y$  带入上边的方程组，然后遍历  $\theta$  的取值：0, 1, 2, 3, ..., 180。分别求出与其对应的  $\rho$  的值，这样我们就得到一系列  $(\rho, \theta)$  的数值对，如果这个数值对在累加器中也存在相应的位置，就在这个位置上加 1。所以现在累加器中的  $(50, 90) = 1$ 。（一个点可能存在与多条直线中，所以对于直线上的每一个点可能是累加器中的多个值同时加 1）。

现在取直线上的第二个点。重复上边的过程。更新累加器中的值。现在累加器中  $(50, 90)$  的值为 2。你每次做的就是更新累加器中的值。对直线上的每个点都执行上边的操作，每次操作完成之后，累加器中的值就加 1，但其他

地方有时会加 1, 有时不会。按照这种方式下去, 到最后累加器中 (50,90) 的值肯定是最大的。如果你搜索累加器中的最大值, 并找到其位置 (50,90), 这就说明图像中有一条直线, 这条直线到原点的距离为 50, 它的垂线与横轴的夹角为 90 度。下面的动画很好的演示了这个过程 (Image Courtesy: Amos Storkey)。



这就是霍夫直线变换工作的方式。很简单, 也许你自己就可以使用 Numpy 搞定它。下图显示了一个累加器。其中最亮的两个点代表了图像中两条直线的参数。(Image courtesy: Wikipedia)。



## 25.1 OpenCV 中的霍夫变换

上面介绍的整个过程在 OpenCV 中都被封装进了一个函数: `cv2.HoughLines()`。返回值就是  $(\rho, \theta)$ 。 $\rho$  的单位是像素,  $\theta$  的单位是弧度。这个函数的第一个参数是一个二值化图像, 所以在进行霍夫变换之前要首先进行二值化, 或者进行 Canny 边缘检测。第二和第三个值分别代表  $\rho$  和  $\theta$  的精确度。第四个参数是阈值, 只有累加其中的值高于阈值时才被认为是一条直线, 也可以把它看成能检测到的直线的最短长度 (以像素点为单位)。

```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 18 19:38:31 2014

@author: duan
"""

import cv2
import numpy as np

img = cv2.imread('dave.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray,50,150,apertureSize = 3)

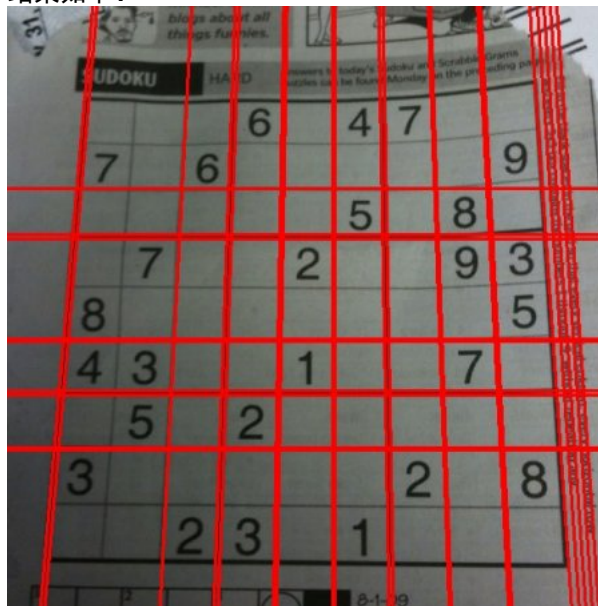
lines = cv2.HoughLines(edges,1,np.pi/180,200)
for rho,theta in lines[0]:
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a*rho
    y0 = b*rho
    x1 = int(x0 + 1000*(-b))
    y1 = int(y0 + 1000*(a))
    x2 = int(x0 - 1000*(-b))
    y2 = int(y0 - 1000*(a))

    cv2.line(img,(x1,y1),(x2,y2),(0,0,255),2)

cv2.imwrite('houghlines3.jpg',img)

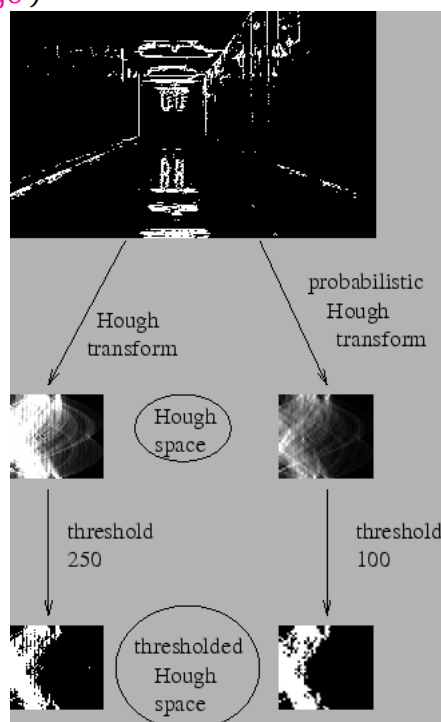
```

结果如下：



## 25.2 Probabilistic Hough Transform

从上边的过程我们可以发现：仅仅是一条直线都需要两个参数，这需要大量的计算。Probabilistic\_Hough\_Transform 是对霍夫变换的一种优化。它不会对每一个点都进行计算，而是从一幅图像中随机选取（是不是也可以使用图像金字塔呢？）一个点集进行计算，对于直线检测来说这已经足够了。但是使用这种变换我们必须降低阈值（总的点数都少了，阈值肯定也要小呀！）。下图是对两种方法的对比。（Image Courtesy : Franck Bettinger's home page）



OpenCV 中使用的 Matas, J. , Galambos, C. 和 Kittler, J.V. 提出的 Progressive Probabilistic Hough Transform。这个函数是 `cv2.HoughLinesP()`。它有两个参数。

- **minLineLength** - 线的最短长度。比这个短的线都会被忽略。
- **MaxLineGap** - 两条线段之间的最大间隔，如果小于此值，这两条直线就被看成是一条直线。

更加给力的是，这个函数的返回值就是直线的起点和终点。而在前面的例子中，我们只得到了直线的参数，而且你必须要找到所有的直线。而在这里一切都很直接很简单。

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jan 18 19:39:33 2014

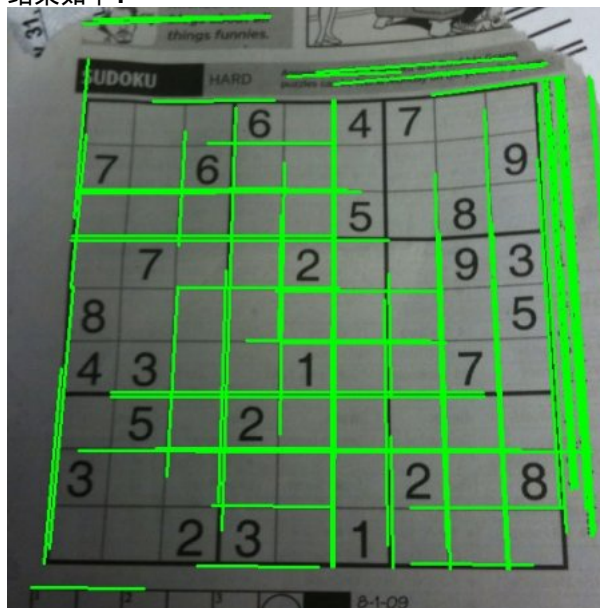
@author: duan
"""

import cv2
import numpy as np

img = cv2.imread('dave.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray,50,150,apertureSize = 3)
minLineLength = 100
maxLineGap = 10
lines = cv2.HoughLinesP(edges,1,np.pi/180,100,minLineLength,maxLineGap)
for x1,y1,x2,y2 in lines[0]:
    cv2.line(img,(x1,y1),(x2,y2),(0,255,0),2)

cv2.imwrite('houghlines5.jpg',img)
```

结果如下：



## 更多资源

1. [http://en.wikipedia.org/wiki/Hough\\_transform](http://en.wikipedia.org/wiki/Hough_transform)

## 练习

## 26 Hough 圆环变换

### 目标

- 学习使用霍夫变换在图像中找圆形（环）。
- 学习函数：**cv2.HoughCircles()**。

### 原理

圆形的数学表达式为  $(x - x_{center})^2 + (y - y_{center})^2 = r^2$ ，其中  $(x_{center}, y_{center})$  为圆心的坐标， $r$  为圆的直径。从这个等式中我们可以看出：一个圆环需要 3 个参数来确定。所以进行圆环霍夫变换的累加器必须是 3 维的，这样的话效率就会很低。所以 OpenCV 用一个比较巧妙的办法，霍夫梯度法，它可以使用边界的梯度信息。

我们要使用的函数为 **cv2.HoughCircles()**。文档中对它的参数有详细的解释。这里我们就直接看代码吧。



```

# -*- coding: utf-8 -*-
"""
Created on Sat Jan 18 19:53:50 2014

@author: duan
"""

import cv2
import numpy as np

img = cv2.imread('opencv_logo.png',0)
img = cv2.medianBlur(img,5)
cimg = cv2.cvtColor(img,cv2.COLOR_GRAY2BGR)

circles = cv2.HoughCircles(img,cv2.HOUGH_GRADIENT,1,20,
                           param1=50,param2=30,minRadius=0,maxRadius=0)

circles = np.uint16(np.around(circles))
for i in circles[0,:]:
    # draw the outer circle
    cv2.circle(cimg,(i[0],i[1]),i[2],(0,255,0),2)
    # draw the center of the circle
    cv2.circle(cimg,(i[0],i[1]),2,(0,0,255),3)

cv2.imshow('detected circles',cimg)
cv2.waitKey(0)
cv2.destroyAllWindows()
#Python: cv2.HoughCircles(image, method, dp, minDist, circles, param1, param2, minRadius, maxRadius)
#Parameters:
#image – 8-bit, single-channel, grayscale input image.
# 返回结果为 Output vector of found circles. Each vector is encoded as a
#3-element floating-point vector (x, y, radius) .
#circle_storage – In C function this is a memory storage that will contain
#the output sequence of found circles.
#method – Detection method to use. Currently, the only implemented method is
#CV_HOUGH_GRADIENT , which is basically 21HT , described in [Yuen90].
#dp – Inverse ratio of the accumulator resolution to the image resolution.
#For example, if dp=1 , the accumulator has the same resolution as the input image.
#If dp=2 , the accumulator has half as big width and height.
#minDist – Minimum distance between the centers of the detected circles.
#If the parameter is too small, multiple neighbor circles may be falsely
#detected in addition to a true one. If it is too large, some circles may be missed.
#param1 – First method-specific parameter. In case of CV_HOUGH_GRADIENT ,
#it is the higher threshold of the two passed to the Canny() edge detector
# (the lower one is twice smaller).
#param2 – Second method-specific parameter. In case of CV_HOUGH_GRADIENT ,
# it is the accumulator threshold for the circle centers at the detection stage.
#The smaller it is, the more false circles may be detected. Circles,
# corresponding to the larger accumulator values, will be returned first.
#minRadius – Minimum circle radius.
#maxRadius – Maximum circle radius.

```

结果如下：



更多资源

练习

## 27 分水岭算法图像分割

### 目标

本节我们将要学习

- 使用分水岭算法基于掩模的图像分割
- 函数：`cv2.watershed()`

### 原理

任何一副灰度图像都可以被看成拓扑平面，灰度值高的区域可以被看成是山峰，灰度值低的区域可以被看成是山谷。我们向每一个山谷中灌不同颜色的水。随着水的位的升高，不同山谷的水就会相遇汇合，为了防止不同山谷的水汇合，我们需要在水汇合的地方构建起堤坝。不停的灌水，不停的构建堤坝知道所有的山峰都被水淹没。我们构建好的堤坝就是对图像的分割。这就是分水岭算法的背后哲理。你可以通过访问网站[CMM webpage on watershed](#)来加深自己的理解。

但是这种方法通常都会得到过度分割的结果，这是由噪声或者图像中其他不规则的因素造成的。为了减少这种影响，OpenCV 采用了基于掩模的分水岭算法，在这种算法中我们要设置那些山谷点会汇合，那些不会。这是一种交互式的图像分割。我们要做的就是给我们已知的对象打上不同的标签。如果某个区域肯定是前景或对象，就使用某个颜色（或灰度值）标签标记它。如果某个区域肯定不是对象而是背景就使用另外一个颜色标签标记。而剩下的不能确定是前景还是背景的区域就用 0 标记。这就是我们的标签。然后实施分水岭算法。每一次灌水，我们的标签就会被更新，当两个不同颜色的标签相遇时就构建堤坝，直到将所有山峰淹没，最后我们得到的边界对象（堤坝）的值为 -1。

### 27.1 代码

下面的例子中我们将就和距离变换和分水岭算法对紧挨在一起的对象进行分割。

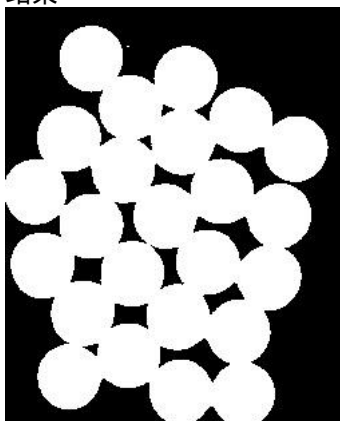
如下图所示，这些硬币紧挨在一起。就算你使用阈值操作，它们任然是紧挨着的。



我们从找到硬币的近似估计开始。我们可以使用 Otsu's 二值化。

```
# -*- coding: utf-8 -*-  
"""  
Created on Sun Jan 19 12:10:49 2014  
  
@author: duan  
"""  
  
import numpy as np  
import cv2  
from matplotlib import pyplot as plt  
  
img = cv2.imread('water_coins.jpg')  
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)  
ret, thresh = cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
```

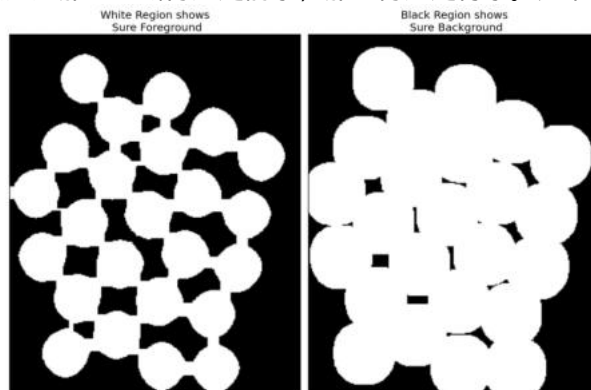
结果



现在我们要去除图像中的所有的白噪声。这就需要使用形态学中的开运算。为了去除对象上小的空洞我们需要使用形态学闭运算。所以现在我们知道靠近对象中心的区域肯定是前景，而远离对象中心的区域肯定是背景。而不能确定

的区域就是硬币之间的边界。

所以我们要提取肯定是硬币的区域。腐蚀操作可以去除边缘像素。剩下就可以肯定是硬币了。当硬币之间没有接触时，这种操作是有效的。但是由于硬币之间是相互接触的，我们就有了另外一个更好的选择：距离变换再加上合适的阈值。接下来我们要找到肯定不是硬币的区域。这是就需要进行膨胀操作了。膨胀可以将对象的边界延伸到背景中去。这样由于边界区域被去处理，我们就可以知道那些区域肯定是前景，那些肯定是背景。如下图所示。



剩下的区域就是我们不知道该如何区分的了。这就是分水岭算法要做的。这些区域通常是前景与背景的交界处（或者两个前景的交界）。我们称之为边界。从肯定是不是背景的区域中减去肯定是前景的区域就得到了边界区域。

```
# noise removal
kernel = np.ones((3,3),np.uint8)
opening = cv2.morphologyEx(thresh,cv2.MORPH_OPEN,kernel, iterations = 2)

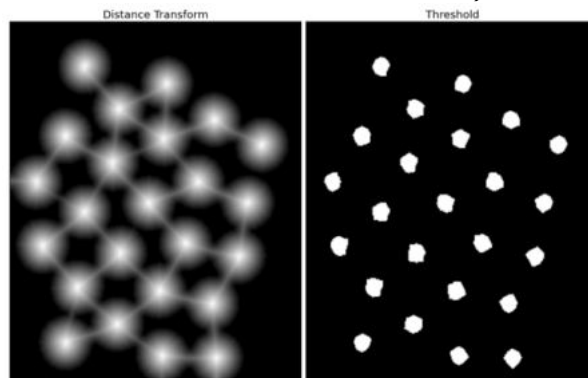
# sure background area
sure_bg = cv2.dilate(opening,kernel,iterations=3)

# Finding sure foreground area
# 距离变换的基本含义是计算一个图像中非零像素点到最近的零像素点的距离，也就是到零像素点的最短距离
# 个最常见的距离变换算法就是通过连续的腐蚀操作来实现，腐蚀操作的停止条件是所有前景像素都被完全
# 腐蚀。这样根据腐蚀的先后顺序，我们就得到各个前景像素点到前景中心骨架像素点的
# 距离。根据各个像素点的距离值，设置为不同的灰度值。这样就完成了二值图像的距离变换
#cv2.distanceTransform(src, distanceType, maskSize)
# 第二个参数 0,1,2 分别表示 CV_DIST_L1, CV_DIST_L2 , CV_DIST_C
dist_transform = cv2.distanceTransform(opening,1,5)
ret, sure_fg = cv2.threshold(dist_transform,0.7*dist_transform.max(),255,0)

# Finding unknown region
sure_fg = np.uint8(sure_fg)
unknown = cv2.subtract(sure_bg,sure_fg)
```

如结果所示，在阈值化之后的图像中，我们得到了肯定是硬币的区域，而且硬币之间也被分割开了。（有些情况下你可能只需要对前景进行分割，而不需要将紧挨在一起的对象分开，此时就没有必要使用距离变换了，腐蚀就足够了。

当然腐蚀也可以用来提取肯定是前景的区域。)



现在知道了那些是背景那些是硬币了。那我们就可以创建标签 (一个与原图像大小相同, 数据类型为 `in32` 的数组), 并标记其中的区域了。对我们已经确定分类的区域 (无论是前景还是背景) 使用不同的正整数标记, 对我们不确定的区域使用 0 标记。我们可以使用函数 `cv2.connectedComponents()` 来做这件事。它会把将背景标记为 0, 其他的对象使用从 1 开始的正整数标记。

但是, 我们知道如果背景标记为 0, 那分水岭算法就会把它当成未知区域了。所以我们想使用不同的整数标记它们。而对不确定的区域 (函数 `cv2.connectedComponents` 输出的结果中使用 `unknown` 定义未知区域) 标记为 0。

```
# Marker labelling
ret, markers1 = cv2.connectedComponents(sure_fg)

# Add one to all labels so that sure background is not 0, but 1
markers = markers1+1

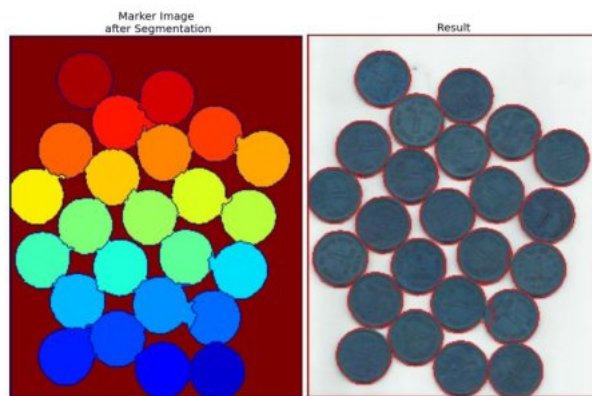
# Now, mark the region of unknown with zero
markers[unknown==255] = 0
```

结果使用 JET 颜色地图表示。深蓝色区域为未知区域。肯定是硬币的区域使用不同的颜色标记。其余区域就是用浅蓝色标记的背景了。

现在标签准备好了。到最后一步: 实施分水岭算法了。标签图像将会被修改, 边界区域的标记将变为 -1。

```
markers3 = cv2.watershed(img, markers)
img[markers3 == -1] = [255, 0, 0]
```

结果如下。有些硬币的边界被分割的很好, 也有一些硬币之间的边界分割的不好。



## 更多资源

1. [CMM webpage on watershed](#)

## 练习

1. OpenCV 自带的示例中有一个交互式分水岭分割程序：**watershed.py**。自己玩玩吧。

## 28 使用 GrabCut 算法进行交互式前景提取

### 目标

在本节中我们将要学习：

- GrabCut 算法原理，使用 GrabCut 算法提取图像的前景
- 创建一个交互是程序完成前景提取

### 原理

GrabCut 算法是由微软剑桥研究院的 Carsten\_Rother, Vladimir\_Kolmogorov 和 Andrew\_Blake 在文章《[GrabCut: interactive foreground extraction using iterated graph cuts](#)》中共同提出的。此算法在提取前景的操作过程中需要很少的人机交互，结果非常好。

从用户的角度来看它到底是如何工作的呢？开始时用户需要用有一个矩形将前景区域框住（前景区域应该完全被包括在矩形框内部）。然后算法进行迭代式分割直达达到最好结果。但是有时分割的结果不够理想，比如把前景当成了背景，或者把背景当成了前景。在这种情况下，就需要用户来进行修改了。用户只需要在不理想的部位画一笔（点一下鼠标）就可以了。画一笔就等于在告诉计算机：“嗨，老兄，你把这里弄反了，下次迭代的时候记得改过来呀！”。然后，在下一轮迭代的时候你就会得到一个更好的结果了。

如下图所示。运动员和足球被蓝色矩形包围在一起。其中有我做的几个修正，白色画笔表明这里是前景，黑色画笔表明这里是背景。最后我得到了一个很好的结果。



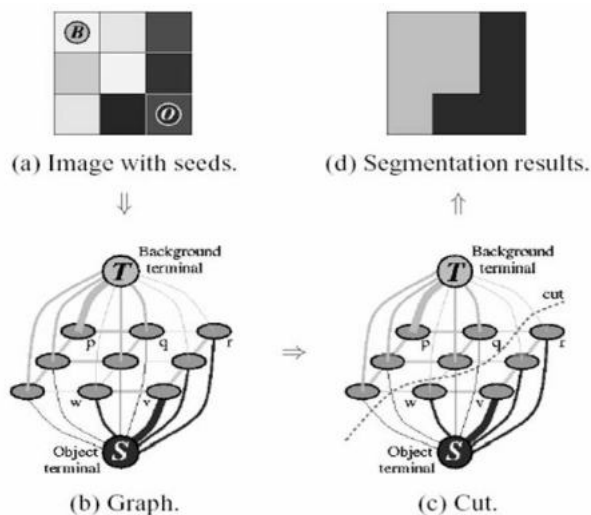
在整个过程中到底发生了什么呢？

- 用户输入一个矩形。矩形外的所有区域肯定都是背景（我们在前面已经提到，所有的对象都要包含在矩形框内）。矩形框内的东西是未知的。同样用户确定前景和背景的任何操作都不会被程序改变。
- 计算机会对我们的输入图像做一个初始化标记。它会标记前景和背景像素。



- 使用一个高斯混合模型 (GMM) 对前景和背景建模。
- 根据我们的输入, GMM 会学习并创建新的像素分布。对那些分类未知的像素 (可能是前景也可能是背景), 可以根据它们与已知分类 (如背景) 的像素的关系来进行分类 (就像是在做聚类操作)。
- 这样就会根据像素的分布创建一副图。图中的节点就是像素点。除了像素点做节点之外还有两个节点: Source\_node 和 Sink\_node。所有的前景像素都和 Source\_node 相连。所有的背景像素都和 Sink\_node 相连。
- 将像素连接到 Source\_node/sink\_node 的 (边) 的权重由它们属于同一类 (同是前景或同是背景) 的概率来决定。两个像素之间的权重由边的信息或者两个像素的相似性来决定。如果两个像素的颜色有很大的不同, 那么它们之间的边的权重就会很小。
- 使用 mincut 算法对上面得到的图进行分割。它会根据最低成本方程将图分为 Source\_node 和 Sink\_node。成本方程就是被剪掉的所有边的权重之和。在裁剪之后, 所有连接到 Source\_node 的像素被认为是前景, 所有连接到 Sink\_node 的像素被认为是背景。
- 继续这个过程直到分类收敛。

下图演示了这个过程 (Image Courtesy: <http://www.cs.ru.ac.za/research/g02m1682/>):



## 28.1 演示

现在我们进入 OpenCV 中的 grabcut 算法。OpenCV 提供了函数: `cv2.grabCut()`。我们先看看它的参数:

- **img** - 输入图像
- **mask**-掩模图像，用来确定那些区域是背景，前景，可能是前景/背景等。  
可以设置为: `cv2.GC_BGD,cv2.GC_FGD,cv2.GC_PR_BGD,cv2.GC_PR_FGD`,  
或者直接输入 0,1,2,3 也行。
- **rect** - 包含前景的矩形，格式为 (x,y,w,h)
- **bdgModel, fgdModel** - 算法内部使用的数组。你只需要创建两个大小为 (1,65)，数据类型为 `np.float64` 的数组。
- **iterCount** - 算法的迭代次数
- **mode** 可以设置为 `cv2.GC_INIT_WITH_RECT` 或 `cv2.GC_INIT_WITH_MASK`,  
也可以联合使用。这是用来确定我们进行修改的方式，矩形模式或者掩模模式。

首先，我们来看使用矩形模式。加载图片，创建掩模图像，构建 `bdgModel` 和 `fgdModel`。传入矩形参数。都是这么直接。让算法迭代 5 次。由于我们在使用矩形模式所以修改模式设置为 `cv2.GC_INIT_WITH_RECT`。运行 `grabcut`。算法会修改掩模图像，在新的掩模图像中，所有的像素被分为四类：背景，前景，可能是背景/前景使用 4 个不同的标签标记（前面参数中提到过）。然后我们来修改掩模图像，所有的 0 像素和 1 像素都被归为 0（例如背景），所有的 1 像素和 3 像素都被归为 1（前景）。我们最终的掩模图像就这样准备好了。用它和输入图像相乘就得到了分割好的图像。

```

# -*- coding: utf-8 -*-
"""
Created on Sun Jan 19 16:24:24 2014

@author: duan
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('messi5.jpg')
mask = np.zeros(img.shape[:2],np.uint8)

bgdModel = np.zeros((1,65),np.float64)
fgdModel = np.zeros((1,65),np.float64)

rect = (50,50,450,290)
# 函数的返回值是更新的 mask, bgdModel, fgdModel
cv2.grabCut(img,mask,rect,bgdModel,fgdModel,5,cv2.GC_INIT_WITH_RECT)

mask2 = np.where((mask==2)|(mask==0),0,1).astype('uint8')
img = img*mask2[:, :, np.newaxis]

plt.imshow(img),plt.colorbar(),plt.show()

```

结果如下：



哎呀，梅西的头发被我们弄没了！让我们来帮他找回头发。所以我们要在那里画一笔（设置像素为 1，肯定是前景）。同时还有一些我们并不需要的草地。我们需要去除它们，我们再在这个区域画一笔（设置像素为 0，肯定是背景）。现在可以象前面提到的那样来修改掩模图像了。

实际上我是怎么做的呢？我们使用图像编辑软件打开输入图像，添加一个图层，使用笔刷工具在需要的地方使用白色绘制（比如头发，鞋子，球等）；使用黑色笔刷在不需要的地方绘制（比如，logo，草地等）。然后将其他地方用灰色填充，保存成新的掩模图像。在 OpenCV 中导入这个掩模图像，根据新的掩模图像对原来的掩模图像进行编辑。代码如下：

```
# newmask is the mask image I manually labelled
newmask = cv2.imread('newmask.png',0)

# wherever it is marked white (sure foreground), change mask=1
# wherever it is marked black (sure background), change mask=0
mask[newmask == 0] = 0
mask[newmask == 255] = 1

mask, bgdModel, fgdModel = cv2.grabCut(img,mask, None, bgdModel, fgdModel, 5, cv2.GC_INIT_WITH_MASK)

mask = np.where((mask==2)|(mask==0),0,1).astype('uint8')
img = img*mask[:, :, np.newaxis]
plt.imshow(img), plt.colorbar(), plt.show()
```

结果如下：



就是这样。你也可以不使用矩形初始化，直接进入掩码图像模式。使用 2 像素和 3 像素（可能是背景/前景）对矩形区域的像素进行标记。然后象我们在第二个例子中那样对肯定是前景的像素标记为 1 像素。然后直接在掩模图像模式下使用 grabCut 函数。

## 更多资源

### 练习

1. OpenCV 自带的示例中有一个使用 grabcut 算法的交互式工具 grabcut.py，自己玩一下吧。
2. 你可以创建一个交互是取样程序，可以绘制矩形，带有滑动条（调节笔刷的粗细）等。

## 部分 V

# 图像特征提取与描述

## 29 理解图像特征

### 目标

本节我会试着帮你理解什么是图像特征，为什么图像特征很重要，为什么角点很重要等。

### 29.1 解释

我相信你们大多数人都玩过拼图游戏吧。首先你们拿到一张图片的一堆碎片，要做的就是把这些碎片以正确的方式排列起来从而重建这幅图像。问题是，你怎样做到的呢？如果把你做游戏的原理写成计算机程序，那计算机就也会玩拼图游戏了。如果计算机可以玩拼图，我们就可以给计算机一大堆自然图片，然后就可以让计算机把它拼成一张大图了。如果计算机可以自动拼接自然图片，那我们是不是可以给计算机关于一个建筑的的大量图片，然后让计算机给我们创建一个 3D 的模型呢？

问题和联想可以无边无际。但是所有的这些问题都是建立在一个基础问题之上的。这个问题就是：我们是如何玩拼图的？我们是如何把一堆碎片拼在一起的？我们有时如何把一个个自然场景拼接成一个单独图像的？

答案就是：我们要寻找一些唯一的特征，这些特征要适于被跟踪，容易被比较。如果我们要定义这样一种特征，虽然我们不知道它是什么但很难用语言来描述。如果让你找出一个可以在不同图片之间相互比较的好的特征，你肯定能搞定。这就是为什么小孩子也会玩拼图的原因。我们在一副图像中搜索这样的特征，我们能找到它们，而且也能在其他图像中找到这些特征，然后再把它们拼接到一块。（在拼图游戏中，我们更注重的是图片之间的连续性）。我们的这些能力都是天生的。

所以我们的一个问题现在扩展成了几个，但是更加确切了。这些特征是什么呢？（我们的答案必须也能被计算机理解）。

好吧，很难说人是怎样找出这些特征的。这些能力已经刻在我们的大脑中了。但是如果 we 深入的观察一些图像并搜索不同的 pattern，我们会发现一些有趣的事。一下图为例：

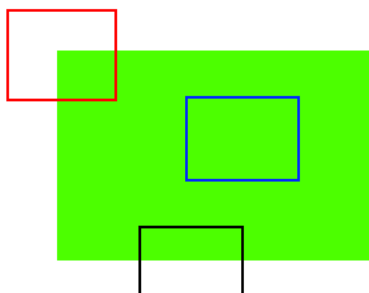


图像很简单。在图像的上方给出了六个小图。你要做的就是找到这些小图在原始图像中的位置。你能找到多少正确结果呢？

A 和 B 是平面，而且它们的图像中很多地方都存在。很难找到这些小图的准确位置。

C 和 D 更简单。它们是建筑的边缘。你可以找到它们的近似位置，但是准确位置还是很难找到。这是因为：沿着边缘，所有的地方都一样。所以边缘是比平面更好的特征，但是还不够好（在拼图游戏中要找连续的边缘）。

最后 E 和 F 是建筑的一些角点。它们能很容易的被找到。因为在角点的地方，无论你向哪个方向移动小图，结果都会有很大的不同。所以可以把它们当成一个好的特征。为了更好的理解这个概念我们举个更简单的例子。



如上图所示，蓝色框中的区域是一个平面很难被找到和跟踪。无论你向那个方向移动蓝色框，长的都一样。对于黑色框中的区域，它是一个边缘。如果你沿垂直方向移动，它会改变。但是如果沿水平方向移动就不会改变。而红色框中的角点，无论你向那个方向移动，得到的结果都不同，这说明它是唯一的。所以，基本上来说角点是一个好的图像特征。（不仅仅是角点，有些情况斑点也是好的图像特征）。

现在我们终于回答了前面的问题了，“这些特征是什么？”。但是下一个问题又来了。我们怎样找到它们？或者说我们怎样找到角点？我们也已经用一种直

观的方式做了回答，比如在图像中找一些区域，无论你想那个方向移动这些区域变化都很大。在下一节中我们会用计算机语言来实现这个想法。所以找到图像特征的技术被称为**特征检测**。

现在我们找到了图像特征（假设你已经搞定）。在找到这些之后，你应该在其他图像中也找到同样的特征。我们应该怎么做呢？我们选择特征周围的一个区域，然后用我们自己的语言来描述它，比如“上边是蓝天，下边是建筑，在建筑上有很多玻璃等”，你就可以在其他图片中搜索相同的区域了。基本上看来，你是在描述特征。同样，计算机也要对特征周围的区域进行描述，这样它才能在其他图像中找到相同的特征。我们把这种描述称为**特征描述**。当你有了特征很它们的描述后，你就可以在所有的图像中找这个相同的特征了，找到之后你就可以做任何你想做的了。

本章我们就是要使用 OpenCV 中的各种算法来查找图像的特征，然后描述它们，对它们进行匹配等。

## 更多资源

### 练习

## 30 Harris 角点检测

### 目标

- 理解 Harris 角点检测的概念
- 学习函数: `cv2.cornerHarris()`, `cv2.cornerSubPix()`

### 原理

在上一节我们已经知道了角点的一个特性: 向任何方向移动变化都很大。Chris\_Harris 和 Mike\_Stephens 早在 1988 年的文章《A Combined Corner and Edge Detector》中就已经提出了焦点检测的方法, 被称为 Harris 角点检测。他把这个简单的想法转换成了数学形式。将窗口向各个方向移动  $(u, v)$  然后计算所有差异的总和。表达式如下:

$$E(u, v) = \sum_{x, y} \underbrace{w(x, y)}_{\text{window function}} \underbrace{[I(x+u, y+v) - I(x, y)]^2}_{\text{shifted intensity} - \text{intensity}}$$

窗口函数可以是正常的矩形窗口也可以是对每一个像素给予不同权重的高斯窗口

角点检测中要使  $E(u, v)$  的值最大。这就是说必须使方程右侧的第二项的取值最大。对上面的等式进行泰勒级数展开然后再通过几步数学换算 (可以参考其他标准教材), 我们得到下面的等式:

$$E(u, v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

其中

$$M = \sum_{x, y} w(x, y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix}$$

这里  $I_x$  和  $I_y$  是图像在  $x$  和  $y$  方向的导数。(可以使用函数 `cv2.Sobel()` 计算得到)。

然后就是主要部分了。他们根据一个用来判定窗口内是否包含角点的等式进行打分。

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

其中

- $\det(M) = \lambda_1 \lambda_2$
- $\text{trace}(M) = \lambda_1 + \lambda_2$
- $\lambda_1$  和  $\lambda_2$  是矩阵  $M$  的特征值

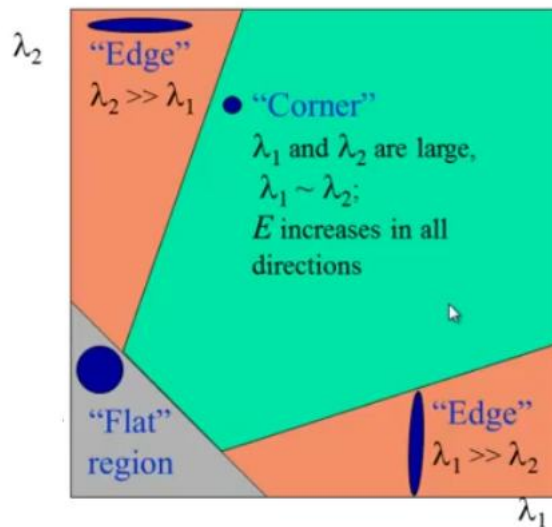
所以根据这些特征中我们可以判断一个区域是否是角点, 边界或者是平面。

- 当  $\lambda_1$  和  $\lambda_2$  都小时,  $|R|$  也小, 这个区域就是一个平坦区域。
- 当  $\lambda_1 \gg \lambda_2$  或者  $\lambda_1 \ll \lambda_2$ , 时  $R$  小于 0, 这个区域是边缘



- 当  $\lambda_1$  和  $\lambda_2$  都很大，并且  $\lambda_1 \sim \lambda_2$  中的时， $R$  也很大，（ $\lambda_1$  和  $\lambda_2$  中的最小值都大于阈值）说明这个区域是角点。

可以用下图来表示我们的结论：



所以 Harris 角点检测的结果是一个由角点分数构成的灰度图像。选取适当的阈值对结果图像进行二值化我们就检测到了图像中的角点。我们将用一个简单的图片来演示一下。

### 30.1 OpenCV 中的 Harris 角点检测

Open 中的函数 `cv2.cornerHarris()` 可以用来进行角点检测。参数如下：

- **img** - 数据类型为 float32 的输入图像。
- **blockSize** - 角点检测中要考虑的领域大小。
- **ksize** - Sobel 求导中使用的窗口大小
- **k** - Harris 角点检测方程中的自由参数，取值参数为 [0.04, 0.06].

例子如下：

```

# -*- coding: utf-8 -*-
"""
Created on Mon Jan 20 18:53:24 2014

@author: duan
"""

import cv2
import numpy as np

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

gray = np.float32(gray)

# 输入图像必须是 float32, 最后一个参数在 0.04 到 0.05 之间
dst = cv2.cornerHarris(gray,2,3,0.04)

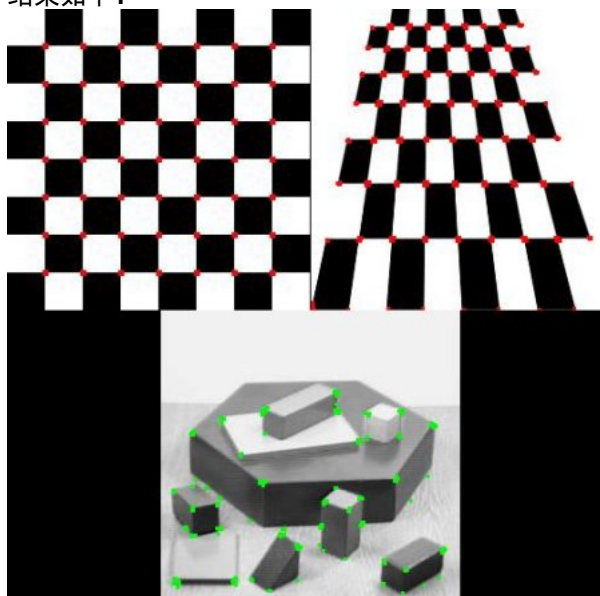
#result is dilated for marking the corners, not important
dst = cv2.dilate(dst,None)

# Threshold for an optimal value, it may vary depending on the image.
img[dst>0.01*dst.max()]=[0,0,255]

cv2.imshow('dst',img)
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()

```

结果如下：



## 30.2 亚像素级精确度的角点

有时我们需要最大精度的角点检测。OpenCV 为我们提供了函数 `cv2.cornerSubPix()`，它可以提供亚像素级别的角点检测。下面是一个例子。首先我们要找到 Harris 角点，然后将角点的重心传给这个函数进行修正。Harris 角点用红色像素标出，绿色像素是修正后的像素。在使用这个函数是我们要定义一个迭代停止条件。当迭代次数达到或者精度条件满足后迭代就会停止。我们同样需要定义进行角点搜索的邻域大小。

```

# -*- coding: utf-8 -*-
"""
Created on Mon Jan 20 18:55:47 2014

@author: duan
"""

import cv2
import numpy as np

filename = 'chessboard2.jpg'
img = cv2.imread(filename)
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

# find Harris corners
gray = np.float32(gray)
dst = cv2.cornerHarris(gray,2,3,0.04)
dst = cv2.dilate(dst,None)
ret, dst = cv2.threshold(dst,0.01*dst.max(),255,0)
dst = np.uint8(dst)

# find centroids
#connectedComponentsWithStats(InputArray image, OutputArray labels, OutputArray stats,
#OutputArray centroids, int connectivity=8, int ltype=CV_32S)
ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)

# define the criteria to stop and refine the corners
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.001)

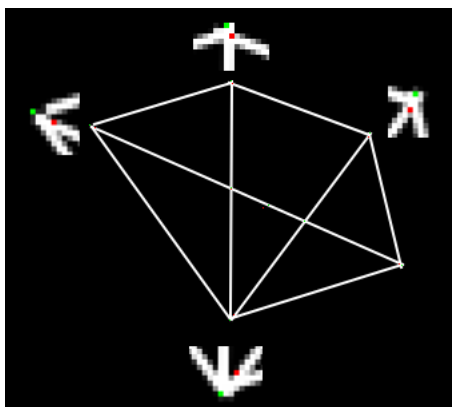
#Python: cv2.cornerSubPix(image, corners, winSize, zeroZone, criteria)
#zeroZone – Half of the size of the dead region in the middle of the search zone
#over which the summation in the formula below is not done. It is used sometimes
# to avoid possible singularities of the autocorrelation matrix. The value of (-1,-1)
# indicates that there is no such a size.
# 返回值由角点坐标组成的一个数组（而非图像）
corners = cv2.cornerSubPix(gray,np.float32(centroids),(5,5),(-1,-1),criteria)

# Now draw them
res = np.hstack((centroids,corners))
#np.int0 可以用来省略小数点后面的数字（非四舍五入）。
res = np.int0(res)
img[res[:,1],res[:,0]]=[0,0,255]
img[res[:,3],res[:,2]] = [0,255,0]

cv2.imwrite('subpixel5.png',img)

```

结果如下，为了方便查看我们对角点的部分进行了放大：



更多资源

练习

## 31 Shi-Tomasi 角点检测 & 适合于跟踪的图像特征

### 目标

本节我们将要学习：

- 另外一个角点检测技术：Shi-Tomasi 焦点检测
- 函数：`cv2.goodFeatureToTrack()`

### 原理

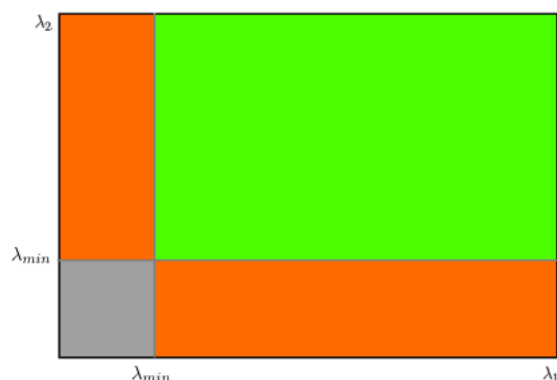
上一节我们学习了 Harris 角点检测，后来 1994 年，J.Shi 和 C.Tomasi 在他们的文章《Good\_Features\_to\_Track》中对这个算法做了一个小小的修改，并得到了更好的结果。我们知道 Harris 角点检测的打分公式为：

$$R = \lambda_1 \lambda_2 - k (\lambda_1 + \lambda_2)^2$$

但 Shi-Tomasi 使用的打分函数为：

$$R = \min(\lambda_1, \lambda_2)$$

如果打分超过阈值，我们就认为它是一个角点。我们可以把它绘制到  $\lambda_1 \sim \lambda_2$  空间中，就会得到下图：



从这幅图中，我们可以看出来只有当  $\lambda_1$  和  $\lambda_2$  都大于最小值时，才被认为是角点（绿色区域）。

### 31.1 代码

OpenCV 提供了函数：`cv2.goodFeaturesToTrack()`。这个函数可以帮助我们使用 Shi-Tomasi 方法获取图像中 N 个最好的角点（如果你愿意的话，

也可以通过改变参数来使用 Harris 角点检测算法)。通常情况下,输入的应该是灰度图像。然后确定你想要检测到的角点数目。再设置角点的质量水平,0 到 1 之间。它代表了角点的最低质量,低于这个数的所有角点都会被忽略。最后在设置两个角点之间的最短欧式距离。

根据这些信息,函数就能在图像上找到角点。所有低于质量水平的角点都会被忽略。然后再把合格角点按角点质量进行降序排列。函数会采用角点质量最高的那个角点(排序后的第一个),然后将它附近(最小距离之内)的角点都删掉。按着这样的方式最后返回 N 个最佳角点。

在下面的例子中,我们试着找出 25 个最佳角点:

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jan 20 20:40:25 2014

@author: duan
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt

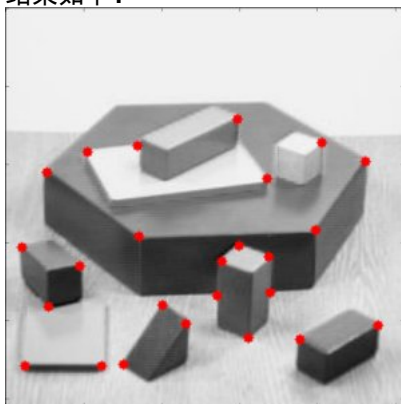
img = cv2.imread('simple.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

corners = cv2.goodFeaturesToTrack(gray,25,0.01,10)
# 返回的结果是 [[ 311., 250.]] 两层括号的数组。
corners = np.int0(corners)

for i in corners:
    x,y = i.ravel()
    cv2.circle(img,(x,y),3,255,-1)

plt.imshow(img),plt.show()
```

结果如下:



我们以后会发现这个函数很适合在目标跟踪中使用。

**更多资源**

**练习**



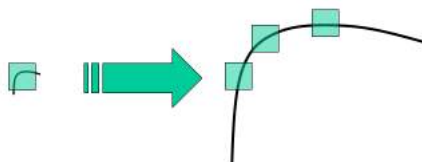
## 32 介绍 SIFT(Scale-Invariant Feature Transform)

### 目标

- 学习 SIFT 算法的概念
- 学习在图像中查找 SIFT 关键点和描述符

### 原理

在前面两节我们学习了一些角点检测技术，比如 Harris 等。它们具有旋转不变特性，即使图片发生了旋转，我们也能找到同样的角点。很明显即使图像发生旋转之后角点还是角点。那如果我们对图像进行缩放呢？角点可能就不再是角点了。以下图为例，在一副小图中使用一个小的窗口可以检测到一个角点，但是如果图像被放大，再使用同样的窗口就检测不到角点了。



所以在 2004 年，D.Lowe 提出了一个新的算法：尺度不变特征变换 (SIFT)，这个算法可以帮助我们提取图像中的关键点并计算它们的描述符。

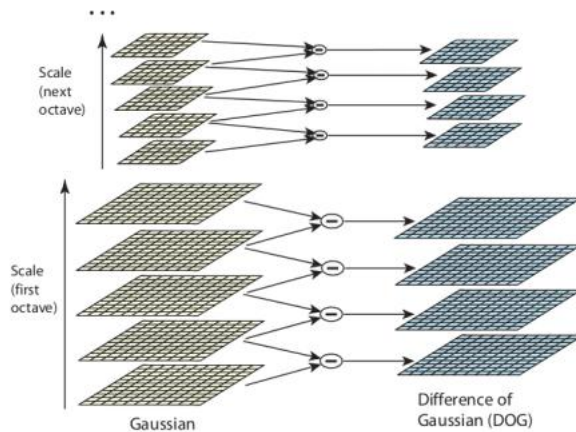
SIFT 算法主要由四步构成。我们来逐步进行学习。

### 尺度空间极值检测

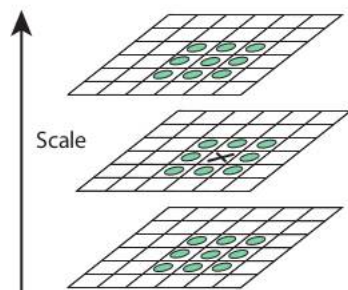
从上图我们可以很明显的看出来在不同的尺度空间不能使用相同的窗口检测极值点。对小的角点要用小的窗口，对大的角点只能使用大的窗口。为了达到这个目的我们要使用尺度空间滤波器。(尺度空间滤波器可以使用一些列具有不同方差  $\sigma$  的高斯卷积核构成)。使用具有不同方差值  $\sigma$  的高斯拉普拉斯算子 (LoG) 对图像进行卷积，LoG 由于具有不同的方差值  $\sigma$  所以可以用来检测不同大小的斑点 (当 LoG 的方差  $\sigma$  与斑点直径相等时能够使斑点完全平滑)。简单来说方差  $\sigma$  就是一个尺度变换因子。例如，上图中使用一个小方差  $\sigma$  的高斯卷积核是可以很好的检测出小的角点，而使用大方差  $\sigma$  的高斯卷积核时可以很好的检测除大的角点。所以我们可以 在尺度空间和二维平面中检测到局部最大值，如  $(x, y, \sigma)$ ，这表示在  $\sigma$  尺度中  $(x, y)$  点可能是一个关键点。(高斯方差的大小与窗口的大小存在一个倍数关系：窗口大小等于 6 倍方差加 1，所以方差的大小也决定了窗口大小)

但是这个 LoG 的计算量非常大，所以 SIFT 算法使用高斯差分算子 (DoG) 来对 LoG 做近似。这里需要再解释一下图像金字塔，我们可以通过减

少采样（如只取奇数行或奇数列）来构成一组图像尺寸（1，0.5，0.25 等）不同的金字塔，然后对这一组图像中的每一张图像使用具有不同方差  $\sigma$  的高斯卷积核构建出具有不同分辨率的图像金字塔（不同的尺度空间）。DoG 就是这组具有不同分辨率的图像金字塔中相邻的两层之间的差值。如下图所示：



在 DoG 搞定之后，就可以在不同的尺度空间和 2D 平面中搜索局部最大值了。对于图像中的一个像素点而言，它需要与自己周围的 8 邻域，以及尺度空间中上下两层中的相邻的 18（ $2 \times 9$ ）个点相比。如果是局部最大值，它就可能是一个关键点。基本上来说关键点是图像在相应尺度空间中的最好代表。如下图所示：



该算法的作者在文章中给出了 SIFT 参数的经验值：**octaves=4**（通过降低采样从而减小图像尺寸，构成尺寸减小的图像金字塔（4 层）？），尺度空间为 5，也就是每个尺寸使用 5 个不同方差的高斯核进行卷积，初始方差是 1.6， $k$  等于  $\sqrt{2}$  等。

### 关键点（极值点）定位

一旦找到关键点，我们就要对它们进行修正从而得到更准确的结果。作者使用尺度空间的泰勒级数展开来获得极值的准确位置，如果极值点的灰度值小于阈值（0.03）就会被忽略掉。在 OpenCV 中这种阈值被称为 **contrastThreshold**。

DoG 算法对边界非常敏感，所以我们必须要把边界去除。前面我们讲的

Harris 算法除了可以用于角点检测之外还可以用于检测边界。作者就是使用了同样的思路。作者使用  $2 \times 2$  的 Hessian 矩阵计算主曲率。从 Harris 角点检测的算法中，我们知道当一个特征值远远大于另外一个特征值时检测到的是边界。所以他们使用了一个简单的函数，如果比例高于阈值（OpenCV 中称为边界阈值），这个关键点就会被忽略。文章中给出的边界阈值为 10。

所以低对比度的关键点和边界关键点都会被去除掉，剩下的就是我们感兴趣的 keypoints 了。

### 为关键点（极值点）指定方向参数

现在我们要为每一个关键点赋予一个方向参数，这样它才会具有旋转不变性。获取关键点（所在尺度空间）的邻域，然后计算这个区域的梯度级和方向。根据计算得到的结果创建一个含有 36 个 bins（每 10 度一个 bin）的方向直方图。（使用当前尺度空间  $\sigma$  值的 1.5 倍为方差的圆形高斯窗口和梯度级做权重）。直方图中的峰值为主方向参数，如果其他的任何柱子的高度高于峰值的 80% 被认为是辅方向。这就会在相同的尺度空间相同的位置构建除具有不同方向的关键点。这对于匹配的稳定性会有所帮助。

### 关键点描述符

新的关键点描述符被创建了。选取与关键点周围一个  $16 \times 16$  的邻域，把它分成 16 个  $4 \times 4$  的小方块，为每个小方块创建一个具有 8 个 bin 的方向直方图。总共加起来有 128 个 bin。由此组成长为 128 的向量就构成了关键点描述符。除此之外还要进行几个测量以达到对光照变化，旋转等的稳定性。

### 关键点匹配

下一步就可以采用关键点特征向量的欧式距离来作为两幅图像中关键点的相似性判定度量。取第一个图的某个关键点，通过遍历找到第二幅图像中的距离最近的那个关键点。但有些情况下，第二个距离最近的关键点与第一个距离最近的关键点靠的太近。这可能是由于噪声等引起的。此时要计算最近距离与第二近距离的比值。如果比值大于 0.8，就忽略掉。这会去除 90% 的错误匹配，同时只去除 5% 的正确匹配。如文章所说。

这就是 SIFT 算法的摘要。非常推荐你阅读原始文献，这会加深你对算法的理解。请记住这个算法是受专利保护的。所以这个算法包含在 OpenCV 中的收费模块中。

### OpenCV 中的 SIFT

现在让我们来看看 OpenCV 中关于 SIFT 的函数吧。让我们从关键点检测和绘制开始吧。首先我们要创建对象。我们可以使用不同的参数，这并不是

必须的，关于参数的解释可以查看文档。

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan 22 19:22:10 2014

@author: duan
"""

import cv2
import numpy as np

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

sift = cv2.SIFT()
kp = sift.detect(gray,None)

img=cv2.drawKeypoints(gray,kp)

cv2.imwrite('sift_keypoints.jpg',img)
```

函数 `sift.detect()` 可以在图像中找到关键点。如果你只想在图像中的一个区域搜索的话，也可以创建一个掩模图像作为参数使用。返回的关键点是一个带有很多不同属性的特殊结构体，这些属性中包含它的坐标（x, y），有意义的邻域大小，确定其方向的角度等。

OpenCV 也提供了绘制关键点的函数：**`cv2.drawKeyPoints()`**，它可以在关键点的部位绘制一个小圆圈。如果你设置参数为 **`cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_`**就会绘制代表关键点大小的圆圈甚至可以绘制除关键点的方向。

```
img=cv2.drawKeypoints(gray,kp,flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imwrite('sift_keypoints.jpg',img)
```

结果如下：



现在来计算关键点描述符，OpenCV 提供了两种方法。

1. 由于我们已经找到了关键点，我们可以使用函数 **sift.compute()** 来计算这些关键点的描述符。例如：`kp, des = sift.compute(gray, kp)`。
2. 如果还没有找到关键点，可以使用函数 **sift.detectAndCompute()** 一步到位直接找到关键点并计算出其描述符。

这里我们来看看第二个方法：

这里 `kp` 是一个关键点列表。`des` 是一个 Numpy 数组，其大小是关键点数目乘以 128。

所以我们得到了关键点和描述符等。现在我们想看看如何在不同图像之间进行关键点匹配，这就是我们在接下来的章节将要学习的内容。

## 更多资源

### 练习

## 33 介绍 SURF(Speeded-Up Robust Features)

### 目标

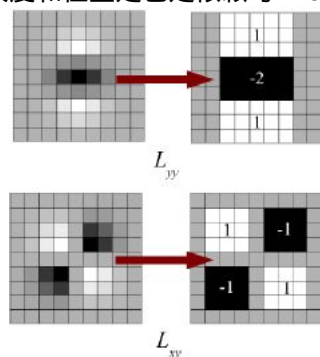
本节我们将要学习：

- SURF 的基础是什么？
- OpenCV 中的 SURF

### 原理

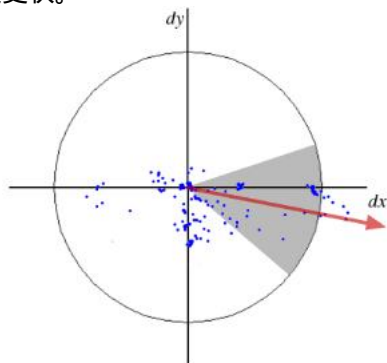
在上一节中我们学习了使用 SIFT 算法进行关键点检测和描述。但是这种算法的执行速度比较慢，人们需要速度更快的算法。在 2006 年 Bay,H., Tuytelaars,T. 和 Van Gool,L 共同提出了 SURF（加速稳健特征）算法。跟它的名字一样，这是个算法是加速版的 SIFT。

在 SIFT 中，Lowe 在构建尺度空间时使用 DoG 对 LoG 进行近似。SURF 使用盒子滤波器（box\_filter）对 LoG 进行近似。下图显示了这种近似。在进行卷积计算时可以利用积分图像（积分图像的一大特点是：计算图像中某个窗口内所有像素和时，计算量的大小与窗口大小无关），是盒子滤波器的一大优点。而且这种计算可以在不同尺度空间同时进行。同样 SURF 算法计算关键点的尺度和位置也是依赖与 Hessian 矩阵行列式的。



为了保证特征矢量具有选装不变性，需要对于每一个特征点分配一个主要方向。需要以特征点为中心，以  $6s$  ( $s$  为特征点的尺度) 为半径的圆形区域内，对图像进行 Harr 小波相应运算。这样做实际就是对图像进行梯度运算，但是利用积分图像，可以提高计算图像梯度的效率，为了求取主方向值，需啊设计一个以方向为中心，张角为 60 度的扇形滑动窗口，以步长为 0.2 弧度左右旋转这个滑动窗口，并对窗口内的图像 Haar 小波的响应值进行累加。主方向为最大的 Haar 响应累加值对应的方向。在很多应用中根本就不需要旋转不变性，所以没有必要确定它们的方向，如果不计算方向的话，又可以使算法提速。SURF 提供了成为 U-SURF 的功能，它具有更快的速度，同时保持了对  $\pm 15$  度旋转的稳定性。OpenCV 对这两种模式同样支持，只需要对参数

upright 进行设置，当 upright 为 0 时计算方向，为 1 时不计算方向，同时速度更快。

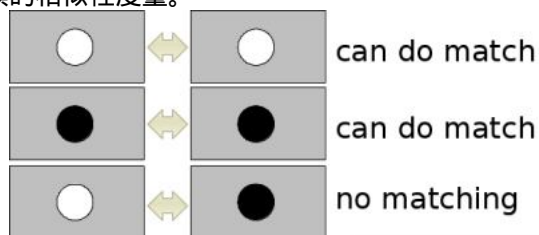


生成特征点的特征矢量需要计算图像的 Haar 小波响应。在一个矩形的区域内，以特征点为中心，沿主方向将  $20s \times 20s$  的图像划分成  $4 \times 4$  个子块，每个子块利用尺寸  $2s$  的 Haar 小波模版进行响应计算，然后对响应值进行统计，组成向量  $v = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$ 。这个描述符的长度为 64。降低的维度可以加速计算和匹配，但又能提供更容易区分的特征。

为了增加特征点的独特性，SURF 还提供了一个加强版 128 维的特征描述符。当  $d_y$  大于 0 和小于 0 时分别对  $d_x$  和  $|d_x|$  的和进行计算，计算  $d_y$  和  $|d_y|$  时也进行区分，这样获得特征就会加倍，但又不会增加计算的复杂度。OpenCV 同样提供了这种功能，当参数 extended 设置为 1 时为 128 维，当参数为 0 时为 64 维，默认情况为 128 维。

在检测特征点的过程中计算了 Hessian 矩阵的行列式，与此同时，计算得到了 Hessian 矩阵的迹，矩阵的迹为对角元素之和。

按照亮度的不同，可以将特征点分为两种，第一种为特征点迹其周围小邻域的亮度比背景区域要亮，Hessian 矩阵的迹为正；另外一种为特征点迹其周围小邻域的亮度比背景区域要暗，Hessian 矩阵为负值。根据这个特性，首先对两个特征点的 Hessian 的迹进行比较。如果同号，说明两个特征点具有相同的对比度；如果异号的话，说明两个特征点的对比度不同，放弃特征点之间的后续的相似性度量。



对于两个特征点描述子的相似性度量，我们采用欧式距离进行计算。

简单来说 SURF 算法采用了很多方法来对每一步进行优化从而提高速度。分析显示在结果效果相当的情况下 SURF 的速度是 SIFT 的 3 倍。SURF 善于处理具有模糊和旋转的图像，但是不善于处理视角变化和光照变化。

### 33.1 OpenCV 中的 SURF

与 SIFT 相同 OpenCV 也提供了 SURF 的相关函数。首先我们要初始化一个 SURF 对象，同时设置好可选参数：64/128 维描述符，Upright/Normal 模式等。所有的细节都已经在文档中解释的很明白了。就像我们在 SIFT 中一样，我们可以使用函数 **SURF.detect()**，**SURF.compute()** 等来进行关键点检测和描述。

首先从查找描述绘制关键点开始。由于和 SIFT 一样所以我们的示例都在 Python 终端中演示。

```
>>> img = cv2.imread('fly.png',0)

# Create SURF object. You can specify params here or later.
# Here I set Hessian Threshold to 400
>>> surf = cv2.SURF(400)

# Find keypoints and descriptors directly
>>> kp, des = surf.detectAndCompute(img,None)

>>> len(kp)
699
```

在一幅图像中显示 699 个关键点太多了。我们把它缩减到 50 个再绘制到图片上。在匹配时，我们可能需要所有的这些特征，不过现在还不需要。所以我们现在提高 Hessian 的阈值。

```
# Check present Hessian threshold
>>> print surf.hessianThreshold
400.0

# We set it to some 50000. Remember, it is just for representing in picture.
# In actual cases, it is better to have a value 300-500
>>> surf.hessianThreshold = 50000

# Again compute keypoints and check its number.
>>> kp, des = surf.detectAndCompute(img,None)

>>> print len(kp)
47
```

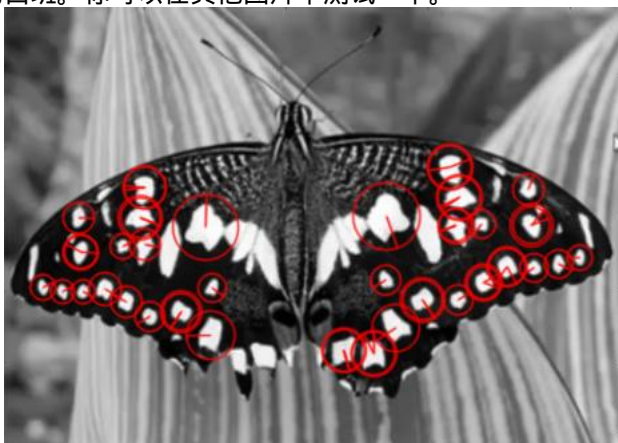
现在低于 50 了，把它们绘制到图像中吧。



```
>>> img2 = cv2.drawKeypoints(img,kp,None,(255,0,0),4)

>>> plt.imshow(img2),plt.show()
```

结果如下。你会发现 SURF 很像一个斑点检测器。它可以检测到蝴蝶翅膀上的白斑。你可以在其他图片中测试一下。



现在我们试一下 U-SURF，它不会检测 keypoints 的方向。

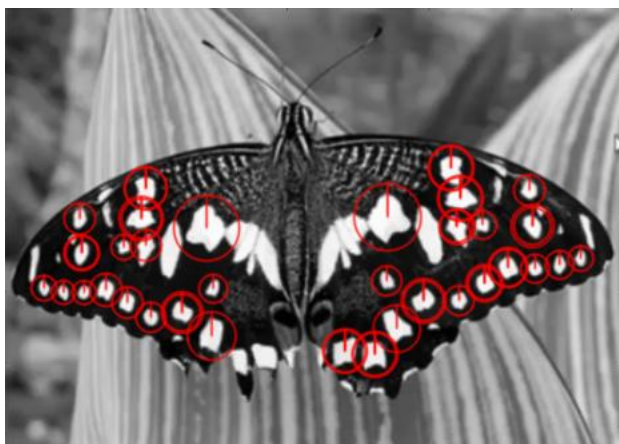
```
# Check upright flag, if it False, set it to True
>>> print surf.upright
False

>>> surf.upright = True

# Recompute the feature points and draw it
>>> kp = surf.detect(img,None)
>>> img2 = cv2.drawKeypoints(img,kp,None,(255,0,0),4)

>>> plt.imshow(img2),plt.show()
```

结果如下。所有的 keypoints 的朝向都是一致的。它比前面的快很多。如果你的工作对 keypoints 的朝向没有特别的要求（如全景图拼接）等，这种方法会更快。



最后我们再看看关键点描述符的大小，如果是 64 维的就改成 128 维。

```
# Find size of descriptor
>>> print surf.descriptorSize()
64

# That means flag, "extended" is False.
>>> surf.extended
False

# So we make it to True to get 128-dim descriptors.
>>> surf.extended = True
>>> kp, des = surf.detectAndCompute(img, None)
>>> print surf.descriptorSize()
128
>>> print des.shape
(47, 128)
```

接下来要做的就是匹配了，我们会在后面讨论。

## 更多资源

## 练习

## 34 角点检测的 FAST 算法

### 目标

- 理解 FAST 算法的基础
- 使用 OpenCV 中的 FAST 算法相关函数进行角点检测

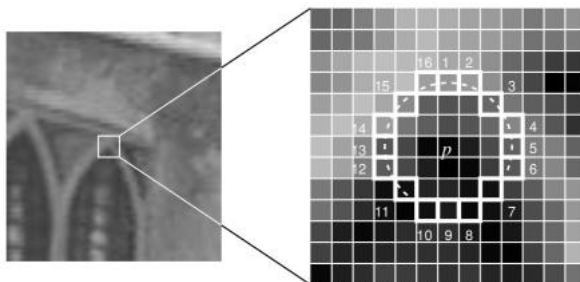
### 原理

我们前面学习了几个特征检测器，它们大多数效果都很好。但是从实时处理的角度来看，这些算法都不够快。一个最好例子就是 SLAM（同步定位与地图构建），移动机器人，它们的计算资源非常有限。

为了解决这个问题，Edward\_Rosten 和 Tom\_Drummond 在 2006 年提出里 FAST 算法。我们下面将会对此算法进行一个简单的介绍。你可以参考原始文献获得更多细节（本节中的所有图像都是曲子原始文章）。

### 34.1 使用 FAST 算法进行特征提取

1. 在图像中选取一个像素点  $p$ ，来判断它是不是关键点。 $I_p$  等于像素点  $p$  的灰度值。
2. 选择适当的阈值  $t$ 。
3. 如下图所示在像素点  $p$  的周围选择 16 个像素点进行测试。



4. 如果在这 16 个像素点中存在  $n$  个连续像素点的灰度值都高于  $I_p + t$ ，或者低于  $I_p - t$ ，那么像素点  $p$  就被认为是一个角点。如上图中的虚线所示， $n$  选取的值为 12。
5. 为了获得更快的效果，还采用了而外的加速办法。首先对候选点的周围每个 90 度的点：1，9，5，13 进行测试（先测试 1 和 19，如果它们符合阈值要求再测试 5 和 13）。如果  $p$  是角点，那么这四个点中至少有 3 个要符合阈值要求。如果不是的话肯定不是角点，就放弃。对通过这步测试的点再继续进行测试（是否有 12 的点符合阈值要求）。这个检测器的效率很高，但是它有如下几条缺点：

- 当  $n < 12$  时它不会丢弃很多候选点 (获得的候选点比较多)。
- 像素的选取不是最优的, 因为它的效果取决与要解决的问题和角点的分布情况。
- 高速测试的结果被抛弃
- 检测到的很多特征点都是连在一起的。

前 3 个问题可以通过机器学习的方法解决, 最后一个问题可以使用非最大值抑制的方法解决。

## 34.2 机器学习的角点检测器

1. 选择一组训练图片 (最好是跟最后应用相关的图片)
2. 使用 FAST 算法找出每幅图像的特征点
3. 对每一个特征点, 将其周围的 16 个像素存储构成一个向量。对所有图像都这样做构建一个特征向量  $P$
4. 每一个特征点的 16 像素点都属于下列三类中的一种。

$$S_{p \rightarrow x} = \begin{cases} d, & I_{p \rightarrow x} \leq I_p - t & (\text{darker}) \\ s, & I_p - t < I_{p \rightarrow x} < I_p + t & (\text{similar}) \\ b, & I_p + t \leq I_{p \rightarrow x} & (\text{brighter}) \end{cases}$$

5. 根据这些像素点的分类, 特征向量  $P$  也被分为 3 个子集:  $P_d, P_s, P_b$
6. 定义一个新的布尔变量  $K_p$ , 如果  $p$  是角点就设置为 True, 如果不是就设置为 False。
7. 使用 ID3 算法 (决策树分类器) Use the ID3 algorithm (decision tree classifier) to query each subset using the variable  $K_p$  for the knowledge about the true class. It selects the  $x$  which yields the most information about whether the candidate pixel is a corner, measured by the entropy of  $K_p$ .
8. This is recursively applied to all the subsets until its entropy is zero.
9. 将构建好的决策树运用于其他图像的快速的检测。

### 34.3 非极大值抑制

使用极大值抑制的方法可以解决检测到的特征点相连的问题

1. 对所有检测到特征点构建一个打分函数  $V$ 。 $V$  就是像素点  $p$  与周围 16 个像素点差值的绝对值之和。
2. 计算临近两个特征点的打分函数  $V$ 。
3. 忽略  $V$  值最低的特征点

### 34.4 总结

FAST 算法比其它角点检测算法都快。

但是在噪声很高时不够稳定，这是由阈值决定的。

### 34.5 OpenCV 中 FAST 特征检测器

很其他特征点检测一样我们可以在 OpenCV 中直接使用 FAST 特征检测器。如果你愿意的话，你还可以设置阈值，是否进行非最大值抑制，要使用的邻域大小  $n$  等。

邻域设置为下列 3 中之一：`cv2.FAST_FEATURE_DETECTOR_TYPE_5_8`, `cv2.FAST_FEATURE_DETECTOR_TYPE_7_7` 和 `cv2.FAST_FEATURE_DETECTOR_TYPE_9_16`。下面是使用 FAST 算法进行特征点检测的简单代码。

```

# -*- coding: utf-8 -*-
"""
Created on Thu Jan 23 12:06:18 2014

@author: duan
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt

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

# Initiate FAST object with default values
fast = cv2.FastFeatureDetector()

# find and draw the keypoints
kp = fast.detect(img,None)
img2 = cv2.drawKeypoints(img, kp, color=(255,0,0))

# Print all default params
print "Threshold: ", fast.getInt('threshold')
print "nonmaxSuppression: ", fast.getBool('nonmaxSuppression')
print "neighborhood: ", fast.getInt('type')
print "Total Keypoints with nonmaxSuppression: ", len(kp)

cv2.imwrite('fast_true.png',img2)

# Disable nonmaxSuppression
fast.setBool('nonmaxSuppression',0)
kp = fast.detect(img,None)

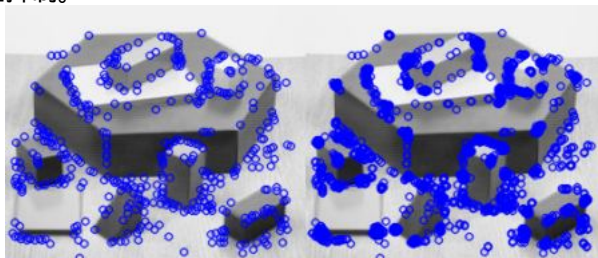
print "Total Keypoints without nonmaxSuppression: ", len(kp)

img3 = cv2.drawKeypoints(img, kp, color=(255,0,0))

cv2.imwrite('fast_false.png',img3)

```

结果如下。第一幅图是使用了非最大值抑制的结果，第二幅没有使用非最大值抑制。



## 更多资源

1. Edward Rosten and Tom Drummond, “Machine learning for high speed corner detection” in 9th European Conference on Computer Vision, vol. 1, 2006, pp. 430–443.
2. Edward Rosten, Reid Porter, and Tom Drummond, “Faster and better: a machine learning approach to corner detection” in IEEE Trans. Pattern Analysis and Machine Intelligence, 2010, vol 32, pp. 105-119.

## 练习

## 35 BRIEF(Binary Robust Independent Elementary Features)

### 目标

本节

- 我们学习 BRIEF 算法的基础

### 原理

我们知道 SIFT 算法使用的是 128 维的描述符。由于它是使用的浮点数，所以要使用 512 个字节。同样 SURF 算法最少使用 256 个字节（64 为维描述符）。创建一个包含上千个特征的向量需要消耗大量的内存，在嵌入式等资源有限的设备上这样是合适的。匹配时还会消耗更多的内存和时间。

但是在实际的匹配过程中如此多的维度是没有必要的。我们可以使用 PCA, LDA 等方法来进行降维。甚至可以使用 LSH（局部敏感哈希）将 SIFT 浮点数的描述符转换成二进制字符串。对这些字符串再使用汉明距离进行匹配。汉明距离的计算只需要进行 XOR 位运算以及位计数，这种计算很适合在现代的 CPU 上进行。但我们还是要先找到描述符才能使用哈希，这不能解决最初的内存消耗问题。

BRIEF 应运而生。它不去计算描述符而是直接找到一个二进制字符串。这种算法使用的是已经平滑后的图像，它会按照一种特定的方式选取一组像素点对  $n_d(x, y)$ ，然后在这些像素点对之间进行灰度值对比。例如，第一个点对的灰度值分别为  $p$  和  $q$ 。如果  $p$  小于  $q$ ，结果就是 1，否则就是 0。就这样对  $n_d$  个点对进行对比得到一个  $n_d$  维的二进制字符串。

$n_d$  可以是 128, 256, 512。OpenCV 对这些都提供了支持，但在默认情况下是 256（OpenC 是使用字节表示它们的，所以这些值分别对应与 16, 32, 64）。当我们获得这些二进制字符串之后就可以使用汉明距离对它们进行匹配了。

非常重要的一点是：BRIEF 是一种特征描述符，它不提供查找特征的方法。所以我们不得不使用其他特征检测器，比如 SIFT 和 SURF 等。原始文献推荐使用 CenSurE 特征检测器，这种算法很快。而且 BRIEF 算法对 CenSurE 关键点的描述效果要比 SURF 关键点的描述更好。

简单来说 BRIEF 是一种对特征点描述符计算和匹配的快速方法。这种算法可以实现很高的识别率，除非出现平面内的大旋转。

### 35.1 OpenCV 中的 BRIEF

下面的代码使用了 CenSurE 特征检测器和 BRIEF 描述符。（在 OpenCV 中 CenSurE 检测器被叫做 STAR 检测器）。



```

# -*- coding: utf-8 -*-
"""
Created on Thu Jan 23 20:28:03 2014

@author: duan
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt

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

# Initiate STAR detector
star = cv2.FeatureDetector_create("STAR")

# Initiate BRIEF extractor
brief = cv2.DescriptorExtractor_create("BRIEF")

# find the keypoints with STAR
kp = star.detect(img,None)

# compute the descriptors with BRIEF
kp, des = brief.compute(img, kp)

print brief.getInt('bytes')
print des.shape

```

函数 `brief.getInt('bytes')` 会以字节格式给出  $n_d$  的大小，默认值为 32。下面就是匹配了，我们会在其他章节中介绍。

## 更多资源

1. Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua, "BRIEF: Binary Robust Independent Elementary Features", 11th European Conference on Computer Vision (ECCV), Heraklion, Crete. LNCS Springer, September 2010.
2. LSH (Locality Sensitive Hasing) at wikipedia.

## 练习

## 36 ORB (Oriented FAST and Rotated BRIEF)

### 目标

- 我们要学习 ORB 算法的基础

### 原理

对于一个 OpenCV 的狂热爱好者来说 ORB 最重要的一点就是：它来自 "OpenCV\_Labs"。这个算法是在 2011 年提出的。在计算开支，匹配效率以及更主要的是专利问题方面 ORB 算法是 SIFT 和 SURF 算法的一个很好的替代品。SIFT 和 SURF 算法是有专利保护的，如果你要使用它们，就可能要花钱。但是 ORB 不需要 !!!

ORB 基本是 FAST 关键点检测和 BRIEF 关键点描述器的结合体，并通过很多修改增强了性能。首先它使用 FAST 找到关键点，然后再使用 Harris 角点检测对这些关键点进行排序找到其中的前 N 个点。它也使用金字塔从而产生尺度不变性特征。但是有一个问题，FAST 算法步计算方向。那旋转不变性怎样解决呢？作者进行了如下修改。

它使用灰度矩的算法计算出角点的方向。以角点到角点所在（小块）区域质心的方向为向量的方向。为了进一步提高旋转不变性，要计算以角点为中心半径为  $r$  的圆形区域的矩，再根据矩计算除方向。

对于描述符，ORB 使用的是 BRIEF 描述符。但是我们已经知道 BRIEF 对与旋转是不稳定的。所以我们在生成特征前，要把关键点领域的这个 patch 的坐标轴旋转到关键点的方向。For any feature set of  $n$  binary tests at location  $(x_i, y_i)$ , define a  $2 \times n$  matrix,  $S$  which contains the coordinates of these pixels. Then using the orientation of patch,  $\theta$ , its rotation matrix is found and rotates the  $S$  to get steered(rotated) version  $S_\theta$ .

ORB discretize the angle to increments of  $2\pi/30$  (12 degrees), and construct a lookup table of precomputed BRIEF patterns. As long as the keypoint orientation  $\theta$  is consistent across views, the correct set of points  $S_\theta$  will be used to compute its descriptor.

BRIEF has an important property that each bit feature has a large variance and a mean near 0.5. But once it is oriented along key-point direction, it loses this property and become more distributed. High variance makes a feature more discriminative, since it responds differentially to inputs. Another desirable property is to have the tests uncorrelated, since then each test will contribute to the result. To resolve all these, ORB runs a greedy search among all possible binary tests to find the ones that have both high variance and means close to 0.5, as well as being uncorrelated. The result is called

rBRIEF.

For descriptor matching, multi-probe LSH which improves on the traditional LSH, is used. The paper says ORB is much faster than SURF and SIFT and ORB descriptor works better than SURF. ORB is a good choice in low-power devices for panorama stitching etc.

实验证明, BRIEF 算法的每一位的均值接近 0.5, 并且方差很大。steered\_BRIEF 算法的每一位的均值比较分散 (均值为 0.5, 0.45, 0.35... 等值的关键点数相当), 这导致方差减小。数据的方差大的一个好处是: 使得特征更容易分辨。为了对 steered\_BRIEF 算法使得特征的方差减小的弥补和减小数据间的相关性, 用一个学习算法 (learning method) 选择二进制测试的一个子集。

在描述符匹配中使用了对传统 LSH 改善后的多探针 LSH。文章中说 ORB 算法比 SURF 和 SIFT 算法快的多, ORB 描述符也比 SURF 好很多。ORB 是低功耗设备的最佳选择。

### 36.1 OpenCV 中的 ORB 算法

和前面一样我们首先要使用函数 **cv3.ORB()** 或者 **feature2d** 通用接口创建一个 ORB 对象。它有几个可选参数。最有用的应该是 **nfeature**, 默认值为 500, 它表示了要保留特征的最大数目。**scoreType** 设置使用 Harris 打分还是使用 FAST 打分对特征进行排序 (默认是使用 Harris 打分) 等。参数 **WTA\_K** 决定了产生每个 oriented\_BRIEF 描述符要使用的像素点的数目。默认值是 2, 也就是一次选择两个点。在这种情况下进行匹配, 要使用 **NORM\_HAMMING** 距离。如果 WTA\_K 被设置成 3 或 4, 那匹配距离就要设置为 **NORM\_HAMMING2**。

下面是一个使用 ORB 的简单代码。

```

# -*- coding: utf-8 -*-
"""
Created on Thu Jan 23 22:41:39 2014

@author: duan
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt

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

# Initiate STAR detector
orb = cv2.ORB()

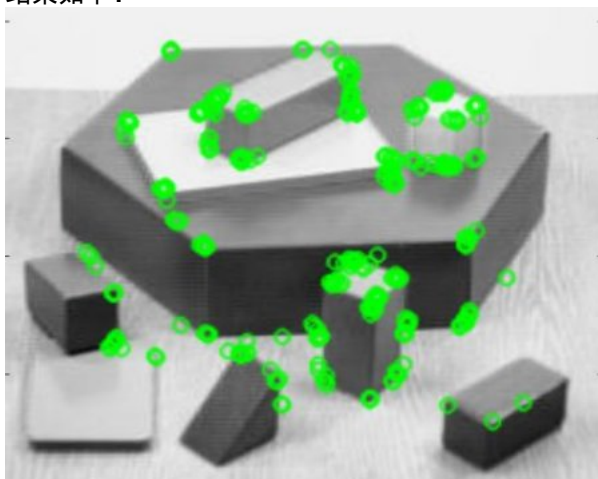
# find the keypoints with ORB
kp = orb.detect(img,None)

# compute the descriptors with ORB
kp, des = orb.compute(img, kp)

# draw only keypoints location,not size and orientation
img2 = cv2.drawKeypoints(img,kp,color=(0,255,0), flags=0)
plt.imshow(img2),plt.show()

```

结果如下：



我们将在其他章节介绍 ORB 特征匹配。

## 更多资源

1. Ethan Rublee, Vincent Rabaud, Kurt Konolige, Gary R. Bradski: ORB: An efficient alternative to SIFT or SURF. ICCV 2011:

2564-2571.

**练习**

## 37 特征匹配

### 目标

- 我们将要学习在图像间进行特征匹配
- 使用 OpenCV 中的蛮力 (Brute-Force) 匹配和 FLANN 匹配

### 37.1 Brute-Force 匹配的基础

蛮力匹配器是很简单的。首先在第一幅图像中选取一个关键点然后依次与第二幅图像的每个关键点进行 (描述符) 距离测试, 最后返回距离最近的关键点。

对于 BF 匹配器, 我们首先要使用 **cv2.BFMatcher()** 创建一个 BF-Matcher 对象。它有两个可选参数。第一个是 **normType**。它是用来指定要使用的距离测试类型。默认值为 **cv2.Norm\_L2**。这很适合 SIFT 和 SURF 等 (**cv2.NORM\_L1** 也可以)。对于使用二进制描述符的 ORB, BRIEF, BRISK 算法等, 要使用 **cv2.NORM\_HAMMING**, 这样就会返回两个测试对象之间的汉明距离。如果 ORB 算法的参数设置为 **VTA\_K=3** 或 **4**, **normType** 就应该设置成 **cv2.NORM\_HAMMING2**。

第二个参数是布尔变量 **crossCheck**, 默认值为 **False**。如果设置为 **True**, 匹配条件就会更加严格, 只有到 A 中的第 i 个特征点与 B 中的第 j 个特征点距离最近, 并且 B 中的第 j 个特征点到 A 中的第 i 个特征点也是最近 (A 中没有其他点到 j 的距离更近) 时才会返回最佳匹配 (i, j)。也就是这两个特征点要互相匹配才行。这样就能提供统一的结果, 这可以用来替代 D.Lowe 在 SIFT 文章中提出的比值测试方法。

BFMatcher 对象具有两个方法, **BFMatcher.match()** 和 **BFMatcher.knnMatch()**。第一个方法会返回最佳匹配。第二个方法为每个关键点返回 k 个最佳匹配 (降序排列之后取前 k 个), 其中 k 是由用户设定的。如果除了匹配之外还要做其他事情的话可能会用上 (比如进行比值测试)。

就像使用 **cv2.drawKeypoints()** 绘制关键点一样, 我们可以使用 **cv2.drawMatches()** 来绘制匹配的点。它会将这两幅图像先水平排列, 然后在最佳匹配的点之间绘制直线 (从原图像到目标图像)。如果前面使用的是 **BFMatcher.knnMatch()**, 现在我们可以使用函数 **cv2.drawMatchesKnn** 为每个关键点和它的 k 个最佳匹配点绘制匹配线。如果 k 等于 2, 就会为每个关键点绘制两条最佳匹配直线。如果我们要选择性绘制的话就要给函数传入一个掩模。

让我们分别看一个 ORB 和一个 SURF 的例子吧。(使用不同距离计算方法)。

## 37.2 对 ORB 描述符进行暴力匹配

现在我们看一个在两幅图像之间进行特征匹配的简单例子。在本例中我们有一个查询图像和一个目标图像。我们要使用特征匹配的方法在目标图像中寻找查询图像的位置。(这两幅图像分别是/sample/c/box.png, 和/sample/c/box\_in\_scene.png)

我们使用 ORB 描述符来进行特征匹配。首先我们需要加载图像计算描述符。

```
# -*- coding: utf-8 -*-
"""
Created on Fri Jan 24 15:10:41 2014

@author: duan
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt

img1 = cv2.imread('box.png',0)          # queryImage
img2 = cv2.imread('box_in_scene.png',0) # trainImage
```

下面我们要创建一个 BFMatcher 对象, 并将距离计算设置为 **cv2.NORM\_HAMMING**(因为我们使用的是 ORB), 并将 **crossCheck** 设置为 **True**。然后使用 **Matcher.match()** 方法获得两幅图像的最佳匹配。然后将匹配结果按特征点之间的距离进行降序排列, 这样最佳匹配就会排在前面了。最后我们只将前 10 个匹配绘制出来 (太多了看不清, 如果愿意的话你可以多画几条)。

```
# Initiate SIFT detector
orb = cv2.ORB()

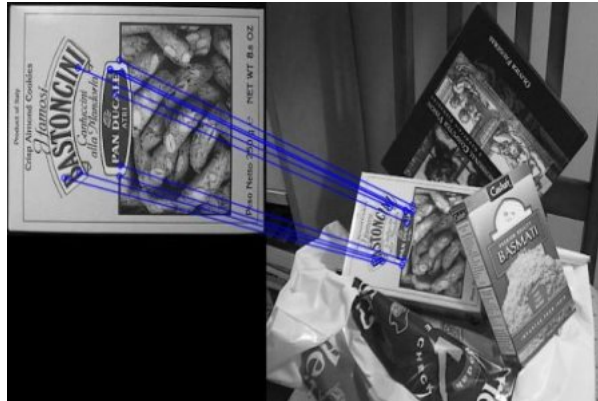
# find the keypoints and descriptors with SIFT
kp1, des1 = orb.detectAndCompute(img1,None)
kp2, des2 = orb.detectAndCompute(img2,None)

# create BFMatcher object
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# Match descriptors.
matches = bf.match(des1,des2)

# Sort them in the order of their distance.
matches = sorted(matches, key = lambda x:x.distance)
```

下面就是我得到的结果。



### 37.3 匹配器对象是什么？

`matches = bf.match(des1, des2)` 返回值是一个 `DMatch` 对象列表。这个 `DMatch` 对象具有下列属性：

- **DMatch.distance** - 描述符之间的距离。越小越好。
- **DMatch.trainIdx** - 目标图像中描述符的索引。
- **DMatch.queryIdx** - 查询图像中描述符的索引。
- **DMatch.imgIdx** - 目标图像的索引。

### 37.4 对 SIFT 描述符进行蛮力匹配和比值测试

现在我们使用 `BFMatcher.knnMatch()` 来获得  $k$  对最佳匹配。在本例中我们设置  $k = 2$ ，这样我们就可以使用 D.Lowe 文章中的比值测试了。



```

import numpy as np
import cv2
from matplotlib import pyplot as plt

img1 = cv2.imread('box.png',0)          # queryImage
img2 = cv2.imread('box_in_scene.png',0) # trainImage

# Initiate SIFT detector
sift = cv2.SIFT()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

# BFMatcher with default params
bf = cv2.BFMatcher()
matches = bf.knnMatch(des1,des2, k=2)

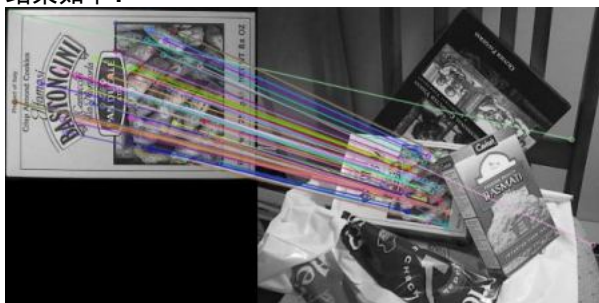
# Apply ratio test
# 比值测试，首先获取与 A 距离最近的点 B（最近）和 C（次近），只有当 B/C
# 小于阈值时（0.75）才被认为是匹配，因为假设匹配是一一对应的，真正的匹配的理想距离为 0
good = []
for m,n in matches:
    if m.distance < 0.75*n.distance:
        good.append([m])

# cv2.drawMatchesKnn expects list of lists as matches.
img3 = cv2.drawMatchesKnn(img1,kp1,img2,kp2,good[:10],flags=2)

plt.imshow(img3),plt.show()

```

结果如下：



## 37.5 FLANN 匹配器

FLANN 是快速最近邻搜索包 (Fast\_Library\_for\_Approximate\_Nearest\_Neighbors) 的简称。它是一个对大数据集和高维特征进行最近邻搜索的算法的集合，而且这些算法都已经被优化过了。在面对大数据集时它的效果要好于 BFMatcher。我们来对第二个例子使用 FLANN 匹配看看它的效果。

使用 FLANN 匹配，我们需要传入两个字典作为参数。这两个用来确定要使用的算法和其他相关参数等。第一个是 **IndexParams**。各种不同算法的信息可以在 FLANN 文档中找到。这里我们总结一下，对于 SIFT 和 SURF 等，我们可以传入的参数是：

```
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
```

但使用 ORB 时，我们要传入的参数如下。注释掉的值是文献中推荐使用的，但是它们并不适合所有情况，其他值的效果可能会更好。

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

img1 = cv2.imread('box.png',0)          # queryImage
img2 = cv2.imread('box_in_scene.png',0) # trainImage

# Initiate SIFT detector
sift = cv2.SIFT()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

# BFMatcher with default params
bf = cv2.BFMatcher()
matches = bf.knnMatch(des1,des2, k=2)

# Apply ratio test
good = []
for m,n in matches:
    if m.distance < 0.75*n.distance:
        good.append([m])

# cv2.drawMatchesKnn expects list of lists as matches.
img3 = cv2.drawMatchesKnn(img1,kp1,img2,kp2,good,flags=2)

plt.imshow(img3),plt.show()
```

第二个字典是 SearchParams。用它来指定递归遍历的次数。值越高结果越准确，但是消耗的时间也越多。如果你想修改这个值，传入参数：  
*search\_params = dict(checks = 100)*。

有了这些信息我们就可以开始了。

```

# -*- coding: utf-8 -*-
"""
Created on Fri Jan 24 15:36:42 2014

@author: duan
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt

img1 = cv2.imread('box.png',0) # queryImage
img2 = cv2.imread('box_in_scene.png',0) # trainImage

# Initiate SIFT detector
sift = cv2.SIFT()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

# FLANN parameters
FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks=50) # or pass empty dictionary

flann = cv2.FlannBasedMatcher(index_params,search_params)

matches = flann.knnMatch(des1,des2,k=2)

# Need to draw only good matches, so create a mask
matchesMask = [[0,0] for i in xrange(len(matches))]

# ratio test as per Lowe's paper
for i,(m,n) in enumerate(matches):
    if m.distance < 0.7*n.distance:
        matchesMask[i]=[1,0]

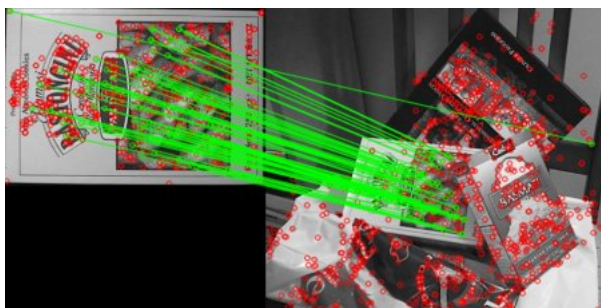
draw_params = dict(matchColor = (0,255,0),
                    singlePointColor = (255,0,0),
                    matchesMask = matchesMask,
                    flags = 0)

img3 = cv2.drawMatchesKnn(img1,kp1,img2,kp2,matches,None,**draw_params)

plt.imshow(img3,),plt.show()

```

结果如下：



**更多资源**

**练习**

## 38 使用特征匹配和单应性查找对象

### 目标

- 联合使用特征提取和 `calib3d` 模块中的 `findHomography` 在复杂图像中查找已知对象。

### 38.1 基础

还记得上一节我们做了什么吗？我们使用一个查询图像，在其中找到一些特征点（关键点），我们又在另一幅图像中也找到了一些特征点，最后对这两幅图像之间的特征点进行匹配。简单来说就是：我们在一张杂乱的图像中找到了一个对象（的某些部分）的位置。这些信息足以帮助我们在目标图像中准确的找到（查询图像）对象。

为了达到这个目的我们可以使用 `calib3d` 模块中的 `cv2.findHomography()` 函数。如果将这两幅图像中的特征点集传给这个函数，他就会找到这个对象的透视图变换。然后我们就可以使用函数 `cv2.perspectiveTransform()` 找到这个对象了。至少要 4 个正确的点才能找到这种变换。

我们已经知道在匹配过程可能会有一些错误，而这些错误会影响最终结果。为了解决这个问题，算法使用 RANSAC 和 LEAST\_MEDIAN(可以通过参数来设定)。所以好的匹配提供的正确的估计被称为 `inliers`，剩下的被称为 `outliers`。`cv2.findHomography()` 返回一个掩模，这个掩模确定了 `inlier` 和 `outlier` 点。

让我们来搞定它吧 !!!

### 38.2 代码

和通常一样我们先在图像中来找到 SIFT 特征点，然后再使用比值测试找到最佳匹配。

```

# -*- coding: utf-8 -*-
"""
Created on Fri Jan 24 20:11:52 2014

@author: duan
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt

MIN_MATCH_COUNT = 10

img1 = cv2.imread('box.png',0)          # queryImage
img2 = cv2.imread('box_in_scene.png',0) # trainImage

# Initiate SIFT detector
sift = cv2.SIFT()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks = 50)

flann = cv2.FlannBasedMatcher(index_params, search_params)

matches = flann.knnMatch(des1,des2,k=2)

# store all the good matches as per Lowe's ratio test.
good = []
for m,n in matches:
    if m.distance < 0.7*n.distance:
        good.append(m)

```

现在我们设置只有存在 10 个以上匹配时才去查找目标 ( MIN\_MATCH\_COUNT=10 ), 否则显示警告消息: “现在匹配不足 !”

如果找到了足够的匹配, 我们要提取两幅图像中匹配点的坐标。把它们传入到函数中计算透视变换。一旦我们找到 3x3 的变换矩阵, 就可以使用它将查询图像的四个顶点 ( 四个角 ) 变换到目标图像中去了。然后再绘制出来。

```

if len(good)>MIN_MATCH_COUNT:
    # 获取关键点的坐标
    src_pts = np.float32([ kp1[m.queryIdx].pt for m in good ]).reshape(-1,1,2)
    dst_pts = np.float32([ kp2[m.trainIdx].pt for m in good ]).reshape(-1,1,2)

    # 第三个参数 Method used to computed a homography matrix. The following methods are possible:
    #0 - a regular method using all the points
    #CV_RANSAC - RANSAC-based robust method
    #CV_LMEDS - Least-Median robust method
    # 第四个参数取值范围在 1 到 10, 拒绝一个点对的阈值。原图像的点经过变换后点与目标图像上对应点的误差
    # 超过误差就认为是 outlier
    # 返回值中 M 为变换矩阵。
    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC,5.0)
    matchesMask = mask.ravel().tolist()
    # 获得原图像的高和宽
    h,w = img1.shape
    # 使用得到的变换矩阵对原图像的四个角进行变换, 获得在目标图像上对应的坐标。
    pts = np.float32([ [0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2)
    dst = cv2.perspectiveTransform(pts,M)
    # 原图像为灰度图
    cv2.polylines(img2,[np.int32(dst)],True,255,10, cv2.LINE_AA)

else:
    print "Not enough matches are found - %d/%d" % (len(good),MIN_MATCH_COUNT)
    matchesMask = None

```

最后我再绘制 inliers (如果能成功的找到目标图像的话) 或者匹配的关键点 (如果失败)。

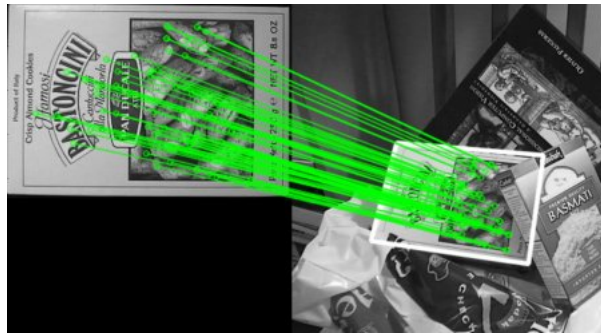
```

draw_params = dict(matchColor = (0,255,0), # draw matches in green color
                    singlePointColor = None,
                    matchesMask = matchesMask, # draw only inliers
                    flags = 2)

img3 = cv2.drawMatches(img1,kp1,img2,kp2,good,None,**draw_params)
plt.imshow(img3, 'gray'),plt.show()

```

结果如下。复杂图像中被找到的目标图像被标记成白色。



**更多资源**

**练习**



## 部分 VI

# 视频分析

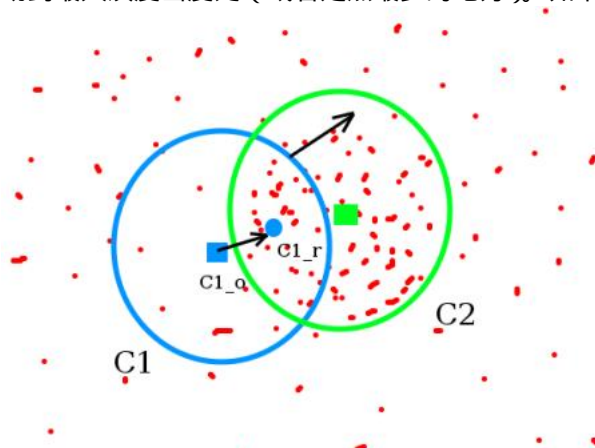
## 39 Meanshift 和 Camshift

### 目标

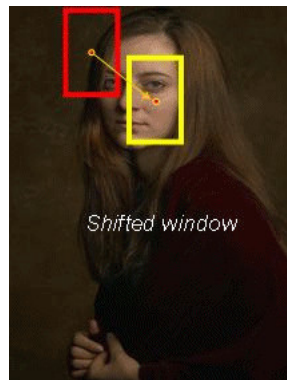
- 本节我们要学习使用 Meanshift 和 Camshift 算法在视频中找到并跟踪目标对象

### 39.1 Meanshift

Meanshift 算法的基本原理是很简单的。假设我们有一堆点（比如直方图反向投影得到的点），和一个小的圆形窗口，我们要完成的任务就是将这个窗口移动到最大灰度密度处（或者是点最多的地方）。如下图所示：



初始窗口是蓝色的“C1”，它的圆心为蓝色方框“C1\_o”，而窗口中所有点质心却是“C1\_r”（小的蓝色圆圈），很明显圆心和点的质心没有重合。所以移动圆心 C1\_o 到质心 C1\_r，这样我们就得到了一个新的窗口。这时又可以找到新窗口内所有点的质心，大多数情况下还是不重合的，所以重复上面的操作：将新窗口的中心移动到新的质心。就这样不停的迭代操作直到窗口的中心和其所包含点的质心重合为止（或者有一小误差）。按照这样的操作我们的窗口最终会落在像素值（和）最大的地方。如上图所示“C2”是窗口的最后位址，我们可以看出来这个窗口中的像素点最多。整个过程如下图所示：



通常情况下我们要使用直方图方向投影得到的图像和目标对象的起始位置。当目标对象的移动会反映到直方图反向投影图中。就这样，meanshift 算法就把我们的窗口移动到图像中灰度密度最大的区域了。

## 39.2 OpenCV 中的 Meanshift

要在 OpenCV 中使用 Meanshift 算法首先我们要对目标对象进行设置，计算目标对象的直方图，这样在执行 meanshift 算法时我们就可以将目标对象反向投影到每一帧中去了。另外我们还需要提供窗口的起始位置。在这里我们值计算 H (Hue) 通道的直方图，同样为了避免低亮度造成的影响，我们使用函数 `cv2.inRange()` 将低亮度的值忽略掉。

```

# -*- coding: utf-8 -*-
"""
Created on Mon Jan 27 08:02:04 2014

@author: duan
"""

import numpy as np
import cv2

cap = cv2.VideoCapture('slow.flv')

# take first frame of the video
ret, frame = cap.read()

# setup initial location of window
r,h,c,w = 250,90,400,125 # simply hardcoded the values
track_window = (c,r,w,h)

# set up the ROI for tracking
roi = frame[r:r+h, c:c+w]
hsv_roi = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
mask = cv2.inRange(hsv_roi, np.array((0., 60.,32.)), np.array((180.,255.,255.)))
roi_hist = cv2.calcHist([hsv_roi],[0],mask,[180],[0,180])
cv2.normalize(roi_hist,roi_hist,0,255,cv2.NORM_MINMAX)

# Setup the termination criteria, either 10 iteration or move by atleast 1 pt
term_crit = ( cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1 )

while(1):
    ret , frame = cap.read()

    if ret == True:
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        dst = cv2.calcBackProject([hsv],[0],roi_hist,[0,180],1)

        # apply meanshift to get the new location
        ret, track_window = cv2.meanShift(dst, track_window, term_crit)

        # Draw it on image
        x,y,w,h = track_window
        img2 = cv2.rectangle(frame, (x,y), (x+w,y+h), 255,2)
        cv2.imshow('img2',img2)

        k = cv2.waitKey(60) & 0xff
        if k == 27:
            break
        else:
            cv2.imwrite(chr(k)+".jpg",img2)

    else:
        break

cv2.destroyAllWindows()
cap.release()

```

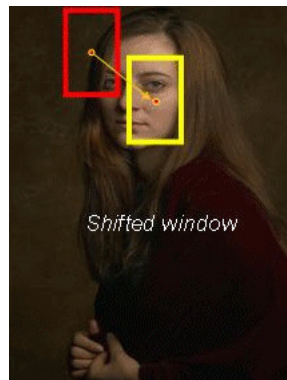
下面是我使用 meanshift 算法对一个视频前三帧分析的结果：



### 39.3 Camshift

你认真看上面的结果了吗？这里面还有一个问题。我们的窗口的大小是固定的，而汽车由远及近（在视觉上）是一个逐渐变大的过程，固定的窗口是不合适的。所以我们需要根据目标的大小和角度来对窗口的大小和角度进行修订。OpenCV Labs 为我们带来的解决方案（1988 年）：一个被叫做 CAMshift 的算法。

这个算法首先要使用 meanshift，meanshift 找到（并覆盖）目标之后，再去调整窗口的大小， $s = 2x\sqrt{\frac{M00}{256}}$ 。它还会计算目标对象的最佳外接椭圆的角度，并以此调节窗口角度。然后使用更新后的窗口大小和角度来在原来的位置继续进行 meanshift。重复这个过程知道达到需要的精度。



### 39.4 OpenCV 中的 Camshift

与 Meanshift 基本一样，但是返回的结果是一个带旋转角度的矩形（这是我们的结果），以及这个矩形的参数（被用到下一次迭代过程中）。下面是代码：

```

# -*- coding: utf-8 -*-
"""
Created on Mon Jan 27 08:03:49 2014

@author: duan
"""

import numpy as np
import cv2

cap = cv2.VideoCapture('slow.flv')

# take first frame of the video
ret, frame = cap.read()

# setup initial location of window
r,h,c,w = 250,90,400,125 # simply hardcoded the values
track_window = (c,r,w,h)

# set up the ROI for tracking
roi = frame[r:r+h, c:c+w]
hsv_roi = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
mask = cv2.inRange(hsv_roi, np.array((0., 60.,32.)), np.array((180.,255.,255.)))
roi_hist = cv2.calcHist([hsv_roi],[0],mask,[180],[0,180])
cv2.normalize(roi_hist,roi_hist,0,255,cv2.NORM_MINMAX)

# Setup the termination criteria, either 10 iteration or move by atleast 1 pt
term_crit = ( cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1 )

while(1):
    ret , frame = cap.read()

    if ret == True:
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        dst = cv2.calcBackProject([hsv],[0],roi_hist,[0,180],1)

        # apply meanshift to get the new location
        ret, track_window = cv2.CamShift(dst, track_window, term_crit)

        # Draw it on image
        pts = cv2.boxPoints(ret)
        pts = np.int0(pts)
        img2 = cv2.polylines(frame,[pts],True, 255,2)
        cv2.imshow('img2',img2)

        k = cv2.waitKey(60) & 0xff
        if k == 27:
            break
        else:
            cv2.imwrite(chr(k)+".jpg",img2)

    else:
        break

cv2.destroyAllWindows()
cap.release()

```

对三帧图像分析的结果如下：



## 更多资源

1. French Wikipedia page on Camshift. (The two animations are taken from here)
2. Bradski, G.R., "Real time face and object tracking as a component of a perceptual user interface," Applications of Computer Vision, 1998. WACV '98. Proceedings., Fourth IEEE Workshop on , vol., no., pp.214,219, 19-21 Oct 1998

## 练习

1. OpenCV 的官方示例中有一个 camshift 的交互式演示，搞定它吧！

```

#!/usr/bin/env python

'''
Camshift tracker
=====

This is a demo that shows mean-shift based tracking
You select a color objects such as your face and it tracks it.
This reads from video camera (0 by default, or the camera number the user enters)

http://www.robinhewitt.com/research/track/camshift.html

Usage:
-----
    camshift.py [<video source>]

    To initialize tracking, select the object with mouse

Keys:
-----
    ESC    - exit
    b      - toggle back-projected probability visualization
'''

import numpy as np
import cv2

# local module
import video

class App(object):
    def __init__(self, video_src):
        self.cam = video.create_capture(video_src)
        ret, self.frame = self.cam.read()
        cv2.namedWindow('camshift')
        cv2.setMouseCallback('camshift', self.onmouse)

        self.selection = None
        self.drag_start = None
        self.tracking_state = 0
        self.show_backproj = False

    def onmouse(self, event, x, y, flags, param):
        x, y = np.int16([x, y])
        if event == cv2.EVENT_LBUTTONDOWN:
            self.drag_start = (x, y)
            self.tracking_state = 0
            # 官方示例中下面一行判断有问题, 作如下修改就可以了
            if self.drag_start and event == cv2.EVENT_MOUSEMOVE:
                # print x,y
                if flags==cv2.EVENT_FLAG_LBUTTON:
                    # print 'ok'
                    h, w = self.frame.shape[:2]
                    x0, y0 = np.maximum(0, np.minimum([x0, y0], [x, y]))
                    x1, y1 = np.minimum([w, h], np.maximum([x0, y0], [x, y]))
                    self.selection = None
                    if x1-x0 > 0 and y1-y0 > 0:
                        self.selection = (x0, y0, x1, y1)
                        print self.selection
                    else:
                        self.drag_start = None
                        if self.selection is not None:
                            self.tracking_state = 1

    def show_hist(self):
        bin_count = self.hist.shape[0]

```





## 40 光流

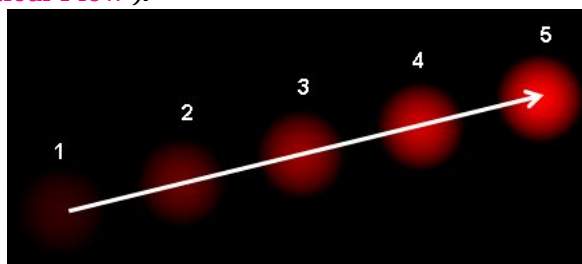
### 目标

本节我们将要学习：

- 光流的概念以及 Lucas-Kanade 光流法
- 使用函数 `cv2.calcOpticalFlowPyrLK()` 对图像中的特征点进行跟踪

### 40.1 光流

由于目标对象或者摄像机的移动造成的图像对象在连续两帧图像中的移动被称为光流。它是一个 2D 向量场，可以用来显示一个点从第一帧图像到第二帧图像之间的移动。如下图所示 ( [Image Courtesy: Wikipedia article on Optical Flow](#) ):



上图显示了一个点在连续的五帧图像间的移动。箭头表示光流场向量。光流在很多领域中都很有用：

- 由运动重建结构
- 视频压缩
- Video Stabilization 等

光流是基于一下假设的：

1. 在连续的两帧图像之间（目标对象的）像素的灰度值不改变。
2. 相邻的像素具有相同的运动

第一帧图像中的像素  $I(x, y, t)$  在时间  $dt$  后移动到第二帧图像的  $(x+dx, y+dy)$  处。根据第一条假设：灰度值不变。所以我们可以得到：

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

对等号右侧进行泰勒级数展开，消去相同项，两边都除以  $dt$ ，得到如下方程：

$$f_x u + f_y v + f_t = 0$$

其中：

$$f_x = \frac{\partial f}{\partial x}; f_y = \frac{\partial f}{\partial y}$$

$$u = \frac{dx}{dt}; v = \frac{dy}{dt}$$

上边的等式叫做光流方程。其中  $f_x$  和  $f_y$  是图像梯度，同样  $f_t$  是时间方向的梯度。但  $(u, v)$  是不知道的。我们不能在一个等式中求解两个未知数。有几个方法可以帮我们解决这个问题，其中的一个是 Lucas-Kanade 法

## 40.2 Lucas-Kanade 法

现在我们要使用第二条假设，邻域内的所有点都有相似的运动。Lucas-Kanade 法就是利用一个 3x3 邻域中的 9 个点具有相同运动的这一点。这样我们就可以找到这 9 个点的光流方程，用它们组成一个具有两个未知数 9 个等式的方程组，这是一个约束条件过多的方程组。一个好的解决方法就是使用最小二乘拟合。下面就是求解结果：

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i f_{x_i}^2 & \sum_i f_{x_i} f_{y_i} \\ \sum_i f_{x_i} f_{y_i} & \sum_i f_{y_i}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i f_{x_i} f_{t_i} \\ -\sum_i f_{y_i} f_{t_i} \end{bmatrix}$$

（有没有发现上边的逆矩阵与 Harris 角点检测器非常相似，这说明角点很适合被用来做跟踪）

从使用者的角度来看，想法很简单，我们取跟踪一些点，然后我们就会获得这些点的光流向量。但是还有一些问题。直到现在我们处理的都是很小的运动。如果有大的运动怎么办呢？图像金字塔。我们可以使用图像金字塔的顶层，此时小的运动被移除，大的运动装换成了小的运动，现在再使用 Lucas-Kanade 算法，我们就会得到尺度空间上的光流。

## 40.3 OpenCV 中的 Lucas-Kanade 光流

上述所有过程都被 OpenCV 打包成了一个函数：`cv2.calcOpticalFlowPyrLK()`。现在我们使用这个函数创建一个小程序来跟踪视频中的一些点。要跟踪那些点呢？我们使用函数 `cv2.goodFeatureToTrack()` 来确定要跟踪的点。我们首先在视频的第一帧图像中检测一些 Shi-Tomasi 角点，然后我们使用 Lucas-Kanade 算法迭代跟踪这些角点。我们要给函数 `cv2.calcOpticalFlowPyrLK()`

传入前一帧图像和其中的点，以及下一帧图像。函数将返回带有状态数的点，如果状态数是 1，那说明在下一帧图像中找到了这个点（上一帧中角点），如果状态数是 0，就说明没有在下一帧图像中找到这个点。我们再把些点作为参数传给函数，如此迭代下去实现跟踪。代码如下：

（上面的代码没有对返回角点的正确性进行检查。图像中的一些特征点甚至在丢失以后，光流还会找到一个预期相似的点。所以为了实现稳定的跟踪，我们应该每个一定间隔就要进行一次角点检测。OpenCV 的官方示例中带有这样一个例子，它是每 5 帧进行一个特征点检测。它还对光流点使用反向检测来选取好的点进行跟踪。示例为 `/samples/python2/lk_track.py`）

```

# -*- coding: utf-8 -*-
"""
Created on Mon Jan 27 12:28:25 2014

@author: duan
"""

import numpy as np
import cv2

cap = cv2.VideoCapture('slow.flv')

# params for ShiTomasi corner detection
feature_params = dict( maxCorners = 100,
                       qualityLevel = 0.3,
                       minDistance = 7,
                       blockSize = 7 )

# Parameters for lucas kanade optical flow
#maxLevel 为使用的图像金字塔层数
lk_params = dict( winSize = (15,15),
                  maxLevel = 2,
                  criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 0.03))

# Create some random colors
color = np.random.randint(0,255,(100,3))

# Take first frame and find corners in it
ret, old_frame = cap.read()
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
p0 = cv2.goodFeaturesToTrack(old_gray, mask = None, **feature_params)

# Create a mask image for drawing purposes
mask = np.zeros_like(old_frame)

while(1):
    ret, frame = cap.read()
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # calculate optical flow 能够获取点的新位置
    p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None, **lk_params)

    # Select good points
    good_new = p1[st==1]
    good_old = p0[st==1]

    # draw the tracks
    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(30) & 0xff
    if k == 27:
        break

    # Now update the previous frame and previous points
    old_gray = frame_gray.copy()
    p0 = good_new.reshape(-1,1,2)

cv2.destroyAllWindows()
cap.release()

```

下面是我的到的结果：



## 40.4 OpenCV 中的稠密光流

Lucas-Kanade 法是计算一些特征点的光流（我们上面的例子使用的是 Shi-Tomasi 算法检测到的角点）。OpenCV 还提供了一种计算稠密光流的方法。它会计算图像中的所有点的光流。这是基于 Gunner\_Farneback 的算法（2003 年）。

下面的例子就是使用上面的算法计算稠密光流。结果是一个带有光流向量  $(u, v)$  的双通道数组。通过计算我们能得到光流的大小和方向。我们使用颜色对结果进行编码以便于更好的观察。方向对应于 H（Hue）通道，大小对应于 V（Value）通道。代码如下：

```

# -*- coding: utf-8 -*-
"""
Created on Mon Jan 27 12:28:46 2014

@author: duan
"""

import cv2
import numpy as np
cap = cv2.VideoCapture("vtest.avi")

ret, frame1 = cap.read()
prvs = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)
hsv = np.zeros_like(frame1)
hsv[..., 1] = 255

while(1):
    ret, frame2 = cap.read()
    next = cv2.cvtColor(frame2, cv2.COLOR_BGR2GRAY)
    #cv2.calcOpticalFlowFarneback(prev, next, pyr_scale, levels, winsize, iterations, poly_n,
    #poly_sigma, flags)
    #pyr_scale - parameter, specifying the image scale (<1) to build pyramids for each image;
    #pyr_scale=0.5 means a classical pyramid, where each next layer is twice smaller than the
    #previous one.
    #poly_n - size of the pixel neighborhood used to find polynomial expansion in each pixel;
    #typically poly_n =5 or 7.
    #poly_sigma - standard deviation of the Gaussian that is used to smooth derivatives used
    #as a basis for the polynomial expansion; for poly_n=5, you can set poly_sigma=1.1, for
    #poly_n=7, a good value would be poly_sigma=1.5.
    #flag 可选 0 或 1, 0 计算快, 1 慢但准确
    flow = cv2.calcOpticalFlowFarneback(prvs, next, None, 0.5, 3, 15, 3, 5, 1.2, 0)

    #cv2.cartToPolar Calculates the magnitude and angle of 2D vectors.
    mag, ang = cv2.cartToPolar(flow[..., 0], flow[..., 1])
    hsv[..., 0] = ang*180/np.pi/2
    hsv[..., 2] = cv2.normalize(mag, None, 0, 255, cv2.NORM_MINMAX)
    rgb = cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)

    cv2.imshow('frame2', rgb)
    k = cv2.waitKey(30) & 0xff
    if k == 27:
        break
    elif k == ord('s'):
        cv2.imwrite('opticalfb.png', frame2)
        cv2.imwrite('opticalhsv.png', rgb)
        prvs = next

cap.release()
cv2.destroyAllWindows()

```

结果如下：



OpenCV 的官方示例中有一个更高级的稠密光流/[samples/python2/opt\\_flow.py](#)，去搞定它吧！

## 更多资源

### 练习

1. Check the code in [samples/python2/lk\\_track.py](#). Try to understand the code.
2. Check the code in [samples/python2/opt\\_flow.py](#). Try to understand the code.



## 41 背景减除

### 目标

- 本节我们将要学习 OpenCV 中的背景减除方法

#### 41.1 基础

在很多基础应用中背景检出都是一个非常重要的步骤。例如顾客统计，使用一个静态摄像头来记录进入和离开房间的人数，或者是交通摄像头，需要提取交通工具的信息等。在所有的这些例子中，首先要将人或车单独提取出来。技术上来说，我们需要从静止的背景中提取移动的前景。

如果你有一张背景（仅有背景不含前景）图像，比如没有顾客的房间，没有交通工具的道路等，那就好办了。我们只需要在新的图像中减去背景就可以得到前景对象了。但是在大多数情况下，我们没有这样的（背景）图像，所以我们需要从我们有的图像中提取背景。如果图像中的交通工具还有影子的话，那这个工作就更难了，因为影子也在移动，仅仅使用减法会把影子也当成前景。真是一件很复杂的事情。

为了实现这个目的科学家们已经提出了几种算法。OpenCV 中已经包含了其中三种比较容易使用的方法。我们一个一个学习一下吧。

#### 41.2 BackgroundSubtractorMOG

这是一个以混合高斯模型为基础的前景/背景分割算法。它是 P.KadewTraKuPong 和 R.Bowden 在 2001 年提出的。它使用  $K$  ( $K=3$  或  $5$ ) 个高斯分布混合对背景像素进行建模。使用这些颜色（在整个视频中）存在时间的长短作为混合的权重。背景的颜色一般持续的时间最长，而且更加静止。一个像素怎么会有分布呢？在  $x, y$  平面上一个像素就是一个像素没有分布，但是我们现在讲的背景建模是基于时间序列的，因此每一个像素点所在的位置在整个时间序列中就会有多个值，从而构成一个分布。

在编写代码时，我们需要使用函数：**cv2.createBackgroundSubtractorMOG()** 创建一个背景对象。这个函数有些可选参数，比如要进行建模场景的时间长度，高斯混合成分的数量，阈值等。将他们全部设置为默认值。然后在整个视频中我们是需要使用 **backgroundsubtractor.apply()** 就可以得到前景的掩模了。

下面是一个简单的例子：

```

# -*- coding: utf-8 -*-
"""
Created on Mon Jan 27 18:13:09 2014

@author: duan
"""

import numpy as np
import cv2

cap = cv2.VideoCapture(0)

fgbg = cv2.createBackgroundSubtractorMOG()

while(1):
    ret, frame = cap.read()

    fgmask = fgbg.apply(frame)

    cv2.imshow('frame', fgmask)
    k = cv2.waitKey(30) & 0xff
    if k == 27:
        break

cap.release()
cv2.destroyAllWindows()

```

### 41.3 BackgroundSubtractorMOG2

这个也是以高斯混合模型为基础的背景/前景分割算法。它是以 2004 年和 2006 年 Z.Zivkovic 的两篇文章为基础的。这个算法的一个特点是它为每一个像素选择一个合适数目的高斯分布。(上一个方法中我们使用是 K 高斯分布)。这样就会对由于亮度等发生变化引起的场景变化产生更好的适应。

和前面一样我们需要创建一个背景对象。但在这里我们我们可以选择是否检测阴影。如果 `detectShadows = True` (默认值)，它就会检测并将影子标记出来，但是这样做会降低处理速度。影子会被标记为灰色。

```

# -*- coding: utf-8 -*-
"""
Created on Mon Jan 27 18:16:29 2014

@author: duan
"""

import numpy as np
import cv2

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

fgbg = cv2.createBackgroundSubtractorMOG2()

while(1):
    ret, frame = cap.read()

    fgmask = fgbg.apply(frame)

    cv2.imshow('frame', fgmask)
    k = cv2.waitKey(30) & 0xff
    if k == 27:
        break

cap.release()
cv2.destroyAllWindows()

```

## 41.4 BackgroundSubtractorGMG

此算法结合了静态背景图像估计和每个像素的贝叶斯分割。这是 2012 年 Andrew\_B.Godbehere, Akihiro\_Matsukawa 和 Ken\_Goldberg 在文章中提出的。

它使用前面很少的图像（默认为前 120 帧）进行背景建模。使用了概率前景估计算法（使用贝叶斯估计鉴定前景）。这是一种自适应的估计，新观察到的对象比旧的对象具有更高的权重，从而对光照变化产生适应。一些形态学操作如开运算闭运算等被用来除去不需要的噪音。在前几帧图像中你会得到一个黑色窗口。

对结果进行形态学开运算对与去除噪声很有帮助。

```

# -*- coding: utf-8 -*-
"""
Created on Mon Jan 27 18:18:52 2014

@author: duan
"""

import numpy as np
import cv2

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

kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3))
fgbg = cv2.createBackgroundSubtractorGMG()

while(1):
    ret, frame = cap.read()

    fgmask = fgbg.apply(frame)
    fgmask = cv2.morphologyEx(fgmask, cv2.MORPH_OPEN, kernel)

    cv2.imshow('frame', fgmask)
    k = cv2.waitKey(30) & 0xff
    if k == 27:
        break

cap.release()
cv2.destroyAllWindows()

```

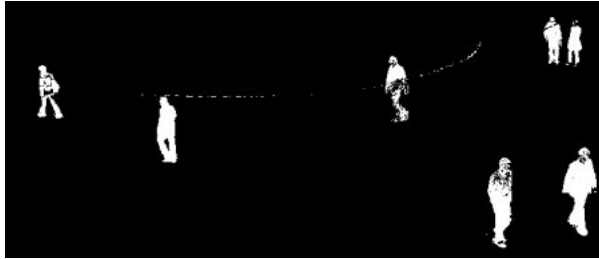
## 41.5 结果

### 原始图像

下图显示了一段视频中的第 200 帧图像



### BackgroundSubtractorMOG 的结果



### BackgroundSubtractorMOG2 的结果

灰色区域代表阴影



### BackgroundSubtractorGMG 的结果

使用形态学开运算将噪音去除。



更多资源

练习

## 部分 VII

# 摄像机标定和 3D 重构

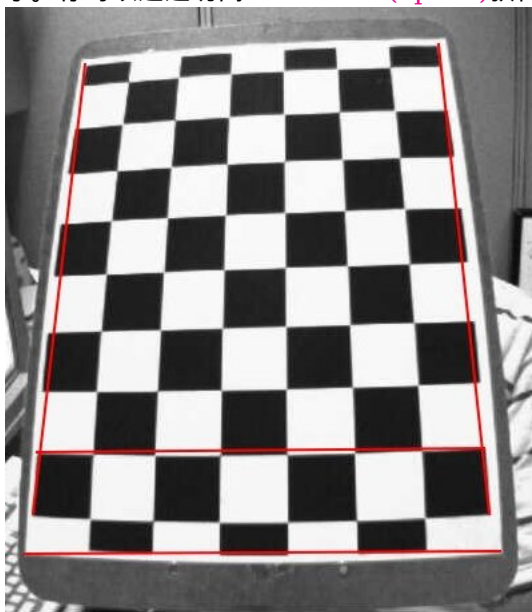
## 42 摄像机标定

### 目标

- 学习摄像机畸变以及摄像机的内部参数和外部参数
- 学习找到这些参数，对畸变图像进行修复

### 42.1 基础

今天的低价单孔摄像机（照相机）会给图像带来很多畸变。畸变主要有两种：径向畸变和切向畸变。如下图所示，用红色直线将棋盘的两个边标注出来，但是你会发现棋盘的边界并不和红线重合。所有我们认为应该是直线的也都凸出来了。你可以通过访问[Distortion \(optics\)](#)获得更多相关细节。



这种畸变可以通过下面的方程组进行纠正：

$$x_{corrected} = x (1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{corrected} = y (1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

于此相似，另外一个畸变是切向畸变，这是由于透镜与成像平面不可能绝对平行造成的。这种畸变会造成图像中的某些点看上去的位置会比我们认为的位置要近一些。它可以通过下列方程组进行校正：

$$x_{corrected} = x + [2p_1xy + p_2(r^2 + 2x^2)]$$

$$y_{corrected} = y + [2p_1xy + p_2(r^2 + 2x^2)]$$

简单来说，如果我们想对畸变的图像进行校正就必须找到五个造成畸变的系数：

$$Distortion\ coefficients = (k_1, k_2, p_1, p_2, k_3)$$

除此之外，我们还需要再找到一些信息，比如摄像机的内部和外部参数。内部参数是摄像机特异的。它包括的信息有焦距 ( $f_x, f_y$ )，光学中心 ( $c_x, c_y$ ) 等。这也被称为摄像机矩阵。它完全取决于摄像机自身，只需要计算一次，以后就可以已知使用了。可以用下面的 3x3 的矩阵表示：

$$camera\ matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

外部参数与旋转和变换向量相对应，它可以将 3D 点的坐标转换到坐标系中。

在 3D 相关应用中，必须要先校正这些畸变。为了找到这些参数，我们必须提供一些包含明显图案模式的样本图片（比如说棋盘）。我们可以在上面找到一些特殊点（如棋盘的四个角点）。我们起到这些特殊点在图片中的位置以及它们的真是位置。有了这些信息，我们就可以使用数学方法求解畸变系数。这就是整个故事的摘要了。为了得到更好的结果，我们至少需要 10 个这样的图案模式。

## 42.2 代码

如上所述，我们至少需要 10 图案模式来进行摄像机标定。OpenCV 自带了一些棋盘图像（/sample/cpp/left001.jpg--left14.jpg），所以我们可以使用它们。为了便于理解，我们可以认为仅有一张棋盘图像。重要的是在进行摄像机标定时我们要输入一组 3D 真实世界中的点以及它们对应 2D 图像中的点。2D 图像的点可以在图像中很容易的找到。（这些点在图像中的位置是棋盘上两个黑色方块相互接触的地方）

那么真实世界中的 3D 的点呢？这些图像来源与静态摄像机和棋盘不同的摆放位置和朝向。所以我们需要知道 (X, Y, Z) 的值。但是为了简单，我们可以说棋盘在 XY 平面是静止的，(所以 Z 总是等于 0) 摄像机在围着棋盘移动。这种假设让我们只需要知道 X, Y 的值就可以了。现在为了求 X, Y 的值，我们只需要传入这些点 (0,0), (1,0), (2,0) ...，它们代表了点的位置。在这个例子中，我们的结果的单位就是棋盘（单个）方块的大小。但是如果我们知道单个方块的大小（加入说 30mm），我们输入的值就可以是 (0,0), (30,0), (60,0) ...，结果的单位就是 mm。（在本例中我们不知道方块的大小，因为不是我们拍的，所以只能用前一种方法了）。

3D 点被称为**对象点**，2D 图像点被称为**图像点**。

#### 42.2.1 设置

为了找到棋盘的图案，我们要使用函数 **cv2.findChessboardCorners()**。我们还需要传入图案的类型，比如说 8x8 的格子或 5x5 的格子等。在本例中我们使用的棋盘是 7x8 的格子。（通常情况下棋盘都是 8x8 或者 7x7）。它会返回角点，如果得到图像的话返回值类型（**Retval**）就会是 **True**。这些角点会按顺序排列（从左到右，从上到下）。

**其他：**这个函数可能不会找出所有图像中应有的图案。所以一个好的方法是编写代码，启动摄像机并在每一帧中检查是否有应有的图案。在我们获得图案之后我们要找到角点并把它们保存成一个列表。在读取下一帧图像之前要设置一定的间隔，这样我们就有足够的时间调整棋盘的方向。继续这个过程直到我们得到足够好的图案。就算是我们举得这个例子，在所有的 14 幅图像中也不知道有几幅是好的。所以我们要读取每一张图像从其中找到好的能用的。

**其他：**除了使用棋盘之外，我们还可以使用环形格子，但是要使用函数 **cv2.findCirclesGrid()** 来找图案。据说使用环形格子只需要很少的图像就可以了。

在找到这些角点之后我们可以使用函数 **cv2.cornerSubPix()** 增加准确度。我们使用函数 **cv2.drawChessboardCorners()** 绘制图案。所有的这些步骤都被包含在下面的代码中了：



```

# -*- coding: utf-8 -*-
"""
Created on Tue Jan 28 08:53:34 2014

@author: duan
"""

import numpy as np
import cv2
import glob

# termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*7,3), np.float32)
objp[:, :2] = np.mgrid[0:7,0:6].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

images = glob.glob('*.jpg')

for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    # Find the chess board corners
    ret, corners = cv2.findChessboardCorners(gray, (7,6),None)

    # If found, add object points, image points (after refining them)
    if ret == True:
        objpoints.append(objp)

        corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
        imgpoints.append(corners2)

        # Draw and display the corners
        img = cv2.drawChessboardCorners(img, (7,6), corners2,ret)
        cv2.imshow('img',img)
        cv2.waitKey(500)

cv2.destroyAllWindows()

```

一副图像和被绘制在上边的图案：



#### 42.2.2 标定

在得到了这些对象点和图像点之后，我们已经准备好来做摄像机标定了。我们要使用的函数是 **cv2.calibrateCamera()**。它会返回摄像机矩阵，畸变系数，旋转和变换向量等。

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[:-1], None, None)
```

#### 42.2.3 畸变校正

现在我们找到我们想要的东西了，我们可以找到一幅图像来对他进行校正了。OpenCV 提供了两种方法，我们都学习一下。不过在那之前我们可以使用从函数 **cv2.getOptimalNewCameraMatrix()** 得到的自由缩放系数对摄像机矩阵进行优化。如果缩放系数  $\alpha = 0$ ，返回的非畸变图像会带有最少量的不想要的像素。它甚至有可能在图像角点去除一些像素。如果  $\alpha = 1$ ，所有的像素都会被返回，还有一些黑图像。它还会返回一个 ROI 图像，我们可以用来对结果进行裁剪。

我们读取一个新的图像 (left2.ipg)

```
img = cv2.imread('left12.jpg')
h, w = img.shape[:2]
newcameramtx, roi=cv2.getOptimalNewCameraMatrix(mtx,dist,(w,h),1,(w,h))
```

**使用 cv2.undistort()** 这是最简单的方法。只需使用这个函数和上边得到的 ROI 对结果进行裁剪。

```
# undistort
dst = cv2.undistort(img, mtx, dist, None, newcameramtx)

# crop the image
x,y,w,h = roi
dst = dst[y:y+h, x:x+w]
cv2.imwrite('calibresult.png',dst)
```

**使用 remapping** 这应该属于“曲线救国”了。首先我们要找到从畸变图像到非畸变图像的映射方程。再使用重映射方程。

```
# undistort
mapx,mapy = cv2.initUndistortRectifyMap(mtx,dist,None,newcameramtx,(w,h),5)
dst = cv2.remap(img,mapx,mapy,cv2.INTER_LINEAR)

# crop the image
x,y,w,h = roi
dst = dst[y:y+h, x:x+w]
cv2.imwrite('calibresult.png',dst)
```

这两中方法给出的结果是相同的。结果如下所示：



你会发现结果图像中所有的边界都变直了。

现在我们可以使用 Numpy 提供写函数（**np.savez**，**np.savetxt** 等）将摄像机矩阵和畸变系数保存以便以后使用。

### 42.3 反向投影误差

我们可以利用反向投影误差对我们找到的参数的准确性进行估计。得到的结果越接近 0 越好。有了内部参数，畸变参数和旋转变换矩阵，我们就可以使用 **cv2.projectPoints()** 将对象点转换到图像点。然后就可以计算变换得到图像与角点检测算法的绝对差了。然后我们计算所有标定图像的误差平均值。

```
mean_error = 0
for i in xrange(len(objpoints)):
    imgpoints2, _ = cv2.projectPoints(objpoints[i], rvecs[i], tvecs[i], mtx, dist)
    error = cv2.norm(imgpoints[i],imgpoints2, cv2.NORM_L2)/len(imgpoints2)
    tot_error += error

print "total error: ", mean_error/len(objpoints)
```

### 更多资源

#### 练习

1. Try camera calibration with circular grid.

## 43 姿势估计

### 目标

- 本节我们要学习使用 **calib3D** 模块在图像中创建 3D 效果

### 43.1 基础

在上一节的摄像机标定中，我们已经得到了摄像机矩阵，畸变系数等。有了这些信息我们就可以估计图像中图案的姿势，比如目标对象是如何摆放，如何旋转等。对一个平面对象来说，我们可以假设  $Z=0$ ，这样问题就转化成摄像机在空间中是如何摆放（然后拍摄）的。所以，如果我们知道对象在空间中的姿势，我们就可以在图像中绘制一些 2D 的线条来产生 3D 的效果。我们来看一下怎么做吧。

我们的问题是，在棋盘的第一个角点绘制 3D 坐标轴（X，Y，Z 轴）。X 轴为蓝色，Y 轴为绿色，Z 轴为红色。在视觉效果上来看，Z 轴应该是垂直与棋盘平面的。

首先我们要加载前面结果中摄像机矩阵和畸变系数。

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 28 11:06:10 2014

@author: duan
"""

import cv2
import numpy as np
import glob

# Load previously saved data
with np.load('B.npz') as X:
    mtx, dist, _, _ = [X[i] for i in ('mtx','dist','rvecs','tvecs')]
```

现在我们来创建一个函数：**draw**，它的参数有棋盘上的角点（使用 **cv2.findChessboardCorners()** 得到）和要绘制的 3D 坐标轴上的点。

```
def draw(img, corners, imgpts):
    corner = tuple(corners[0].ravel())
    img = cv2.line(img, corner, tuple(imgpts[0].ravel()), (255,0,0), 5)
    img = cv2.line(img, corner, tuple(imgpts[1].ravel()), (0,255,0), 5)
    img = cv2.line(img, corner, tuple(imgpts[2].ravel()), (0,0,255), 5)
    return img
```

和前面一样，我们要设置终止条件，对象点（棋盘上的 3D 角点）和坐标轴点。3D 空间中的坐标轴点是为了绘制坐标轴。我们绘制的坐标轴的长度为 3。所以 X 轴从 (0,0,0) 绘制到 (3,0,0)，Y 轴也是。Z 轴从 (0,0,0) 绘制到 (0,0,-3)。负值表示它是朝着（垂直于）摄像机方向。

```
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
objp = np.zeros((6*7,3), np.float32)
objp[:, :2] = np.mgrid[0:7,0:6].T.reshape(-1,2)

axis = np.float32([[3,0,0], [0,3,0], [0,0,-3]]).reshape(-1,3)
```

很通常一样我们需要加载图像。搜寻 7x6 的格子，如果发现，我们就把它优化到亚像素级。然后使用函数: **cv2.solvePnPRansac()** 来计算旋转和变换。但我们有了变换矩阵之后，我们就可以利用它们将这些坐标轴点映射到图像平面中去。简单来说，我们在图像平面上找到了与 3D 空间中的点 (3,0,0), (0,3,0), (0,0,3) 相对应的点。然后我们就可以使用我们的函数 draw() 从图像上的第一个角点开始绘制连接这些点的直线了。搞定 !!!

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 28 11:07:34 2014

@author: duan
"""

for fname in glob.glob('left*.jpg'):
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    ret, corners = cv2.findChessboardCorners(gray, (7,6),None)

    if ret == True:
        corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)

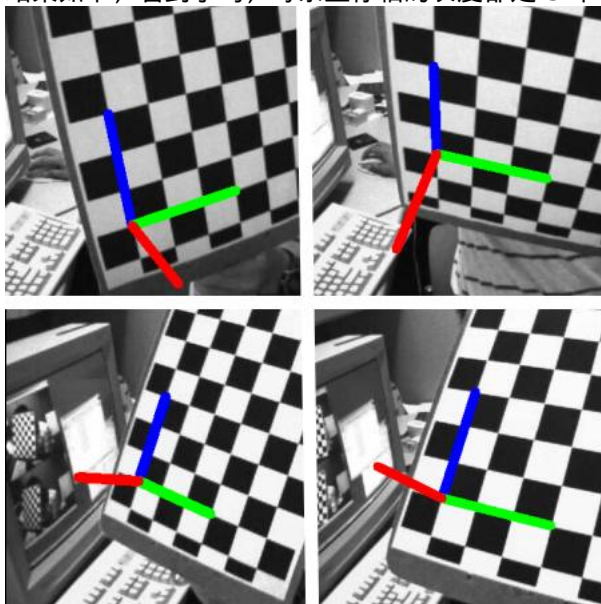
        # Find the rotation and translation vectors.
        rvecs, tvecs, inliers = cv2.solvePnPRansac(objp, corners2, mtx, dist)

        # project 3D points to image plane
        imgpts, jac = cv2.projectPoints(axis, rvecs, tvecs, mtx, dist)

        img = draw(img,corners2,imgpts)
        cv2.imshow('img',img)
        k = cv2.waitKey(0) & 0xff
        if k == 's':
            cv2.imwrite(fname[:6]+'%.png', img)

cv2.destroyAllWindows()
```

结果如下，看到了吗，每条坐标轴的长度都是 3 个格子的长度。



#### 43.1.1 渲染一个立方体

如果你想绘制一个立方体的话要对 **draw()** 函数进行如下修改：  
修改后的 **draw()** 函数：

```
def draw(img, corners, imgpts):
    imgpts = np.int32(imgpts).reshape(-1,2)

    # draw ground floor in green
    img = cv2.drawContours(img, [imgpts[:4]],-1,(0,255,0),-3)

    # draw pillars in blue color
    for i,j in zip(range(4),range(4,8)):
        img = cv2.line(img, tuple(imgpts[i]), tuple(imgpts[j]),(255,3))

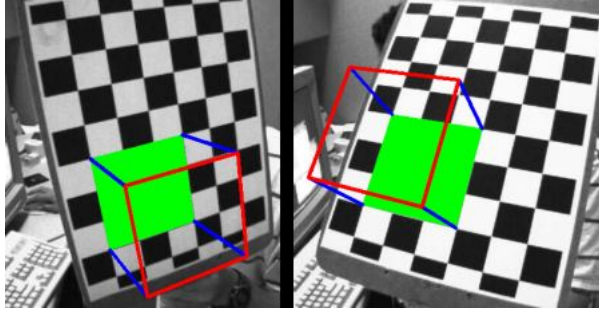
    # draw top layer in red color
    img = cv2.drawContours(img, [imgpts[4:]],-1,(0,0,255),3)

    return img
```

修改后的坐标轴点。它们是 3D 空间中的一个立方体的 8 个角点：

```
axis = np.float32([[0,0,0], [0,3,0], [3,3,0], [3,0,0],
                  [0,0,-3],[0,3,-3],[3,3,-3],[3,0,-3] ])
```

结果如下：



如果你对计算机图形学感兴趣的话，为了增加图像的真实性，你可以使用 OpenGL 来渲染更复杂的图形。（下一个目标）

## 更多资源

## 练习



## 44 对极几何 (Epipolar Geometry)

### 目标

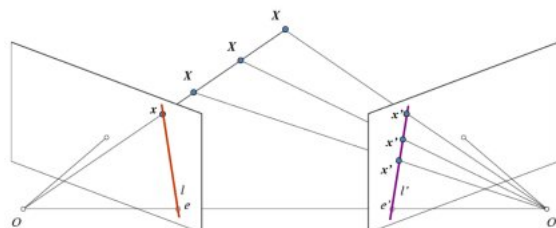
- 本节我们要学习多视角几何基础
- 学习什么是极点，极线，对极约束等

### 44.1 基本概念

在我们使用针孔相机时，我们会丢失大量重要的信息，比如说图像的深度，或者说图像上的点和摄像机的距离，因这是一个从 3D 到 2D 的转换。因此一个重要的问题就产生了，使用这样的摄像机我们能否计算除深度信息呢？答案就是使用多个相机。我们的眼睛就是这样工作的，使用两个摄像机（两个眼睛），这被称为立体视觉。我们来看看 OpenCV 在这方面给我们都提供了什么吧。

（《学习 OpenCV》一书有大量相关知识。）

在进入深度图像之前，我们要先掌握一些多视角几何的基本概念。在本节中我们要处理对极几何。下图为使用两台摄像机同时对一个场景进行拍摄的示意图。



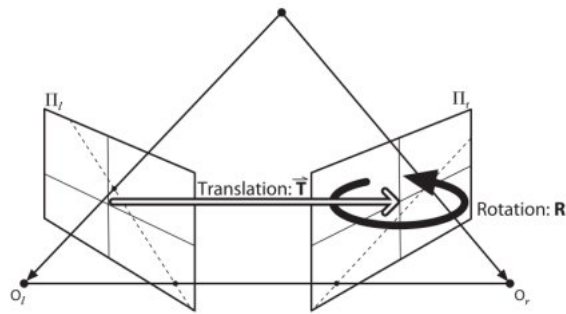
如果只是用一台摄像机我们不可能知道 3D 空间中的  $X$  点到图像平面的距离，因为  $OX$  连线上的每个点投影到图像平面上的点都是相同的。但是如果我们也考虑上右侧图像的话，直线  $OX$  上的点将投影到右侧图像上的不同位置。所以根据这两幅图像，我们就可以使用三角测量计算出 3D 空间中的点到摄像机的距离（深度）。这就是整个思路。

直线  $OX$  上的不同点投射到右侧图像上形成的线  $l'$  被称为与  $x$  点对应的**极线**。也就是说，我们可以在右侧图像中沿着这条极线找到  $x$  点。它可能在这条直线上某个位置（这意味着对两幅图像间匹配特征的二维搜索就转变成了沿着极线的一维搜索。这不仅节省了大量的计算，还允许我们排除许多导致虚假匹配的点）。这被称为**对极约束**。与此相同，所有的点在其他图像中都有与之对应的极线。平面  $XOO'$  被称为**对极平面**。

$O$  和  $O'$  是摄像机的中心。从上面的示意图可以看出，右侧摄像机的中心  $O'$  投影到左侧图像平面的  $e$  点，这个点就被称为**极点**。极点就是摄像机中心连线与图像平面的交点。因此点  $e$  是左侧摄像机的极点。有些情况下，我们可能不会在图像中找到极点，它们可能落在了图像之外（这说明这两个摄像机不能拍摄到彼此）。

所有的极线都要经过极点。所以为了找到极点的位置，我们可以先找到多条极线，这些极线的交点就是极点。

本节我们的重点就是找到极线和极点。为了找到它们，我们还需要两个元素，**本征矩阵 (E)** 和 **基础矩阵 (F)**。本征矩阵包含了物理空间中两个摄像机相关的旋转和平移信息。如下图所示（本图来源自：学习 OpenCV）



基础矩阵  $F$  除了包含  $E$  的信息外还包含了两个摄像机的内参数。由于  $F$  包含了这些内参数，因此它可以它在像素坐标系将两台摄像机关联起来。（如果使用是校正之后的图像并通过除以焦距进行了归一化， $F=E$ ）。简单来说，基础矩阵  $F$  将一副图像中的点映射到另一幅图像中的线（极线）上。这是通过匹配两幅图像上的点来实现的。要计算基础矩阵至少需要 8 个点（使用 8 点算法）。点越多越好，可以使用 RANSAC 算法得到更加稳定的结果。

## 44.2 代码

为了得到基础矩阵我们应该在两幅图像中找到尽量多的匹配点。我们可以使用 SIFT 描述符，FLANN 匹配器和比值检测。

```

# -*- coding: utf-8 -*-
"""
Created on Tue Jan 28 14:20:43 2014

@author: duan
"""

import cv2
import numpy as np
from matplotlib import pyplot as plt

img1 = cv2.imread('myleft.jpg',0) #queryimage # left image
img2 = cv2.imread('myright.jpg',0) #trainimage # right image

sift = cv2.SIFT()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

# FLANN parameters
FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks=50)

flann = cv2.FlannBasedMatcher(index_params,search_params)
matches = flann.knnMatch(des1,des2,k=2)

good = []
pts1 = []
pts2 = []

# ratio test as per Lowe's paper
for i,(m,n) in enumerate(matches):
    if m.distance < 0.8*n.distance:
        good.append(m)
        pts2.append(kp2[m.trainIdx].pt)
        pts1.append(kp1[m.queryIdx].pt)

```

现在得到了一个匹配点列表，我们就可以使用它来计算基础矩阵了。

```

pts1 = np.int32(pts1)
pts2 = np.int32(pts2)
F, mask = cv2.findFundamentalMat(pts1,pts2,cv2.FM_LMEDS)

# We select only inlier points
pts1 = pts1[mask.ravel()==1]
pts2 = pts2[mask.ravel()==1]

```

下一步我们要找到极线。我们会得到一个包含很多线的数组。所以我们要定义一个新的函数将这些线绘制到图像中。

```
def drawlines(img1,img2,lines,pts1,pts2):
    ''' img1 - image on which we draw the epilines for the points in img2
        lines - corresponding epilines '''
    r,c = img1.shape
    img1 = cv2.cvtColor(img1,cv2.COLOR_GRAY2BGR)
    img2 = cv2.cvtColor(img2,cv2.COLOR_GRAY2BGR)
    for r,pt1,pt2 in zip(lines,pts1,pts2):
        color = tuple(np.random.randint(0,255,3).tolist())
        x0,y0 = map(int, [0, -r[2]/r[1] ])
        x1,y1 = map(int, [c, -(r[2]+r[0]*c)/r[1] ])
        img1 = cv2.line(img1, (x0,y0), (x1,y1), color,1)
        img1 = cv2.circle(img1,tuple(pt1),5,color,-1)
        img2 = cv2.circle(img2,tuple(pt2),5,color,-1)
    return img1,img2
```

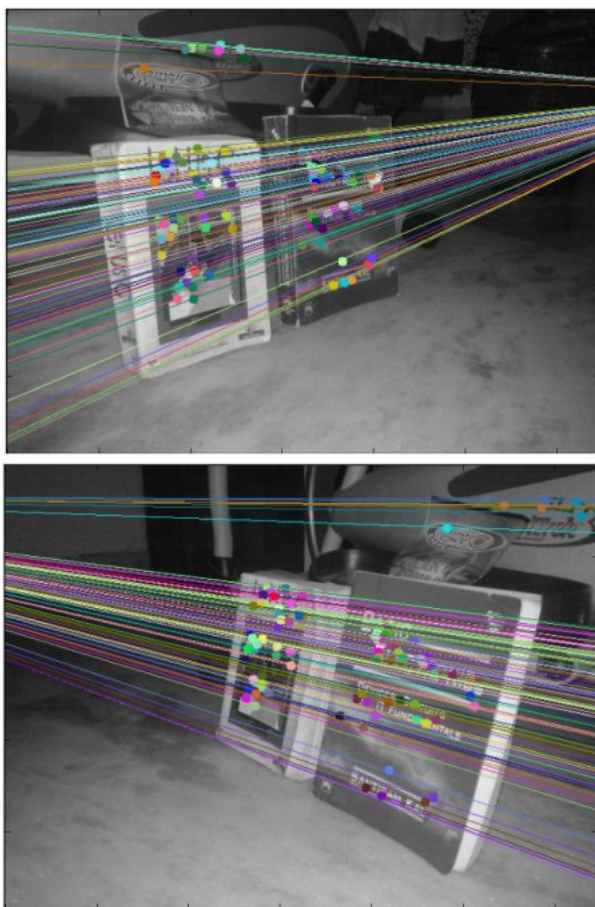
现在我们两幅图像中计算并绘制极线。

```
# Find epilines corresponding to points in right image (second image) and
# drawing its lines on left image
lines1 = cv2.computeCorrespondEpilines(pts2.reshape(-1,1,2), 2,F)
lines1 = lines1.reshape(-1,3)
img5,img6 = drawlines(img1,img2,lines1,pts1,pts2)

# Find epilines corresponding to points in left image (first image) and
# drawing its lines on right image
lines2 = cv2.computeCorrespondEpilines(pts1.reshape(-1,1,2), 1,F)
lines2 = lines2.reshape(-1,3)
img3,img4 = drawlines(img2,img1,lines2,pts2,pts1)

plt.subplot(121),plt.imshow(img5)
plt.subplot(122),plt.imshow(img3)
plt.show()
```

下面是我得到的结果：



从上图可以看出所有的极线都汇聚以图像外的一点，这个点就是极点。  
为了得到更好的结果，我们应该使用分辨率比较高的图像和 non-planar 点。

## 更多资源

### 练习

1. One important topic is the forward movement of camera. Then epipoles will be seen at the same locations in both with epilines emerging from a fixed point. See [this discussion](#).
2. Fundamental Matrix estimation is sensitive to quality of matches, outliers etc. It becomes worse when all selected matches lie on the same plane. Check [this discussion](#).

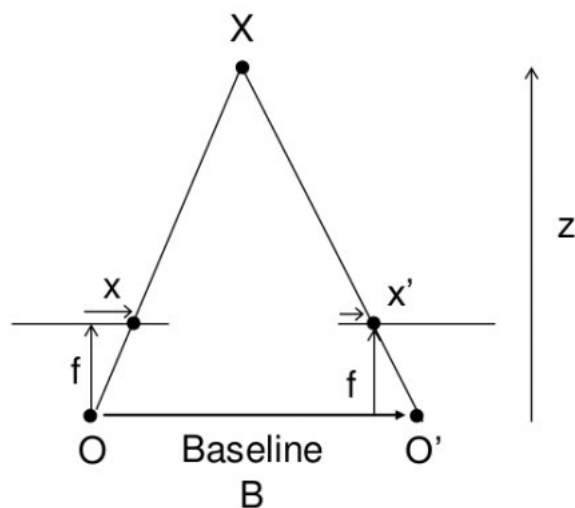
## 45 立体图像中的深度地图

### 目标

- 本节我们要学习为立体图像制作深度地图

### 45.1 基础

在上一节中我们学习了对极约束的基本概念和相关术语。如果同一场景有两幅图像的话我们在直觉上就可以获得图像的深度信息。下面是这幅图和其中的数学公式证明我们的直觉是对的。（图像来源 image courtesy）



The above diagram contains equivalent triangles. Writing their equivalent equations will yield us following result:

$$disparity = x - x' = \frac{Bf}{Z}$$

$x$  和  $x'$  分别是图像中的点到 3D 空间中的点和到摄像机中心的距离。B 是这两个摄像机之间的距离，f 是摄像机的焦距。上边的等式告诉我们点的深度与  $x$  和  $x'$  的差成反比。所以根据这个等式我们就可以得到图像中所有点的深度图。

这样就可以找到两幅图像中的匹配点了。前面我们已经知道了对极约束可以使这个操作更快更准。一旦找到了匹配，就可以计算出 disparity 了。让我们看看在 OpenCV 中怎样做吧。

### 45.2 代码

下面的代码显示了构建深度图的简单过程。

```

# -*- coding: utf-8 -*-
"""
Created on Tue Jan 28 16:03:03 2014

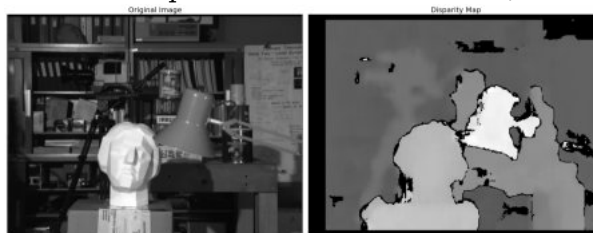
@author: duan
"""
import numpy as np
import cv2
from matplotlib import pyplot as plt

imgL = cv2.imread('tsukuba_l.png',0)
imgR = cv2.imread('tsukuba_r.png',0)

stereo = cv2.createStereoBM(numDisparities=16, blockSize=15)
disparity = stereo.compute(imgL,imgR)
plt.imshow(disparity,'gray')
plt.show()

```

下图左侧为原始图像，右侧为深度图像。如图所示，结果中有很大的噪音。通过调整 numDisparities 和 blockSize 的值，我们会得到更好的结果。



**更多资源**

**练习**

## 部分 VIII

# 机器学习

## 46 K 近邻 (k-Nearest Neighbour)

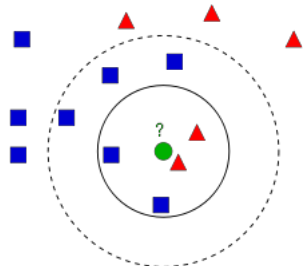
### 46.1 理解 K 近邻

#### 目标

- 本节我们要理解 k 近邻 (kNN) 的基本概念。

#### 原理

kNN 可以说是最简单的监督学习分类器了。想法也很简单，就是找出测试数据在特征空间中的最近邻居。我们将使用下面的图片介绍它。



上图中的对象可以分成两组，蓝色方块和红色三角。每一组也可以称为一个**类**。我们可以把所有的这些对象看成是一个城镇中房子，而所有的房子分别属于蓝色和红色家族，而这个城镇就是所谓的特征空间。（你可以把一个特征空间看成是所有点的投影所在的空间。例如在一个 2D 的坐标空间中，每个数据都两个特征 x 坐标和 y 坐标，你可以在 2D 坐标空间中表示这些数据。如果每个数据都有 3 个特征呢，我们就需要一个 3D 空间。N 个特征就需要 N 维空间，这个 N 维空间就是特征空间。在上图中，我们可以认为是具有两个特征色 2D 空间）。

现在城镇中来了一个新人，他的新房子用绿色圆盘表示。我们要根据他房子的位置把他归为蓝色家族或红色家族。我们把这过程成为**分类**。我们应该怎么做呢？因为我们正在学习看 kNN，那我们就使用一下这个算法吧。

一个方法就是查看他最近的邻居属于那个家族，从图像中我们知道最近的是红色三角家族。所以他被分到红色家族。这种方法被称为简单**近邻**，因为分类仅仅决定与它最近的邻居。

但是这里还有一个问题。红色三角可能是最近的，但如果他周围还有很多蓝色方块怎么办呢？此时蓝色方块对局部的影响应该大于红色三角。所以仅仅检测最近的一个邻居是不足的。所以我们检测 k 个最近邻居。谁在这 k 个邻居中占据多数，那新的成员就属于谁那一类。如果 k 等于 3，也就是在上面图



像中检测 3 个最近的邻居。他有两个红的和一个蓝的邻居，所以他还是属于红色家族。但是如果  $k$  等于 7 呢？他有 5 个蓝色和 2 个红色邻居，现在他就会被分到蓝色家族了。 $k$  的取值对结果影响非常大。更有趣的是，如果  $k$  等于 4 呢？两个红两个蓝。这是一个死结。所以  $k$  的取值最好为奇数。这中根据  $k$  个最近邻居进行分类的方法被称为 **kNN**。

在 kNN 中我们考虑了  $k$  个最近邻居，但是我们给了这些邻居相等的权重，这样做公平吗？以  $k$  等于 4 为例，我们说她是一个死结。但是两个红色三角比两个蓝色方块距离新成员更近一些。所以他更应该被分为红色家族。那用数学应该如何表示呢？我们要根据每个房子与新房子的距离对每个房子赋予不同的权重。距离近的具有更高的权重，距离远的权重更低。然后我们根据两个家族的权重和来判断新房子的归属，谁的权重大就属于谁。这被称为**修改过的 kNN**。

那这里面些是重要的呢？

- 我们需要整个城镇中每个房子的信息。因为我们要测量新来者到所有现存房子的距离，并在其中找到最近的。如果那里有很多房子，就要占用很大的内存和更多的计算时间。
- 训练和处理几乎不需要时间。

现在我们看看 OpenCV 中的 kNN。

#### 46.1.1 OpenCV 中的 kNN

我们这里来举一个简单的例子，和上面一样有两个类。下一节我们会有一个更好的例子。

这里我们将红色家族标记为 **Class-0**，蓝色家族标记为 **Class-1**。还要再创建 25 个训练数据，把它们非别标记为 Class-0 或者 Class-1。Numpy 中随机数产生器可以帮助我们完成这个任务。

然后借助 Matplotlib 将这些点绘制出来。红色家族显示为红色三角蓝色家族显示为蓝色方块。

```

# -*- coding: utf-8 -*-
"""
Created on Tue Jan 28 18:00:18 2014

@author: duan
"""

import cv2
import numpy as np
import matplotlib.pyplot as plt

# Feature set containing (x,y) values of 25 known/training data
trainData = np.random.randint(0,100,(25,2)).astype(np.float32)

# Labels each one either Red or Blue with numbers 0 and 1
responses = np.random.randint(0,2,(25,1)).astype(np.float32)

# Take Red families and plot them
red = trainData[responses.ravel()==0]
plt.scatter(red[:,0],red[:,1],80,'r','^')

# Take Blue families and plot them
blue = trainData[responses.ravel()==1]
plt.scatter(blue[:,0],blue[:,1],80,'b','s')

plt.show()

```

你可能会得到一个与上面类似的图形，但不会完全一样，因为你使用了随机数产生器，每次你运行代码都会得到不同的结果。

下面就是 kNN 算法分类器的初始化，我们要传入一个训练数据集，以及与训练数据对应的分类来训练 kNN 分类器（构建搜索树）。

最后要使用 OpenCV 中的 kNN 分类器，我们给它一个测试数据，让它来进行分类。在使用 kNN 之前，我们应该对测试数据有所了解。我们的数据应该是大小为数据数目乘以特征数目的浮点性数组。然后我们就可以通过计算找到测试数据最近的邻居了。我们可以设置返回的最近邻居的数目。返回值包括：

1. 由 kNN 算法计算得到的测试数据的类别标志（0 或 1）。如果你想使用最近邻算法，只需要将 k 设置为 1，k 就是最近邻的数目。
2. k 个最近邻居的类别标志。
3. 每个最近邻居到测试数据的距离。

让我们看看它是如何工作的。测试数据被标记为绿色。

```

newcomer = np.random.randint(0,100,(1,2)).astype(np.float32)
plt.scatter(newcomer[:,0],newcomer[:,1],80,'g','o')

knn = cv2.KNearest()
knn.train(trainData,responses)
ret, results, neighbours ,dist = knn.find_nearest(newcomer, 3)

print "result: ", results,"\n"
print "neighbours: ", neighbours,"\n"
print "distance: ", dist

plt.show()

```

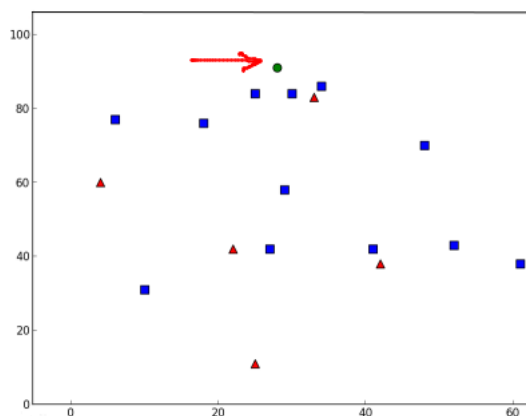
下面是我得到的结果:

```

result: [[ 1.]]
neighbours: [[ 1.  1.  1.]]
distance: [[ 53.  58.  61.]]

```

这说明我们的测试数据有 3 个邻居，他们都是蓝色，所以它被分为蓝色家族。结果很明显，如下图所示：



如果有大量的数据要进行测试，可以直接传入一个数组。对应的结果同样也是数组。

```

# 10 new comers
newcomers = np.random.randint(0,100,(10,2)).astype(np.float32)
ret, results,neighbours,dist = knn.find_nearest(newcomer, 3)
# The results also will contain 10 labels.

```

### 更多资源

1. [NPTEL notes on Pattern Recognition, Chapter 11](#)

### 练习

## 46.2 使用 kNN 对手写数字 OCR

### 目标

- 要根据我们掌握的 kNN 知识创建一个基本的 OCR 程序
- 使用 OpenCV 自带的手写数字和字母数据测试我们的程序

### 46.2.1 手写数字的 OCR

我们的目的是创建一个可以对手写数字进行识别的程序。为了达到这个目的我们需要训练数据和测试数据。OpenCV 安装包中有一副图片 ( /samples/python2/data/digits.png ) , 其中有 5000 个手写数字 ( 每个数字重复 500 遍 )。每个数字是一个 20x20 的小图。所以第一步就是将这个图像分割成 5000 个不同的数字。我们在将拆分后的每一个数字的图像重排成一行含有 400 个像素点的新图像。这个就是我们的特征集, 所有像素的灰度值。这是我们能创建的最简单的特征集。我们使用每个数字的前 250 个样本做训练数据, 剩余的 250 个做测试数据。让我们先准备一下:

```

# -*- coding: utf-8 -*-
"""
Created on Tue Jan 28 20:20:47 2014

@author: duan
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('digits.png')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

# Now we split the image to 5000 cells, each 20x20 size
cells = [np.hsplit(row,100) for row in np.vsplit(gray,50)]

# Make it into a Numpy array. It size will be (50,100,20,20)
x = np.array(cells)

# Now we prepare train_data and test_data.
train = x[:, :50].reshape(-1,400).astype(np.float32) # Size = (2500,400)
test = x[:, 50:100].reshape(-1,400).astype(np.float32) # Size = (2500,400)

# Create labels for train and test data
k = np.arange(10)
train_labels = np.repeat(k,250)[:,np.newaxis]
test_labels = train_labels.copy()

# Initiate kNN, train the data, then test it with test data for k=1
knn = cv2.KNearest()
knn.train(train,train_labels)
ret,result,neighbours,dist = knn.find_nearest(test,k=5)

# Now we check the accuracy of classification
# For that, compare the result with test_labels and check which are wrong
matches = result==test_labels
correct = np.count_nonzero(matches)
accuracy = correct*100.0/result.size
print accuracy

```

现在最基本的 OCR 程序已经准备好了，这个示例中我们得到的准确率为 91%。改善准确度的一个办法是提供更多的训练数据，尤其是判断错误的那些数字。为了避免每次运行程序都要准备和训练分类器，我们最好把它保留，这样在下次运行是时，只需要从文件中读取这些数据开始进行分类就可以了。Numpy 函数 `np.savetxt`, `np.load` 等可以帮助我们搞定这些。

```
np.savez('knn_data.npz', train=train, train_labels=train_labels)

# Now load the data
with np.load('knn_data.npz') as data:
    print data.files
    train = data['train']
    train_labels = data['train_labels']
```

在我的系统中，占用的空间大概为 4.4M。由于我们现在使用灰度值 (uint8) 作为特征，在保存之前最好先把这些数据转换成 np.uint8 格式，这样就只需要占用 1.1M 的空间。在加载数据时再转会到 float32。

### 46.2.2 英文字母的 OCR

接下来我们来做英文字母的 OCR。和上面做法一样，但是数据和特征集有一些不同。现在 OpenCV 给出的不是图片了，而是一个数据文件 (/samples/cpp/letter-recognition.data)。如果打开它的话，你会发现它有 20000 行，第一样看上去就像是垃圾。实际上每一行的第一列是我们的一个字母标记。接下来的 16 个数字是它的不同特征。这些特征来源于 [UCI Machine Learning Repository](#)。你可以在[此页](#)找到更多相关信息。

有 20000 个样本可以使用，我们取前 10000 个作为训练样本，剩下的 10000 个作为测试样本。我们应在先把字母表转换成 asc 码，因为我们不直接处理字母。

```

# -*- coding: utf-8 -*-
"""
Created on Tue Jan 28 20:21:32 2014

@author: duan
"""

import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the data, converters convert the letter to a number
data= np.loadtxt('letter-recognition.data', dtype= 'float32', delimiter = ',',
                converters= {0: lambda ch: ord(ch)-ord('A')})

# split the data to two, 10000 each for train and test
train, test = np.vsplit(data,2)

# split trainData and testData to features and responses
responses, trainData = np.hsplit(train,[1])
labels, testData = np.hsplit(test,[1])

# Initiate the kNN, classify, measure accuracy.
knn = cv2.KNearest()
knn.train(trainData, responses)
ret, result, neighbours, dist = knn.find_nearest(testData, k=5)

correct = np.count_nonzero(result == labels)
accuracy = correct*100.0/10000
print accuracy

```

准确率达到了 93.22%。同样你可以通过增加训练样本的数量来提高准确率。

**更多资源**

**练习**



## 47 支持向量机

### 47.1 理解 SVM

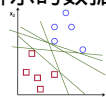
#### 目标

- 对 SVM 有一个直观理解

#### 原理

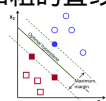
##### 47.1.1 线性数据分割

如下图所示，其中含有两类数据，红的和蓝的。如果是使用 kNN，对于一个测试数据我们要测量它到每一个样本的距离，从而根据最近邻居分类。测量所有的距离需要足够的时间，并且需要大量的内存存储训练样本。但是分类下图所示的数据真的需要占用这么多资源吗？



我们在考虑另外一个想法。我们找到了一条直线， $f(x) = ax_1 + bx_2 + c$ ，它可以将所有的数据分割到两个区域。当我们拿到一个测试数据  $X$  时，我们只需要把它代入  $f(x)$ 。如果  $|f(X)| > 0$ ，它就属于蓝色组，否则就属于红色组。我们把这条线称为**决定边界 (Decision Boundary)**。很简单而且内存使用效率也很高。这种使用一条直线（或者是高位空间种的超平面）上述数据分成两组的方法成为**线性分割**。

从上图中我们看到有很多条直线可以将数据分为蓝红两组，那一条直线是最好的呢？直觉上讲这条直线应该是与两组数据的距离越远越好。为什么呢？因为测试数据可能有噪音影响（真实数据 + 噪声）。这些数据不应该影响分类的准确性。所以这条距离远的直线抗噪声能力也就最强。所以 SVM 要做就是找到一条直线，并使这条直线到（训练样本）各组数据的最短距离最大。下图中加粗的直线经过中心。



要找到决定边界，就需要使用训练数据。我们需要所有的训练数据吗？不，只需要那些靠近边界的数据，如上图中的一个蓝色的圆盘和两个红色的方块。我们叫他们**支持向量**，经过他们的直线叫做**支持平面**。有了这些数据就足以找到决定边界了。我们担心所有的数据。这对于数据简化有帮助。

We need not worry about all the data. It helps in data reduction.

到底发生了什么呢？首先我们找到了分别代表两组数据的超平面。例如，蓝色数据可以用  $\omega^T x + b_0 > 1$  表示，而红色数据可以用  $\omega^T x + b_0 < -1$  表示， $\omega$  叫做**权重向量** ( $\omega = [\omega_1, \omega_2, \dots, \omega_n]$ )， $x$  为**特征向量** ( $x = [x_1, x_2, \dots, x_n]$ )。  $b_0$  被

成为 bias ( 截距 ?)。权重向量决定了决定边界的走向, 而 bias 点决定了它 ( 决定边界 ) 的位置。决定边界被定义为这两个超平面的中间线 ( 平面 ), 表达式为  $\omega^T x + b_0 = 0$ 。从支持向量到决定边界的最短距离为  $distance_{support\ vector} = \frac{1}{\|\omega\|}$ 。边缘长度为这个距离的两倍, 我们需要使这个边缘长度最大。我们要创建一个新的函数  $L(\omega, b_0)$  并使它的值最小:

$$\min_{w, b_0} L(w, b_0) = \frac{1}{2} \|w\|^2 \text{ subject to } t_i(w^T x + b_0) \geq 1 \ \forall i$$

其中  $t_i$  是每一组的标记,  $t_i \in [-1, 1]$ 。

#### 47.1.2 非线性数据分割

想象一下, 如果一组数据不能被一条直线分为两组怎么办? 例如, 在一维空间中 X 类包含的数据点有  $(-3, 3)$ , O 类包含的数据点有  $(-1, 1)$ 。很明显不可能使用线性分割将 X 和 O 分开。但是有一个方法可以帮我们解决这个问题。使用函数  $f(x) = x^2$  对这组数据进行映射, 得到的 X 为 9, O 为 1, 这时就可以使用线性分割了。

或者我们也可以把一维数据转换成二维数据。我们可以使用函数  $f(x) = (x, x^2)$  对数据进行映射。这样 X 就变成了  $(-3, 9)$  和  $(3, 9)$  而 O 就变成了  $(-1, 1)$  和  $(1, 1)$ 。同样可以线性分割, 简单来说就是在低维空间不能线性分割的数据在高维空间很有可能可以线性分割。

通常我们可以将 d 维数据映射到 D 维数据来检测是否可以线性分割 ( $D > d$ )。这种想法可以帮助我们通过对低维输入 ( 特征 ) 空间的计算来获得高维空间的点积。我们可以用下面的例子说明。

假设我们有二维空间的两个点:  $p = (p_1, p_2)$  和  $q = (q_1, q_2)$ 。用  $\phi$  表示映射函数, 它可以按如下方式将二维的点映射到三维空间中:

$$\phi(p) = (p_1^2, p_2^2, \sqrt{2}p_1p_2) \quad \phi(q) = (q_1^2, q_2^2, \sqrt{2}q_1q_2)$$

我们要定义一个核函数  $K(p, q)$ , 它可以用来计算两个点的内积, 如下所示

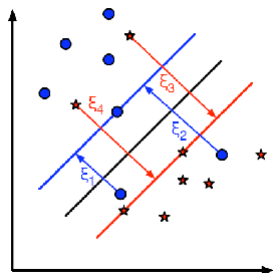
$$\begin{aligned} K(p, q) &= \phi(p) \cdot \phi(q) = \phi(p)^T \phi(q) \\ &= (p_1^2, p_2^2, \sqrt{2}p_1p_2) \cdot (q_1^2, q_2^2, \sqrt{2}q_1q_2) \\ &= p_1q_1 + p_2q_2 + 2p_1q_1p_2q_2 \\ &= (p_1q_1 + p_2q_2)^2 \\ \phi(p) \cdot \phi(q) &= (p \cdot q)^2 \end{aligned}$$

这说明三维空间中的内积可以通过计算二维空间中内积的平方来获得。这可以扩展到更高维的空间。所以根据低维的数据来计算它们的高维特征。在进行完映射后, 我们就得到了一个高维空间数据。

除了上面的这些概念之外, 还有一个问题需要解决, 那就是分类错误。仅仅找到具有最大边缘的决定边界是不够的。我们还需要考虑错误分类带来的误差。有时我们找到的决定边界的边缘可能不是最大的但是错误分类是最少的。所以我们需要对我们的模型进行修正来找到一个更好的决定边界: 最大的边缘, 最小的错误分类。评判标准就被修改为:

$$\min \|w\|^2 + C(\text{distance of misclassified samples to their correct regions})$$

下图显示这个概念。对于训练数据的每一个样本又增加了一个参数  $\xi_i$ 。它表示训练样本到他们所属类（实际所属类）的超平面的距离。对于那些分类正确的样本这个参数为 0，因为它们会落在它们的支持平面上。



现在新的最优化问题就变成了：

$$\min_{w, b_0} L(w, b_0) = \|w\|^2 + C \sum_i \xi_i \text{ subject to } y_i(w^T x_i + b_0) \geq 1 - \xi_i \text{ and } \xi_i \geq 0 \forall i$$

参数  $C$  的取值应该如何选择呢？很明显应该取决于你的训练数据。虽然没有一个统一的答案，但是在选取  $C$  的取值时我们还是应该考虑一下下面的规则：

- 如果  $C$  的取值比较大，错误分类会减少，但是边缘也会减小。其实就是错误分类的代价比较高，惩罚比较大。（在数据噪声很小时我们可以选取较大的  $C$  值。）
- 如果  $C$  的取值比较小，边缘会比较大，但错误分类的数量会升高。其实就是错误分类的代价比较低，惩罚很小。整个优化过程就是为了找到一个具有最大边缘的超平面对数据进行分类。（如果数据噪声比较大时，应该考虑）

## 更多资源

1. [NPTEL notes on Statistical Pattern Recognition, Chapters 25-29.](#)

## 练习

## 47.2 使用 SVM 进行手写数据 OCR

### 目标

本节我们还是要进行手写数据的 OCR，但这次我们使用的是 SVM 而不是 kNN。

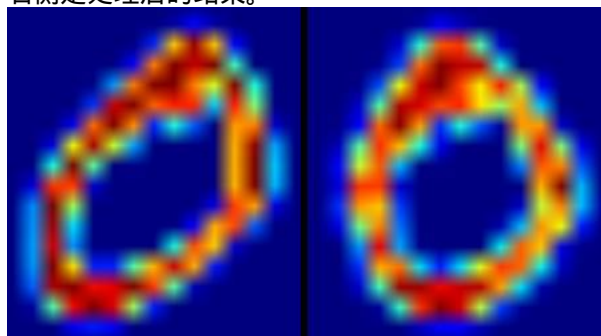
#### 手写数字的 OCR

在 kNN 中我们直接使用像素的灰度值作为特征向量。这次我们要使用方向梯度直方图 **Histogram of Oriented Gradients** (HOG) 作为特征向量。

在计算 HOG 前我们使用图片的二阶矩对其进行抗扭斜 (deskew) 处理。所以我们首先要定义一个函数 **deskew()**，它可以对一个图像进行抗扭斜处理。下面就是 deskew() 函数：

```
def deskew(img):
    m = cv2.moments(img)
    if abs(m['mu02']) < 1e-2:
        return img.copy()
    skew = m['mu11']/m['mu02']
    M = np.float32([[1, skew, -0.5*SZ*skew], [0, 1, 0]])
    img = cv2.warpAffine(img,M,(SZ, SZ),flags=affine_flags)
    return img
```

下图显示了对含有数字 0 的图片进行抗扭斜处理后的效果。左侧是原始图像，右侧是处理后的结果。



接下来我们要计算图像的 HOG 描述符，创建一个函数 **hog()**。为此我们计算图像 X 方向和 Y 方向的 Sobel 导数。然后计算得到每个像素的梯度的方向和大小。把这个梯度转换成 16 位的整数。将图像分为 4 个小的方块，对每一个小方块计算它们的朝向直方图 (16 个 bin)，使用梯度的大小做权重。这样每一个小方块都会得到一个含有 16 个成员的向量。4 个小方块的 4 个向量就组成了这个图像的特征向量 (包含 64 个成员)。这就是我们要训练数据的特征向量。

```

def hog(img):
    gx = cv2.Sobel(img, cv2.CV_32F, 1, 0)
    gy = cv2.Sobel(img, cv2.CV_32F, 0, 1)
    mag, ang = cv2.cartToPolar(gx, gy)
    bins = np.int32(bin_n*ang/(2*np.pi))    # quantizing binvalues in (0...16)
    bin_cells = bins[:10,:10], bins[10:,:10], bins[:10,10:], bins[10:,10:]
    mag_cells = mag[:10,:10], mag[10:,:10], mag[:10,10:], mag[10:,10:]
    hists = [np.bincount(b.ravel(), m.ravel(), bin_n) for b, m in zip(bin_cells, mag_cells)]
    hist = np.hstack(hists)         # hist is a 64 bit vector
    return hist

```

最后，和前面一样，我们将大图分割成小图。使用每个数字的前 250 个作为训练数据，后 250 个作为测试数据。全部代码如下所示：

```

# -*- coding: utf-8 -*-
"""
Created on Wed Jan 29 11:51:59 2014

@author: duan
"""
import cv2
import numpy as np

SZ=20
bin_n = 16 # Number of bins

svm_params = dict( kernel_type = cv2.SVM_LINEAR,
                    svm_type = cv2.SVM_C_SVC,
                    C=2.67, gamma=5.383 )

affine_flags = cv2.WARP_INVERSE_MAP|cv2.INTER_LINEAR

def deskew(img):
    m = cv2.moments(img)
    if abs(m['mu02']) < 1e-2:
        return img.copy()
    skew = m['mu11']/m['mu02']
    M = np.float32([[1, skew, -0.5*SZ*skew], [0, 1, 0]])
    img = cv2.warpAffine(img,M,(SZ, SZ),flags=affine_flags)
    return img

def hog(img):
    gx = cv2.Sobel(img, cv2.CV_32F, 1, 0)
    gy = cv2.Sobel(img, cv2.CV_32F, 0, 1)
    mag, ang = cv2.cartToPolar(gx, gy)
    bins = np.int32(bin_n*ang/(2*np.pi)) # quantizing binvalues in (0..16)
    bin_cells = bins[:10,:10], bins[10:,:10], bins[:,10:10], bins[:,10:]
    mag_cells = mag[:10,:10], mag[10:,:10], mag[:,10:10], mag[:,10:]
    hists = [np.bincount(b.ravel(), m.ravel(), bin_n) for b, m in zip(bin_cells, mag_cells)]
    hist = np.hstack(hists) # hist is a 64 bit vector
    return hist

img = cv2.imread('digits.png',0)

cells = [np.hsplit(row,100) for row in np.vsplit(img,50)]

# First half is trainData, remaining is testData
train_cells = [ i[:50] for i in cells ]
test_cells = [ i[50:] for i in cells]

##### Now training #####

deskewed = [map(deskew,row) for row in train_cells]
hogdata = [map(hog,row) for row in deskewed]
trainData = np.float32(hogdata).reshape(-1,64)
responses = np.float32(np.repeat(np.arange(10),250)[: ,np.newaxis])

svm = cv2.SVM()
svm.train(trainData,responses, params=svm_params)
svm.save('svm_data.dat')

##### Now testing #####

deskewed = [map(deskew,row) for row in test_cells]
hogdata = [map(hog,row) for row in deskewed]
testData = np.float32(hogdata).reshape(-1,bin_n*4)
result = svm.predict_all(testData)

##### Check Accuracy #####
mask = result==responses
correct = np.count_nonzero(mask)
print correct*100.0/result.size

```

准确率达到 94%。你可以尝试一下不同的参数值，看看能不能达到更高的准确率。或者也可以读一下这个领域的文章并用代码实现它。

### 更多资源

1. [Histograms of Oriented Gradients Video](#)

### 练习

1. OpenCV 的示例中包含一个 `digit.py`，它对上面的方法做了一点修改并能得到一个更好的结果。其中还有参考文献。玩一下吧！

## 48 K 值聚类

### 48.1 理解 K 值聚类

#### 目标

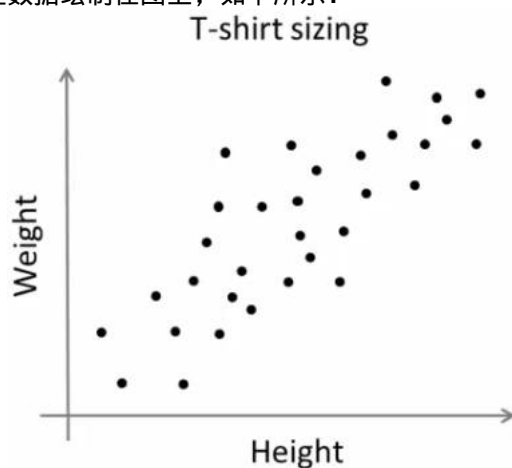
- 本节我们要学习 K 值聚类的概念以及它是如何工作的。

#### 原理

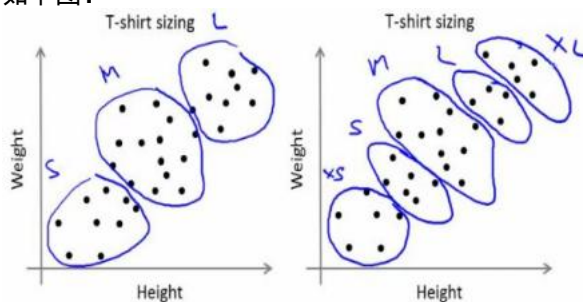
我将用一个最常用的例子来给大家介绍 K 值聚类。

#### 48.1.1 T 恤大小问题

话说有一个公司要生产一批新的 T 恤。很明显他们要生产不同大小的 T 恤来满足不同顾客的需求。所以这个公司收集了很多人的身高和体重信息，并把这些数据绘制在图上，如下所示：



肯定不能把每个大小的 T 恤都生产出来，所以他们把所有的人分为三组：小，中，大，这三组要覆盖所有的人。我们可以使用 K 值聚类的方法将所有人分为 3 组，这个算法可以找到一个最好的分法，并能覆盖所有人。如果不能覆盖全部人的话，公司就只能把这些入分为更多的组，可能是 4 个或 5 个甚至更多。如下图：

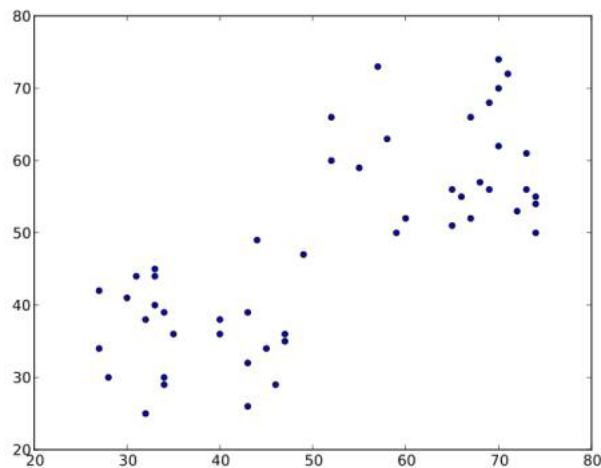




### 48.1.2 它是如何工作的？

这个算法是一个迭代过程，我们会借助图片逐步介绍它。

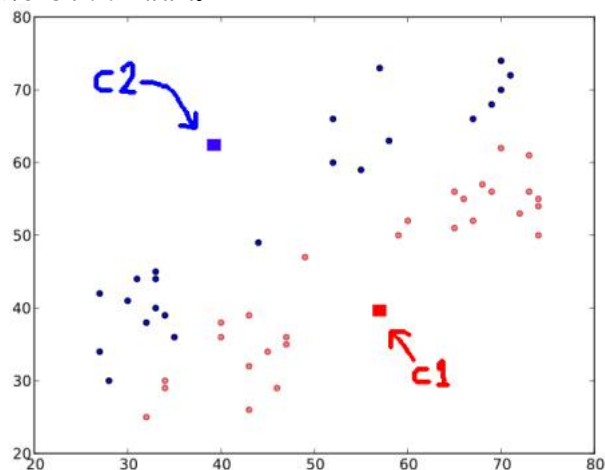
考虑下面这组数据（你也可以把它当成 T 恤问题），我们需要把他们分成两组。



**第一步：**随机选取两个重心点， $C_1$  和  $C_2$ （有时可以选取数据中的两个点作为起始重心）。

**第二步：**计算每个点到这两个重心点的距离，如果距离  $C_1$  比较近就标记为 0，如果距离  $C_2$  比较近就标记为 1。（如果有更多的重心点，可以标记为“2”，“3”等）

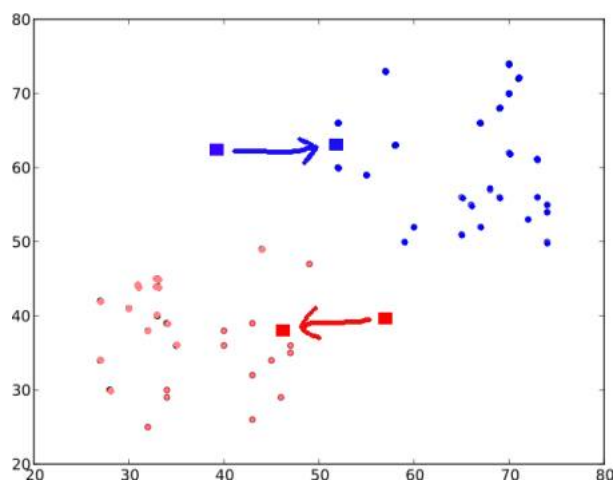
在我们的例子中我们把属于 0 的标记为红色，属于 1 的标记为蓝色。我们就会得到下面这幅图。



**第三步：**重新计算所有蓝色点的重心，和所有红色点的重心，并以这两个点更新重心点的位置。（图片只是为了演示说明而已，并不代表实际数据）

重复步骤 2，更新所有的点标记。

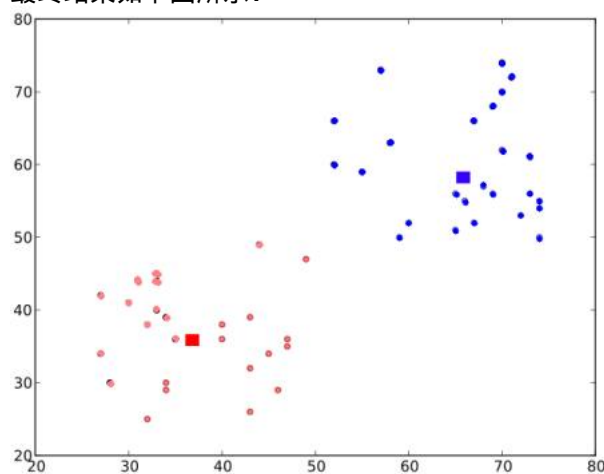
我们就会得到下面的图：



继续迭代步骤 2 和 3，直到两个重心点的位置稳定下来。（当然也可以通过设置迭代次数，或者设置重心移动距离的阈值来终止迭代。）。此时这些点到它们相应重心的距离之和最小。简单来说， $C_1$  到红色点的距离与  $C_2$  到蓝色点的距离之和最小。

$$\text{minimize } \left[ J = \sum_{\text{All Red points}} \text{distance}(C_1, \text{Red\_Point}) + \sum_{\text{All Blue Points}} \text{distance}(C_2, \text{Blue\_Point}) \right]$$

最终结果如下图所示：



这就是对 K 值聚类的一个直观解释。要想知道更多细节和数据解释，你应该读一本关于机器学习的教科书或者参考更多资源中的链接。这只是 K 值聚类的基础。现在对这个算法有很多改进，比如：如何选取好的起始重心点，怎样加速迭代过程等。

### 更多资源

1. [Machine Learning Course, Video lectures by Prof. Andrew Ng](#)  
(Some of the images are taken from this)

练习

## 48.2 OpenCV 中的 K 值聚类

### 目标

- 学习使用 OpenCV 中的函数 **cv2.kmeans()** 对数据进行分类

### 48.2.1 理解函数的参数

#### 输入参数

1. **samples**: 应该是 np.float32 类型的数据，每个特征应该放在一列。
2. **nclusters(K)**: 聚类的最终数目。
3. **criteria**: 终止迭代的条件。当条件满足时，算法的迭代终止。它应该是一个含有 3 个成员的元组，它们是 ( typw, max\_iter, epsilon ):
  - **type** 终止的类型：有如下三种选择：
    - **cv2.TERM\_CRITERIA\_EPS** 只有精确度 epsilon 满足是停止迭代。
    - **cv2.TERM\_CRITERIA\_MAX\_ITER** 当迭代次数超过阈值时停止迭代。
    - **cv2.TERM\_CRITERIA\_EPS + cv2.TERM\_CRITERIA\_MAX\_ITER** 上面的任何一个条件满足时停止迭代。
  - **max\_iter** 表示最大迭代次数。
  - **epsilon** 精确度阈值。
4. **attempts**: 使用不同的起始标记来执行算法的次数。算法会返回紧密度最好的标记。紧密度也会作为输出被返回。
5. **flags**: 用来设置如何选择起始重心。通常我们有两个选择：**cv2.KMEANS\_PP\_CENTERS** 和 **cv2.KMEANS\_RANDOM\_CENTERS**。

#### 输出参数

1. **compactness**: 紧密度，返回每个点到相应重心的距离的平方和。
2. **labels**: 标志数组 ( 与上一节提到的代码相同 )，每个成员被标记为 0, 1 等
3. **centers**: 由聚类的中心组成的数组。

现在我们用 3 个例子来演示如何使用 K 值聚类。

### 48.2.2 仅有一个特征的数据

假设我们有一组数据，每个数据只有一个特征（1 维）。例如前面的 T 恤问题，我们只使用人们的身高来决定 T 恤的大小。

我们先来产生一些随机数据，并使用 Matplotlib 将它们绘制出来。

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan 29 18:28:26 2014

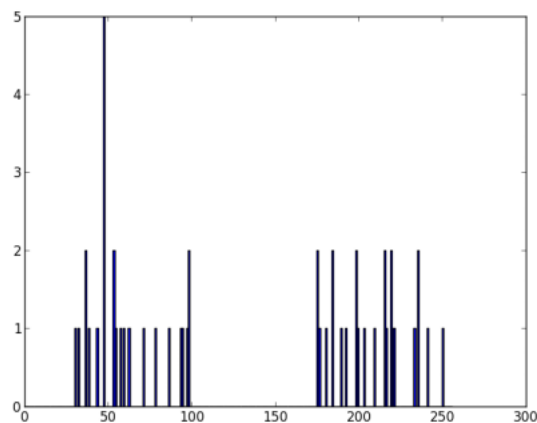
@author: duan
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt

x = np.random.randint(25,100,25)
y = np.random.randint(175,255,25)
z = np.hstack((x,y))
z = z.reshape((50,1))
z = np.float32(z)
plt.hist(z,256,[0,256]),plt.show()
```

现在有一个长度为 50，取值范围为 0 到 255 的向量  $z$ 。我已经将向量  $z$  进行了重排，将它变成了一个列向量。当每个数据含有多个特征是这会很有用。然后我们数据类型转换成 `np.float32`。

我们得到下图：



现在我们使用 KMeans 函数。在这之前我们应该首先设置好终止条件。我的终止条件是：算法执行 10 次迭代或者精确度  $\epsilon = 1.0$ 。

```
# Define criteria = ( type, max_iter = 10 , epsilon = 1.0 )
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)

# Set flags (Just to avoid line break in the code)
flags = cv2.KMEANS_RANDOM_CENTERS

# Apply KMeans
compactness, labels, centers = cv2.kmeans(z, 2, None, criteria, 10, flags)
```

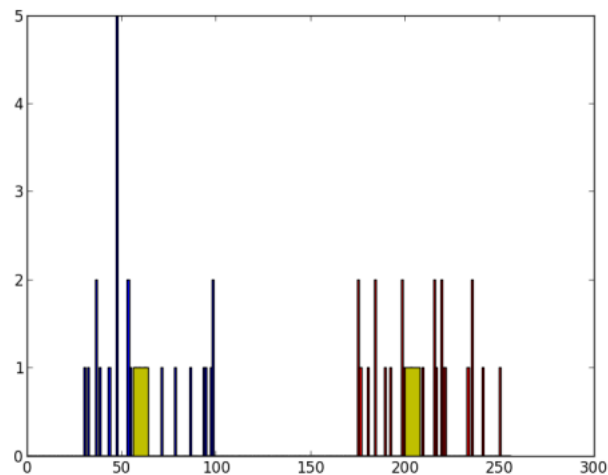
返回值有紧密度 ( compactness ) , 标志和中心。在本例中我的到的中心是 60 和 207。标志的数目与测试数据的多少是相同的, 每个数据都会被标记上“0”, “1”等。这取决与它们的中心是什么。现在我们可以根据它们的标志将把数据分两组。

```
A = z[labels==0]
B = z[labels==1]
```

现在将 A 组数用红色表示, 将 B 组数据用蓝色表示, 重心用黄色表示。

```
# Now plot 'A' in red, 'B' in blue, 'centers' in yellow
plt.hist(A, 256, [0, 256], color = 'r')
plt.hist(B, 256, [0, 256], color = 'b')
plt.hist(centers, 32, [0, 256], color = 'y')
plt.show()
```

下面就是结果：



含有多个特征的数据

在前面的 T 恤例子中我们只考虑了身高，现在我们也把体重考虑进去，也就是两个特征。

在前一节我们的数据是一个单列向量。每一个特征被排列成一列，每一行对应一个测试样本。

在本例中我们的测试数据适应 50x2 的向量，其中包含 50 个人的身高和体重。第一列对应与身高，第二列对应与体重。第一行包含两个元素，第一个是第一个人的身高，第二个是第一个人的体重。剩下的行对应与其他人的身高和体重。如下图所示：

Features		
	Height	Weight
Person 1	H1	W1
Person 2	H2	W2
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
Person 50	H50	W50

现在我们来编写代码：

```

# -*- coding: utf-8 -*-
"""
Created on Wed Jan 29 18:29:23 2014

@author: duan
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt

X = np.random.randint(25,50,(25,2))
Y = np.random.randint(60,85,(25,2))
Z = np.vstack((X,Y))

# convert to np.float32
Z = np.float32(Z)

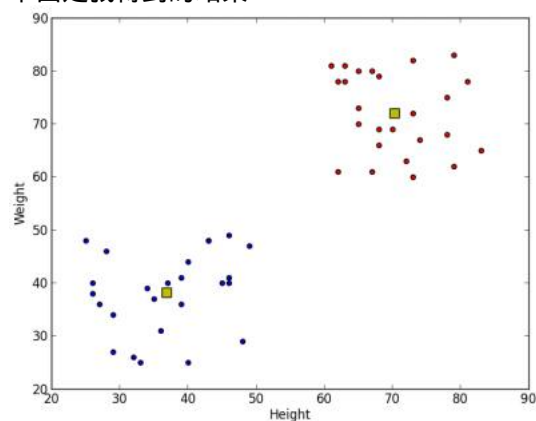
# define criteria and apply kmeans()
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
ret,label,center=cv2.kmeans(Z,2,None,criteria,10,cv2.KMEANS_RANDOM_CENTERS)

# Now separate the data, Note the flatten()
A = Z[label.ravel()==0]
B = Z[label.ravel()==1]

# Plot the data
plt.scatter(A[:,0],A[:,1])
plt.scatter(B[:,0],B[:,1],c = 'r')
plt.scatter(center[:,0],center[:,1],s = 80,c = 'y', marker = 's')
plt.xlabel('Height'),plt.ylabel('Weight')
plt.show()

```

下面是我得到的结果:





### 48.2.3 颜色量化

颜色量化就是减少图片中颜色数目的一个过程。为什么要减少图片中的颜色呢？减少内存消耗！有些设备的资源有限，只能显示很少的颜色。在这种情况下就需要进行颜色量化。我们使用 K 值聚类的方法来进行颜色量化。

没有什么新的知识需要介绍了。现在有 3 个特征：R，G，B。所以我们需要把图片数据变形成  $M \times 3$ （ $M$  是图片中像素点的数目）的向量。聚类完成后，我们用聚类中心值替换与其同组的像素值，这样结果图片就只含有指定数目的颜色了。下面是代码：

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan 29 18:29:46 2014

@author: duan
"""

import numpy as np
import cv2

img = cv2.imread('home.jpg')
Z = img.reshape((-1,3))

# convert to np.float32
Z = np.float32(Z)

# define criteria, number of clusters(K) and apply kmeans()
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 8
ret,label,center=cv2.kmeans(Z,K,None,criteria,10,cv2.KMEANS_RANDOM_CENTERS)

# Now convert back into uint8, and make original image
center = np.uint8(center)
res = center[label.flatten()]
res2 = res.reshape((img.shape))

cv2.imshow('res2',res2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

下面是  $K=8$  的结果：



**更多资源**

**练习**

## 部分 IX

# 计算摄影学

### 49 图像去噪

#### 目标

- 学习使用非局部平均值去噪算法去除图像中的噪音
- 学习函数 `cv2.fastNlMeansDenoising()`, `cv2.fastNlMeansDenoisingColored()` 等

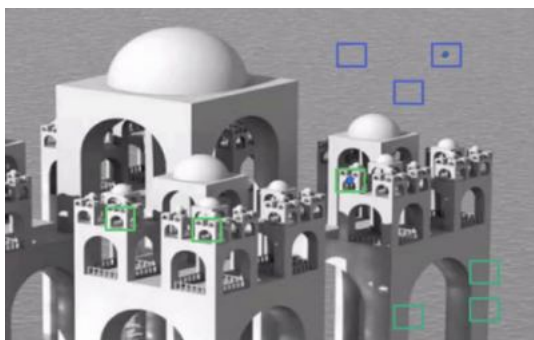
#### 原理

在前面的章节中我们已经学习了很多图像平滑技术，比如高斯平滑，中值平滑等，当噪声比较小时这些技术的效果都是很好的。在这些技术中我们选取像素周围一个小的邻域然后用高斯平均值或者中值平均值取代中心像素。简单来说，像素级别的噪声去除是限制在局部邻域的。

噪声有一个性质。我们认为噪声是平均值为零的随机变量。考虑一个带噪声的像素点， $p = p_0 + n$ ，其中  $p_0$  为像素的真实值， $n$  为这个像素的噪声。我们可以从不同图片中选取大量的相同像素（ $N$ ）然后计算平均值。理想情况下我们会得到  $p = p_0$ 。因为噪声的平均值为 0。

通过简单的设置我们就可以去除这些噪声。将一个静态摄像头固定在一个位置连续拍摄几秒钟。这样我们就会得到足够多的图像帧，或者同一场景的大量图像。写一段代码求解这些帧的平均值（这对你来说应该是小菜一碟）。将最终结果与第一帧图像对比一下。你会发现噪声减小了。不幸的是这种简单的方法对于摄像头和运动场景并不总是适用。大多数情况下我们只有一张导游带有噪音的图像。

想法很简单，我们需要一组相似的图片，通过取平均值的方法可以去除噪音。考虑图像中一个小的窗口（ $5 \times 5$ ），有很大可能图像中的其他区域也存在一个相似的窗口。有时这个相似窗口就在邻域周围。如果我们找到这些相似的窗口并取他们的平均值会怎样呢？对于特定的窗口这样做挺好的。如下图所示。



上图中的蓝色窗口看起来是相似的。绿色窗口看起来也是相似的。所以我们可以选取包含目标像素的一个小窗口，然后在图像中搜索相似的窗口，最后求取所有窗口的平均值，并用这个值取代目标像素的值。这种方法就是非局部平均值去噪。与我们以前学习的平滑技术相比这种算法要消耗更多的时间，但是结果很好。你可以在更多资源中找到更多的细节和在线演示。

对于彩色图像，要先转换到 CIELAB 颜色空间，然后对 L 和 AB 成分分别去噪。

## 49.1 OpenCV 中的图像去噪

OpenCV 提供了这种技术的四个变本。

1. **cv2.fastNlMeansDenoising()** 使用对象为灰度图。
2. **cv2.fastNlMeansDenoisingColored()** 使用对象为彩色图。
3. **cv2.fastNlMeansDenoisingMulti()** 适用于短时间的图像序列（灰度图像）
4. **cv2.fastNlMeansDenoisingColoredMulti()** 适用于短时间的图像序列（彩色图像）

共同参数有：

- **h**：决定过滤器强度。h 值高可以很好的去除噪声，但也会把图像的细节抹去。（取 10 的效果不错）
- **hForColorComponents**：与 h 相同，但使用与彩色图像。（与 h 相同）
- **templateWindowSize**：奇数。（推荐值为 7）
- **searchWindowSize**：奇数。（推荐值为 21）

请查看跟多资源获取这些参数的细节。

这里我们会演示 2 和 3，其余就留给你了。

### 49.1.1 cv2.fastNlMeansDenoisingColored()

和上面提到的一样，它可以被用来去除彩色图像的噪声。（假设是高斯噪声）。下面是示例。

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan 29 21:57:59 2014

@author: duan
"""

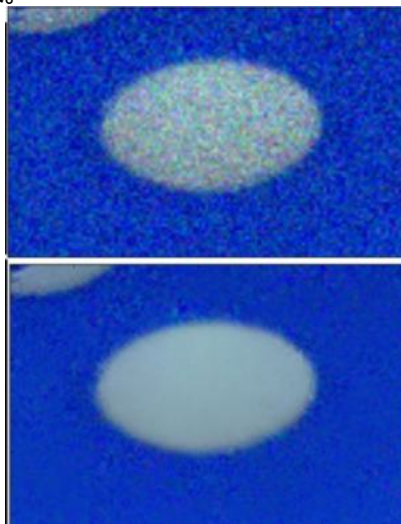
import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('die.png')

dst = cv2.fastNlMeansDenoisingColored(img, None, 10, 10, 7, 21)

plt.subplot(121), plt.imshow(img)
plt.subplot(122), plt.imshow(dst)
plt.show()
```

下面是结果的放大图，我们的输入图像中含有方差为 25 的噪声，下面是结果。



### 49.1.2 cv2.fastNlMeansDenoisingMulti()

现在我们要对一段视频使用这个方法。第一个参数是一个噪声帧的列表。第二个参数 **imgtoDenoiseIndex** 设定那些帧需要去噪，我们可以传入一个帧的索引。第三个参数 **temporaWindowSize** 可以设置用于去噪的相邻

帧的数目，它应该是一个奇数。在这种情况下 **temporalWindowSize** 帧的图像会被用于去噪，中间的帧就是要去噪的帧。例如，我们传入 5 帧图像， $imgToDenoiseIndex = 2$  和  $temporalWindowSize = 3$ 。那么第一帧，第二帧，第三帧图像将被用于第二帧图像的去噪。让我们来看一个例子。

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan 29 21:58:32 2014

@author: duan
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt

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

# create a list of first 5 frames
img = [cap.read()[1] for i in xrange(5)]

# convert all to grayscale
gray = [cv2.cvtColor(i, cv2.COLOR_BGR2GRAY) for i in img]

# convert all to float64
gray = [np.float64(i) for i in gray]

# create a noise of variance 25
noise = np.random.randn(*gray[1].shape)*10

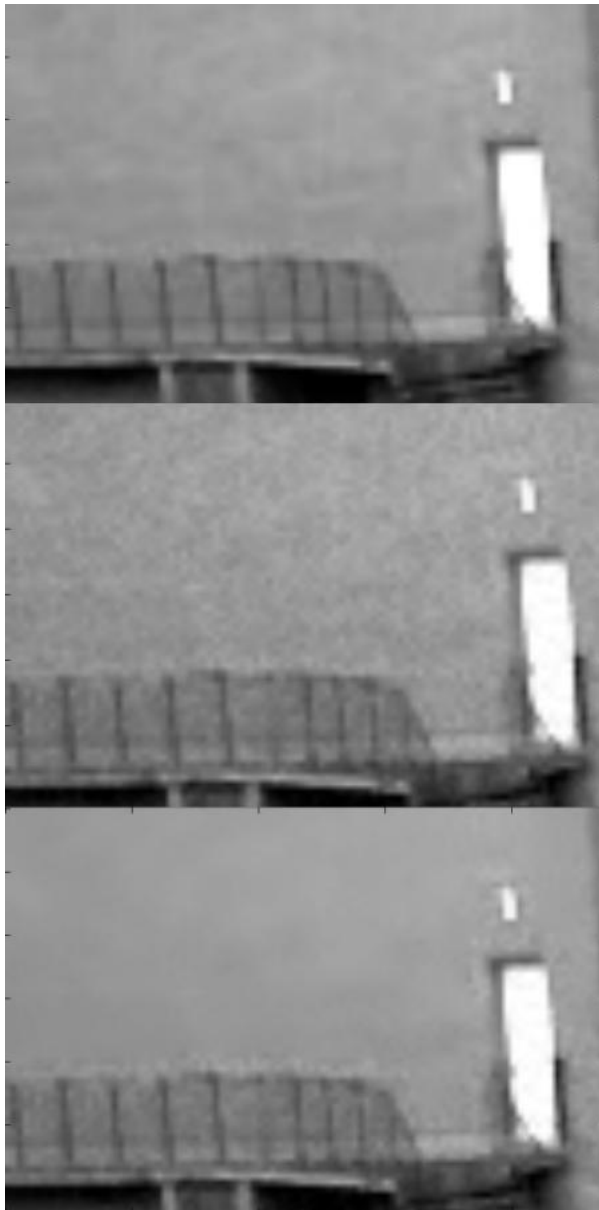
# Add this noise to images
noisy = [i+noise for i in gray]

# Convert back to uint8
noisy = [np.uint8(np.clip(i,0,255)) for i in noisy]

# Denoise 3rd frame considering all the 5 frames
dst = cv2.fastNlMeansDenoisingMulti(noisy, 2, 5, None, 4, 7, 35)

plt.subplot(131),plt.imshow(gray[2],'gray')
plt.subplot(132),plt.imshow(noisy[2],'gray')
plt.subplot(133),plt.imshow(dst,'gray')
plt.show()
```

下图是我得到结果的放大版本。



计算消耗了相当可观的时间。第一张图是原始图像，第二个是带噪音个图像，第三个是去噪音之后的图像。

### 更多资源

1. [http://www.ipol.im/pub/art/2011/bcm\\_nlm/](http://www.ipol.im/pub/art/2011/bcm_nlm/) (It has the details, online demo etc. Highly recommended to visit. Our test image is generated from this link)
2. Online course at coursera (First image taken from here)

## 练习



## 50 图像修补

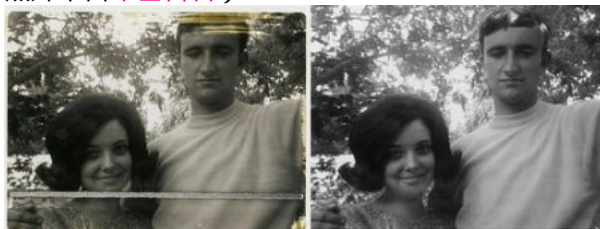
### 目标

本节我们将要学习：

- 使用修补技术去除老照片中小的噪音和划痕
- 使用 OpenCV 中与修补技术相关的函数

### 50.1 基础

在我们每个人的家中可能都会几张退化的老照片，有时候上面不小心在上面弄上了点污渍或者是画了几笔。你有没有想过要修复这些照片呢？我们可以使用笔刷工具轻易在上面涂抹两下，但这没用，你只是用白色笔画取代了黑色笔画。此时我们就要求助于图像修补技术了。这种技术的基本想法很简单：使用坏点周围的像素取代坏点，这样它看起来和周围像素就比较像了。如下图所示（照片来自[维基百科](#)）：



为了实现这个目的，科学家们已经提出了好几种算法，OpenCV 提供了其中的两种。这两种算法都可以通过使用函数 **cv2.inpaint()** 来实施。

第一个算法是根据 Alexandru\_Telea 在 2004 发表的文章实现的。它是基于快速行进算法的。以图像中一个要修补的区域为例。算法从这个区域的边界开始向区域内部慢慢前进，首先填充区域边界像素。它要选取待修补像素周围的一个小的邻域，使用这个邻域内的归一化加权和更新待修复的像素值。权重的选择是非常重要的。对于靠近带修复点的像素点，靠近正常边界像素点和在轮廓上的像素点给予更高的权重。当一个像素被修复之后，使用快速行进算法（FMM）移动到下一个最近的像素。FMM 保证了靠近已知（没有退化的）像素点的坏点先被修复，这与手工启发式操作比较类似。可以通过设置标签参数为 **cv2.INPAINT\_TELEA** 来使用此算法。

第二个算法是根据 Bertalmio, Marcelo, Andrea\_L. Bertozzi, 和 Guillermo\_Sapiro 在 2001 年发表的文章实现的。这个算法是基于流体动力学并使用了偏微分方程。基本原理是启发式的。它首先沿着正常区域的边界向退化区域的前进（因为边界是连续的，所以退化区域非边界与正常区域的边界应该也是连续的）。它通过匹配待修复区域中的梯度向量来延伸等光强线（isophotes，由灰度值相等的点练成的线）。为了实现这个目的，作者是用流体动力学中的一些方法。

完成这一步之后，通过填充颜色来使这个区域内的灰度值变化最小。可以通过设置标签参数为 `cv2.INPAINT_NS` 来使用此算法。

## 50.2 代码

我们要创建一个与输入图像大小相等的掩模图像，将待修复区域的像素设置为 255（其他地方为 0）。所有的操作都很简单。我要修复的图像中有几个黑色笔画。我是使用画笔工具添加的。

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 30 09:25:35 2014

@author: duan
"""

import numpy as np
import cv2

img = cv2.imread('messi_2.jpg')
mask = cv2.imread('mask2.png',0)

dst = cv2.inpaint(img,mask,3,cv2.INPAINT_TELEA)

cv2.imshow('dst',dst)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

结果如下。第一幅图是退化的输入图像，第二幅是掩模图像。第三幅是使用第一个算法的结果，最后一副是使用第二个算法的结果。



## 更多资源

1. Bertalmio, Marcelo, Andrea L. Bertozzi, and Guillermo Sapiro. "Navier-stokes, fluid dynamics, and image and video inpainting." In Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on, vol. 1, pp. I-355. IEEE, 2001.
2. Telea, Alexandru. "An image inpainting technique based on the fast marching method." Journal of graphics tools 9.1 (2004): 23-34.

## 练习

1. OpenCV comes with an interactive sample on inpainting, `samples/python2/inpaint.py`, try it.
2. A few months ago, I watched a video on [Content-Aware Fill](#), an advanced inpainting technique used in Adobe Photoshop. On further search, I was able to find that same technique is already there in GIMP with different name, "Resynthesizer" (You need to install separate plugin). I am sure you will enjoy the technique.

## 部分 X

# 对象检测

## 51 使用 Haar 分类器进行面部检测

### 目标

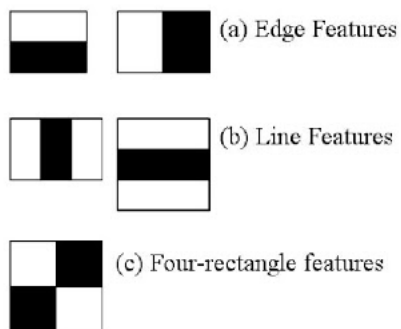
本节我们要学习：

- 以 Haar 特征分类器为基础的面部检测技术
- 将面部检测扩展到眼部检测等。

### 51.1 基础

以 Haar 特征分类器为基础的对象检测技术是一种非常有效的对象检测技术（2001 年 Paul\_Viola 和 Michael\_Jones 提出）。它是基于机器学习的，通过使用大量的正负样本图像训练得到一个 `cascade_function`，最后再用它来做对象检测。

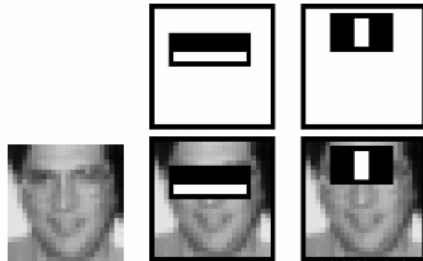
现在我们来学习面部检测。开始时，算法需要大量的正样本图像（面部图像）和负样本图像（不含面部的图像）来训练分类器。我们需要从其中提取特征。下图中的 Haar 特征会被使用。它们就像我们的卷积核。每一个特征是一个值，这个值等于黑色矩形中的像素值之后减去白色矩形中的像素值之和。



使用所有可能的核来计算足够多的特征。（想象一下这需要多少计算量？仅仅是一个 24x24 的窗口就有 160000 个特征）。对于每一个特征的计算我们好需要计算白色和黑色矩形内的像素和。为了解决这个问题，作者引入了积分图像，这可以大大的简化求和运算，对于任何一个区域的像素和只需要对积分图像上的四个像素操作即可。非常漂亮，它可以使运算速度飞快！

但是在我们计算得到的所有的这些特征中，大多数是不相关的。如下图所示。上边一行显示了两个好的特征，第一个特征看上去是对眼部周围区域的描述，因为眼睛总是比鼻子黑一些。第二个特征是描述的是眼睛比鼻梁要黑一些。

但是如果把这两个窗口放到脸颊的话，就一点都不相关。那么我们怎样从超过 160000+ 个特征中选出最好的特征呢？使用 Adaboost。



为了达到这个目的，我们将每一个特征应用于所有的训练图像。对于每一个特征，我们要找到它能够区分出正样本和负样本的最佳阈值。但是很明显，这会产生错误或者错误分类。我们要选取错误率最低的特征，这说明它们是检测面部和非面部图像最好的特征。（这个过程其实不像我们说的这么简单。在开始时每一张图像都具有相同的权重，每一次分类之后，被错分的图像的权重会增大。同样的过程会被再做一遍。然后我们又得到新的错误率和新的权重。重复执行这个过程知道到达要求的准确率或者错误率或者要求数目的特征找到）。

最终的分类器是这些弱分类器的加权和。之所以成为弱分类器是应为只是用这些分类器不足以对图像进行分类，但是与其他的分类器联合起来就是一个很强的分类器了。文章中说 200 个特征就能够提供 95% 的准确度了。他们最后使用 6000 个特征。（从 160000 减到 6000，效果显著呀！）。

现在你有一幅图像，对每一个 24x24 的窗口使用这 6000 个特征来做检查，看它是不是面部。这是不是很低效很耗时呢？的确如此，但作者有更好的解决方法。

在一副图像中大多数区域是非面部区域。所以最好有一个简单的方法来证明这个窗口不是面部区域，如果不是就直接抛弃，不用对它再做处理。而不是集中在研究这个区域是不是面部。按照这种方法我们可以在可能是面部的区域多花点时间。

为了达到这个目的作者提出了级联分类器的概念。不是在一开始就对窗口进行这 6000 个特征测试，将这些特征分成不同组。在不同的分类阶段逐个使用。（通常前面很少的几个阶段使用较少的特征检测）。如果一个窗口第一阶段的检测都过不了就可以直接放弃后面的测试了，如果它通过了就进入第二阶段的检测。如果一个窗口经过了所有的测试，那么这个窗口就被认为是面部区域。这个计划是不是很帅 !!!

作者将 6000 多个特征分为 38 个阶段，前五个阶段的特征数分别为 1, 10, 25, 25 和 50。（上图中的两个特征其实就是从 **Adaboost** 获得的最好特征）。

According to authors, on an average, 10 features out of 6000+ are evaluated per sub-window.

上面是我们对 Viola-Jones 面部检测是如何工作的直观解释。读一下原始文献或者更多资源中非参考文献将会对你有更大帮助。

## 51.2 OpenCV 中的 Haar 级联检测

OpenCV 自带了训练器和检测器。如果你想自己训练一个分类器来检测汽车，飞机等的话，可以使用 OpenCV 构建。其中的细节在这里：[Cascade Classifier Training](#)

现在我们来学习一下如何使用检测器。OpenCV 已经包含了很多已经训练好的分类器，其中包括：面部，眼睛，微笑等。这些 XML 文件保存在 `/opencv/data/haarcascades/` 文件夹中。下面我们将使用 OpenCV 创建一个面部和眼部检测器。

首先我们要加载需要的 XML 分类器。然后以灰度格式加载输入图像或者是视频。

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 30 11:06:23 2014

@author: duan
"""

import numpy as np
import cv2

face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier('haarcascade_eye.xml')

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

现在我们在图像中检测面部。如果检测到面部，它会返回面部所在的矩形区域 `Rect (x,y,w,h)`。一旦我们获得这个位置，我们可以创建一个 ROI 并在其中进行眼部检测。（谁让眼睛长在脸上呢 !!!）

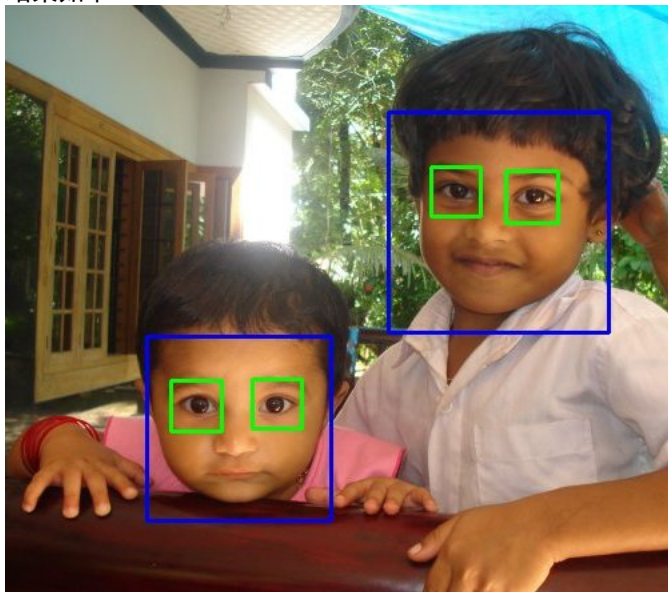
```

#Detects objects of different sizes in the input image.
# The detected objects are returned as a list of rectangles.
#cv2.CascadeClassifier.detectMultiScale(image, scaleFactor, minNeighbors, flags, minSize, maxSize)
#scaleFactor – Parameter specifying how much the image size is reduced at each image
#scale.
#minNeighbors – Parameter specifying how many neighbors each candidate rectangle should
#have to retain it.
#minSize – Minimum possible object size. Objects smaller than that are ignored.
#maxSize – Maximum possible object size. Objects larger than that are ignored.
faces = face_cascade.detectMultiScale(gray, 1.3, 5)
for (x,y,w,h) in faces:
    img = cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]
    eyes = eye_cascade.detectMultiScale(roi_gray)
    for (ex,ey,ew,eh) in eyes:
        cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)

cv2.imshow('img',img)
cv2.waitKey(0)
cv2.destroyAllWindows()

```

结果如下：



## 更多资源

1. Video Lecture on [Face Detection and Tracking](#)
2. An interesting interview regarding Face Detection by [Adam Harvey](#)

## 练习