Table of Contents

OpenCV 中文文档 4.0.0

> 原文：OpenCV documentation 4.0.0
>
> 协议：CC BY-NC-SA 4.0
>
> 欢迎任何人参与和完善：一个人可以走的很快，但是一群人却可以走的更远。

```
OpenCV（开源计算机视觉库）是在 BSD 许可下发布的，因此它可以免费用于学术和商业用途。
它具有 C++，Python 和 Java 接口，支持 Windows，Linux，Mac OS，iOS 和 Android。
OpenCV 专为提高计算效率而设计，专注于实时应用。该库以优化的 C/C++ 编写，可以利用多核处理。
通过 OpenCL 启用，它可以利用底层异构计算平台的硬件加速。

OpenCV 在全球范围内采用，拥有超过 4.7 万用户社区，估计下载量超过 1400 万。
用途范围从交互式艺术，到地雷检查，网上拼接地图或高级机器人。
```

- 在线阅读
- 项目负责人及贡献者：请见各个版本的首页
- ApacheCN 机器学习交流群 629470233
- ApacheCN 学习资源

建议反馈

- 联系项目负责人
  - @Daidai
  - @lyrich
  - @片刻
- 在我们的 apachecn/opencv-doc-zh github 上提 issue.
- 发邮件到 Email: `apachecn@163.com` .
- 在我们的 QQ 群-搜索: 交流方式 中联系群主/管理员即可.

赞助我们

微信	支付宝
	

mkdir books gitbook epub . books/opencv-doc-zh.epub gitbook mobi . books/opencv-doc-zh.mobi gitbook pdf . books/opencv-doc-zh.pdf

OpenCV-Python 教程简介

OpenCV

OpenCV 于 1999 年由 Gary Bradsky 在英特尔创立，第一个版本于 2000 年问世。Vadim Pisarevsky 加入了 Gary Bradsky，负责管理英特尔的俄罗斯软件 OpenCV 团队。2005 年，OpenCV 被用于 Stanley，这辆车赢得了 2005 年美国穿越沙漠 DARPA 机器人挑战大赛。后来，在 Willow Garage 的支持下，在 Gary Bradsky 和 Vadim Pisarevsky 主导下，OpenCV 项目的开发工作变得活跃起来。OpenCV 现在支持与计算机视觉和机器学习相关的众多算法，并且每天都在拓展中。

OpenCV 支持各种编程语言，如 C++，Python，Java 等，可在不同的平台上使用，包括 Windows，Linux，OS X，Android 和 iOS。基于 CUDA 和 OpenCL 的高速 GPU 操作接口也在积极开发中。

OpenCV-Python 是 OpenCV 的 Python API，结合了 OpenCV C++ API 和 Python 语言的最佳特性。

OpenCV-Python

OpenCV-Python 是一个 Python 绑定库，旨在解决计算机视觉问题。

Python 是一种由 Guido van Rossum 开发的通用编程语言，它很快就变得非常流行，主要是因为它的简单性和代码可读性。它使程序员能够用更少的代码表达思想，而不会降低可读性。

与 C/C++ 这类语言相比，Python 的速度更慢。好在，可以使用 C/C++ 轻松的拓展 Python，我们可以在 C/C++ 中编写计算密集型代码，并用 Python 来封装。这给我们带来了两个好处：首先，代码像原始的 C/C++ 代码一样快（因为后台实际上就是 C/C++ 代码在工作），其次，在 Python 中编写代码比在 C/C++ 中更容易。OpenCV-Python 就是 OpenCV C++ 的 Python 封装。

OpenCV-Python 使用了 Numpy，这是一个有着 MATLAB 风格语法，高度优化的用于数值计算的库。所有 OpenCV 数组结构都与 Numpy 数组进行转换。这也使得与使用 Numpy 的其他库（如 SciPy 和 Matplotlib）集成更容易。

OpenCV-Python 教程

OpenCV 引入了一组新的教程，它将指导你学习 OpenCV-Python 中提供的各种功能。本教程主要关注 OpenCV 3.x 版本（尽管大多数内容也适用于 OpenCV 2.x）。

建议先了解 Python 和 Numpy，因为本教程不涉及它们。为了使用 OpenCV-Python 编写优化代码，你必须熟练使用 Numpy。

本教程最初由 Abid Rahman K.在 Alexander Mordvintsev 的指导下作为 Google Summer of Code 2013 计划的一部分启动。

OpenCV 需要你!!!

OpenCV 是一个开源的项目，欢迎所有人对库、文档和教程做出贡献。如果您在本教程中发现任何错误（从小的拼写错误到代码或概念中的严重错误），请在 Github 中 clone OpenCV，并提交一个 pull request 来纠正错误。OpenCV 开发者将会检查你的 pull request，给你反馈，并且（一旦通过审核人员的批准），将更改合并到项目中。然后，你将成为开源贡献者。

随着新模块添加到 OpenCV-Python，本教程将不得不进行扩展。如果你熟悉特定算法，可以编写包含算法基本理论的教程，并能够编写使用样例，请参与到教程的编写中来。

记住，这个项目因我们的共同努力而变得伟大！！！

贡献者

以下是向 OpenCV-Python 提交教程的贡献者列表。

1. Alexander Mordvintsev (GSoC-2013 导师)
2. Abid Rahman K. (GSoC-2013 实习生)

额外资源

1. Python 基本教程 - A Byte of Python
2. Numpy 基本教程
3. Numpy 使用样例
4. OpenCV 文档
5. OpenCV 论坛

在 Ubuntu 系统中安装 OpenCV—Python

目标

在本教程中，我们将学习如何在 Ubuntu 系统中安装 OpenCV-Python。以下步骤针对 Ubuntu 16.04（64 位）和 Ubuntu 14.04（32 位）进行了测试。

在 Ubuntu 中，可以通过两种方式安装 OpenCV-Python：

- 直接使用 Ubuntu 软件库中编译好的二进制文件进行安装
- 从源码编译安装

本章将介绍这两种安装方式。另外，要用到几个第三方库。虽然，OpenCV-Python 仅仅需要 **Numpy**（除了其他依赖库，我们稍后会说），但是，在本教程中，我们还会使用 **Matplotlib** 进行一些简单而美观的绘图（我觉得要比 OpenCV 好很多）。 虽然 Matplotlib 不是必选的，但我们强烈推荐安装。同样的，我们也强烈推荐安装 **IPython** ，一个交互式 Python 终端。

二进制方式安装 OpenCV-Python

当仅用于编程和开发 OpenCV 应用程序时，此方案最为合适。

使用终端中的以下命令安装 python-opencv（以 root 用户身份）。

```
$ sudo apt-get install python-opencv
```

打开 Python IDLE（或 IPython）并在 Python 终端中键入以下命令。

```
import cv2 as cv
print(cv.__version__)
```

如果没有打印任何错误，恭喜！你已成功安装 OpenCV-Python。

这种方法虽然简单，但是有个问题。apt 软件库并非总是含有最新版本的 OpenCV。举例来说，写本教程的时候，apt 软件库中的版本是 2.4.8，而最新的 OpenCV 版本是 3.x。对于 Python API，最新版本始终有着更好的支持和最少的 bug。

因此，如果想要获得最新的代码，最好选择下一种方案，即从源码编译安装。另外，如果你某个时候想要给 OpenCV 贡献代码，你也需要这么做。

从源码中编译 OpenCV

刚开始的时候，从源代码编译可能看起来有点小复杂，但是一旦你成功了，就没有什么复杂的了。

首先，我们将安装一些依赖库。有些是必需的，有些是可选的。如果不需要，可以跳过可选的依赖库。

必需的依赖库

首先，我们需要用 **CMake** 来配置安装，用 **GCC** 来编译，用 **Python-devel** 和 **Numpy** 来构建 Python 绑定，等等。

```
sudo apt-get install cmake
sudo apt-get install python-devel numpy
sudo apt-get install gcc gcc-c++
```

接下来，我们需要 GTK 库支持 GUI 功能，需要 v4l 库支持相机功能，需要 ffmpeg 库和 gstreamer 库支持媒体功能，等等。

```
sudo apt-get install gtk2-devel
sudo apt-get install ffmpeg-devel
sudo apt-get install gstreamer-plugins-base-devel
```

可选的依赖库

上面的依赖库已经足以让你在 Ubuntu 系统上安装 OpenCV。但是，根据你的需求，你可能还要装一些额外的库。下面给出了一个可选依赖库的列表。装或不装，由你决定 :)

OpenCV 自带了 PNG，JPEG，JPEG2000，TIFF，WebP 等图像格式的支持库，但它们可能有点老。如果你想要最新的库，你可以自己安装它们。

```
sudo apt-get install libpng-devel
sudo apt-get install libjpeg-turbo-devel
sudo apt-get install jasper-devel
sudo apt-get install openexr-devel
sudo apt-get install libtiff-devel
sudo apt-get install libwebp-devel
```

下载 OpenCV

从 OpenCV 的GitHub 存储库下载最新的源代码。（如果你想为 OpenCV 做贡献，请这么做。为此，你需要先安装 Git）

```
$ sudo apt-get install git
$ git clone https://github.com/opencv/opencv.git
```

上面的操作将在你的当前目录下创建一个 opencv 文件夹。clone 可能要花一些时间，具体取决于你的网络延迟。

现在，打开一个终端窗口，并进入刚下的“opencv”文件夹。创建一个“build”文件夹，进入“build”。

```
$ mkdir build
$ cd build
```

配置和安装

现在，我们已安装好了所有的依赖库，让我们开始安装 OpenCV。必须使用 CMake 配置安装。它指定了要安装哪些模块，安装路径，要使用的其他库，以及是否需要编译文档和示例等。默认参数已配置好，大部分工作都可以自动化完成。

下面的命令通常用于配置 OpenCV 库编译（在 build 文件夹里执行）：

```
$ cmake ../
```

OpenCV 默认采用“Release”构建类型，安装路径为“/usr/local”。有关 CMake 选项的其他信息，请参阅 OpenCV C++编译指南：

您应该能在 CMake 输出信息中看到这些行（这些意味着已正确安装了 Python）：

```
--   Python 2:
--     Interpreter:                 /usr/bin/python2.7 (ver 2.7.6)
--     Libraries:                   /usr/lib/x86_64-linux-gnu/libpython2.7.so (ver 2.7
--     numpy:                       /usr/lib/python2.7/dist-packages/numpy/core/includ
--     packages path:               lib/python2.7/dist-packages
--
--   Python 3:
--     Interpreter:                 /usr/bin/python3.4 (ver 3.4.3)
--     Libraries:                   /usr/lib/x86_64-linux-gnu/libpython3.4m.so (ver 3.
--     numpy:                       /usr/lib/python3/dist-packages/numpy/core/include
--     packages path:               lib/python3.4/dist-packages
```

现在，使用“make”命令编译文件，使用“make install”来安装。

```
$ make
# sudo make install
```

安装完成后，所有文件都在“/usr/local/”目录。打开终端，并导入 cv2。

```
import cv2 as cv
print(cv.__version__)
```

图像入门

目标

在本次会议中:

- 在这里，你将学习如何读取图像、如何显示图像以及如何将其保存起来
- 你要学习这些函数：**cv.imread()**、**cv.imshow()**、**cv.imwrite()**
- 您还可以选择学习如何使用 Matplotlib 显示图像。

使用 OpenCV

读取图像

使用 **cv.imread()** 函数读取一张图像，图片应该在工作目录中，或者应该提供完整的图像路径。

第二个参数是一个 flag，指定了应该读取图像的方式

- **cv.IMREAD_COLOR**：加载彩色图像，任何图像的透明度都会被忽略，它是默认标志
- **cv.IMREAD_GRAYSCALE**：以灰度模式加载图像
- **cv.IMREAD_UNCHANGED**：加载图像，包括 alpha 通道

Note

- 你可以简单地分别传递整数 1、0 或-1，而不是这三个 flag。

看下面的代码

```
import numpy as np
import cv2 as cv
# 用灰度模式加载图像
img = cv.imread('messi5.jpg', 0)
```

注意

即使图像路径错误，它也不会抛出任何错误，但是 `打印 img` 会给你 `None`

显示图像

用 **cv.imshow()** 函数在窗口中显示图像，窗口自动适应图像的大小。

第一个参数是窗口名，它是一个字符串，第二个参数就是我们的图像。你可以根据需要创建任意数量的窗口，但是窗口名字要不同。

```
cv.imshow('image', img)
cv.waitKey(0)
cv.destroyAllWindows()
```

一个窗口的截图可能看起来像这样 (in Fedora-Gnome machine):

cv.waitKey() 是一个键盘绑定函数，它的参数是以毫秒为单位的时间。该函数为任意键盘事件等待指定毫秒。如果你在这段时间内按下任意键，程序将继续。如果传的是 0，它会一直等待键盘按下。它也可以设置检测特定的击键，例如，按下键 a 等，我们将在下面讨论。

Note

- 除了绑定键盘事件，该函数还会处理许多其他 GUI 事件，因此你必须用它来实际显示图像。

cv.destroyAllWindows() 简单的销毁我们创建的所有窗口。如果你想销毁任意指定窗口，应该使用函数 **cv.destroyWindow()** 参数是确切的窗口名。

Note

有一种特殊情况，你可以先创建一个窗口然后加载图像到该窗口。在这种情况下，你能指定窗口是否可调整大小。它是由这个函数完成的 **cv.namedWindow()**。默认情况下，flag 是 **cv.WINDOW_AUTOSIZE**。但如果你指定了 flag 为 **cv.WINDOW_NORMAL**，你能调整窗口大小。当图像尺寸太大，在窗口中添加跟踪条是很有用的。

看下面的代码：

```
cv.namedWindow('image', cv.WINDOW_NORMAL)
cv.imshow('image',img)
cv.waitKey(0)
cv.destroyAllWindows()
```

保存图像

保存图像，用这个函数 **cv.imwrite()**。

第一个参数是文件名，第二个参数是你要保存的图像。

```
cv.imwrite('messigray.png',img)
```

将该图像用 PNG 格式保存在工作目录。

总结一下

下面的程序以灰度模式读取图像，显示图像，如果你按下 's‘ 会保存和退出图像，或者按下 ESC 退出不保存。

```
import numpy as np
import cv2 as cv

img = cv.imread('messi5.jpg',0)
cv.imshow('image',img)
k = cv.waitKey(0)
if k == 27: # ESC 退出
    cv.destroyAllWindows()
elif k == ord('s'): # 's' 保存退出
    cv.imwrite('messigray.png',img)
    cv.destroyAllWindows()
```

注意

如果你使用的是 64 位机器，你需要修改 `k = cv.waitKey(0)` 像这样： `k = cv.waitKey(0) & 0xFF`

使用 Matplotlib

Matplotlib 是一个 Python 的绘图库，提供了丰富多样的绘图函数。你将在接下来的文章中看到它们。在这里，你将学习如何使用 Matplotlib 来显示图像。你还能用 Matplotlib 缩放图像，保存图像等。

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

img = cv.imread('messi5.jpg',0)
plt.imshow(img, cmap = 'gray', interpolation = 'bicubic')
plt.xticks([]), plt.yticks([]) # 隐藏 X 和 Y 轴的刻度值
plt.show()
```

窗口的屏幕截图是这样的：

参考

Matplotlib 提供了大量的绘图选项。有关更多详情信息，请参阅 Matplotlib 文档。有一些，我们用这种方式将会知道。

注意

彩色图像 OpenCV 用的 BGR 模式，但是 Matplotlib 显示用的 RGB 模式。因此如果图像用 OpenCV 加载，则 Matplotlib 中彩色图像将无法正常显示。更多细节请看练习。

其他资源

1. **Matplotlib Plotting Styles and Features**

练习

1. 当你尝试用 OpenCV 加载图像并用 Matplotlib 来显示，就会出现一些问题。阅读这篇 讨论 并理解它。

视频入门

目标

在本次会议中：

- 学习加载视频、显示视频和保存视频。
- 学习用相机捕捉并显示。
- 你要学习这些函数：**cv.VideoCapture()**，**cv.VideoWriter()**

从相机捕捉视频

通常，我们用相机捕捉直播。OpenCV 为此提供了一个非常简单的接口。我们用相机捕捉一个视频(我用的电脑内置摄像头)，将它转换成灰度视频并显示。仅仅是一个简单的开始。

去获取一个视频，你需要创建一个**VideoCapture**对象。它的参数可以是设备索引或者一个视频文件名。设备索引仅仅是摄像机编号。通常会连接一台摄像机(as in my case)。所以我只传了 0(或者-1)。你可以通过传 1 来选择第二个摄像机，以此类推。之后，你能逐帧捕获。但是最后，不要忘记释放这个 Capture 对象。

```
import numpy as np
import cv2 as cv
cap = cv.VideoCapture(0)
while(True):
    # 一帧一帧捕捉
    ret, frame = cap.read()
    # 我们对帧的操作在这里
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    # 显示返回的每帧
    cv.imshow('frame',gray)
    if cv.waitKey(1) & 0xFF == ord('q'):
        break
# 当所有事完成，释放 VideoCapture 对象
cap.release()
cv.destroyAllWindows()
```

cap.read() 返回一个 bool 值(`True` / `False`)。如果加载成功，它会返回 `True` 。因此，你可以通过这个返回值判断视频是否结束。

有时，cap 可能没有初始化 capture。在这种情况下，此代码显示错误。你可以通过该方法 **cap.isOpened()** 检查它是否初始化。如果它是 True，那么是好的，否则用 **cap.open()** 打开在使用。

你也可以通过使用 **cap.get(propId)** 函数获取一些视频的特征，这里的 propId 是一个 0-18 的数字，每个数字代表视频的一个特征 (如果这个视频有)，或者使用 cv::VideoCapture::get() 获取全部细节。它们中有些值可以使用 **cap.set(propId, value)** 修改。Value 就是你想要的新值

例如：我可以用 **cap.get(cv.CAP_PROP_FRAME_WIDTH)** 获得宽，**cap.get(cv.CAP_PROP_FRAME_HEIGHT)** 获得高。它返回的是 640x480，但是我想把它修改为 320x240。仅使用 **ret = cap.set(cv.CAP_PROP_FRAME_WIDTH,320)** 和 **ret = cap.set(cv.CAP_PROP_FRAME_HEIGHT,240)**

Note

- 如果给你报错了，确保用任意其他的相机程序 (如 Linux 下的 Cheese 程序) 可以正常工作

播放视频文件

它和从相机捕获一样，只需要用视频文件名更改相机索引。同时显示 frame，为 cv.waitKey() 使用合适的时间。如果它太小，视频将非常快，如果太大，视频将很慢 (嗯，这就是如何显示慢动作)。正常情况下，25 毫秒就可以了。

```
import numpy as np
import cv2 as cv
cap = cv.VideoCapture('vtest.avi')
while(cap.isOpened()):
    ret, frame = cap.read()
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    cv.imshow('frame',gray)
    if cv.waitKey(1) & 0xFF == ord('q'):
        break
cap.release()
cv.destroyAllWindows()
```

Note

- 确保 ffmpeg 和 gstreamer 安装合适的版本。有时，使用 Video Capture 是比较头痛的，主要是因为错误的安装 ffmpeg 或 gstreamer。

保存视频

我们捕获视频，逐帧处理然后保存下来。对于图像来说，是非常的简单，就用 cv.imwrite()。这里需要做更多的工作。

这次我们创建一个 **VideoWriter** 对象。我们应该指定输出文件的名字 (例如：output.avi)。然后我们应该指定 **FourCC** 码 (下一段有介绍)。然后应该传递每秒帧数和帧大小。最后一个是 **isColor** flag。如果是 True，编码器期望彩色帧，否则它适用于灰度帧。

FourCC 是用于指定视频解码器的 4 字节代码。这里 fourcc.org 是可用编码的列表。它取决于平台，下面编码就很好。

- In Fedora: DIVX, XVID, MJPG, X264, WMV1, WMV2. (XVID 是最合适的. MJPG 结果比较大. X264 结果比较小)
- In Windows: DIVX (还需要测试和添加跟多内容)
- In OSX: MJPG (.mp4), DIVX (.avi), X264 (.mkv).

对于 MJPG， FourCC 的代码作为 **cv.VideoWriter_fourcc('M','J','P','G')** 或 **cv.VideoWriter_fourcc(*'MJPG')** 传递。

下面的代码从相机捕获，在垂直方向翻转每一帧然后保存它。

```
import numpy as np
import cv2 as cv
cap = cv.VideoCapture(0)
# 声明编码器和创建 VideoWrite 对象
fourcc = cv.VideoWriter_fourcc(*'XVID')
out = cv.VideoWriter('output.avi',fourcc, 20.0, (640,480))
while(cap.isOpened()):
    ret, frame = cap.read()
    if ret==True:
        frame = cv.flip(frame,0)
        # 写入已经翻转好的帧
        out.write(frame)
        cv.imshow('frame',frame)
        if cv.waitKey(1) & 0xFF == ord('q'):
            break
    else:
        break
# 释放已经完成的工作
cap.release()
out.release()
cv.destroyAllWindows()
```

其他资源

练习

绘图功能

目标

在本次会议中：

- 用 OpenCV 画不同的几何图形
- 你要学习这些函数：**cv.line()**, **cv.circle()** , **cv.rectangle()**, **cv.ellipse()**, **cv.putText()** 等。

Code

上面的这些函数，你能看到一些相同的参数：

- img：你想画的图片
- color：形状的颜色，如 BGR，它是一个元组，例如：蓝色(255,0,0)。对于灰度图，只需传一个标量值。
- thickness： 线或圆等的厚度。如果传 **-1** 就是像圆的闭合图形，它将填充形状。 *默认 thickness = 1*
- lineType：线条类型，如 8 连接，抗锯齿线等。默认情况下，它是 8 连接。**cv.LINE_AA** 画出抗锯齿线，非常好看的曲线。

画线

去画一条线，你需要传递线条的开始和结束的坐标。我们将创建一个黑色图像，并在坐上角到右下角画一条蓝色的线

```
import numpy as np
import cv2 as cv
# 创建一个黑色的图像
img = np.zeros((512,512,3), np.uint8)
# 画一条 5px 宽的蓝色对角线
cv.line(img,(0,0),(511,511),(255,0,0),5)
```

画矩形

画一个矩形，你需要矩形的左上角和右下角。这次我们将会在图像的右上角画一个绿色的矩形。

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

画圆

画一个圆，你需要它的圆心和半径。我们将在上面绘制的矩形上画一个内圆。

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

画椭圆

画一个椭圆，你需要传好几个参数。一个参数是圆心位置 (x,y)。下个参数是轴的长度 (长轴长度，短轴长度)。角度是椭圆在你逆时针方向的旋转角度。startAngle 和 endAngle 表示从长轴顺时针方向测量的椭圆弧的起点和终点。如整圆就传 0 和 360。更多细节请看 **cv.ellipse()** 的文档。下面是在这个图像中间画的一个半椭圆例子。

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

画多边形

画多边形，首先你需要顶点的做坐标。将这些点组成一个形状为 ROWSx1x2 的数组，ROWS 是顶点数，它应该是 int32 类型。这里我们绘制一个顶点是黄色的小多边形。

```
pts = np.array([[10,5],[20,30],[70,20],[50,10]], np.int32)
pts = pts.reshape((-1,1,2))
cv.polylines(img,[pts],True,(0,255,255))
```

Note

- 如果地三个是 False，你将获得所有点的折线，而不是一个闭合形状。
- **cv.polylines()** 能画很多线条。只需创建你想绘制所有线条的列表，然后将其传给这个函数。所有线条都将单独绘制。绘制一组线条比调用 **cv.line()** 好很多，快很多。

给图像加文字

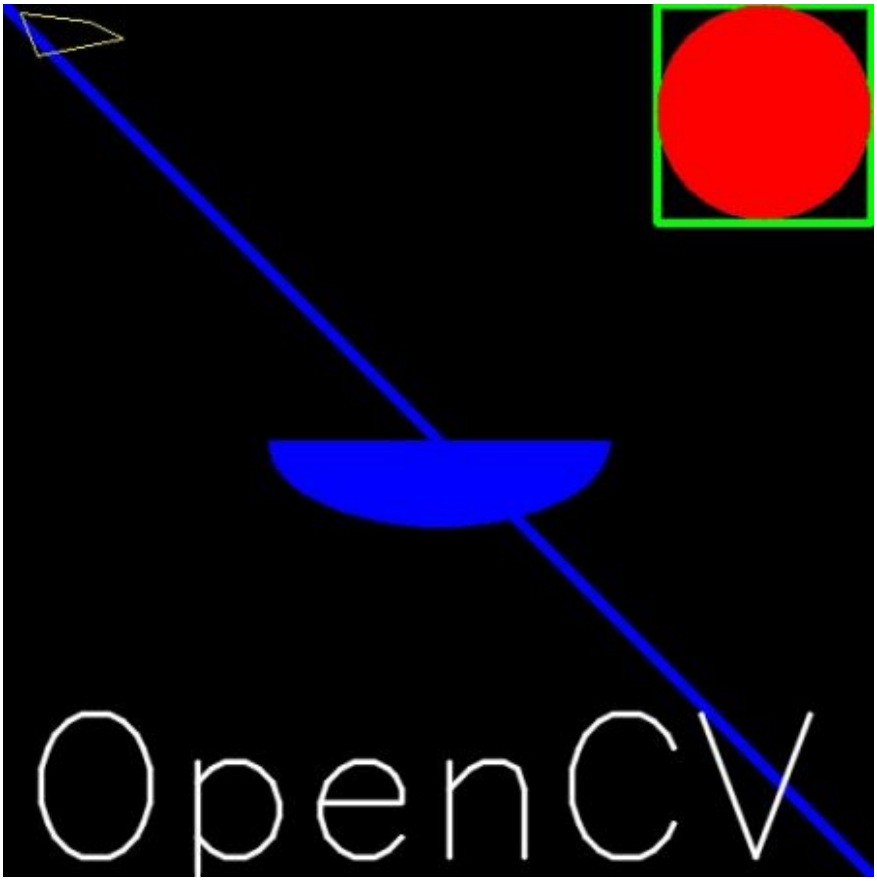
在图像上加文字，你需要指定以下内容。

- 你想写的文字数据。
- 你想写的位置坐标 (如 左下角开始)。
- 字体类型 (支持的字体，查看 **cv.putText()** 文档)
- 常规的如颜色，粗细，线型等。为了更好看，线型使用 lintType = cv.LINE_AA

我们将在图像上写一个白色的 **OpenCV** 。

结果

所以是时候看看我们绘制的最终结果。正如你之前学习那样，显示图像并查看。



其他资源

1. 椭圆函数中使用的角度不是我们的圆角。获取更多信息，参考 **this discussion**。

练习

1. 尝试着用 OpenCV 绘制函数画一个 OpenCV 的 logo。

鼠标作为画笔

目标

在本次会议中：

- 学习用 OpenCV 处理鼠标事件
- 你将学习这些函数：**cv.setMouseCallback()**

简单例子

这里，我们创建一个简单的程序，在图像的任何位置双击在上面画一个圆。

首先我们创建一个鼠标回调函数，该函数在鼠标事件发生时执行。鼠标事件可以是与鼠标有关的任何内容，比如鼠标左键按下，左键弹起，左键双击等等。所有鼠标事件都给我们提供坐标 (x,y)。通过这个事件和位置，我们能做任何我们喜欢的事情。要列出所有可用事件，在 Python 终端执行以下代码：

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

创建鼠标回调函数是有特定的格式，在任何地方都一样。它仅仅是函数的功能不同。因此我们的鼠标回调函数是做一件事，就是我们双击的地方画圆。所以看下面的代码。代码注释能让你明白。

```
import numpy as np
import cv2 as cv
# 鼠标回调函数
def draw_circle(event,x,y,flags,param):
    if event == cv.EVENT_LBUTTONDBLCLK:
        cv.circle(img,(x,y),100,(255,0,0),-1)
# 创建一个黑色图像，一个窗口，然后和回调绑定
img = np.zeros((512,512,3), np.uint8)
cv.namedWindow('image')
cv.setMouseCallback('image',draw_circle)
while(1):
    cv.imshow('image',img)
    if cv.waitKey(20) & 0xFF == 27:
        break
cv.destroyAllWindows()
```

更多高级例子

现在我们寻求更好的应用。这次，我们通过拖动鼠标绘制矩形或者圆 (取决于我们选的模式)，就像在 Paint 程序中一样。因此我们的鼠标回调函数有两个，一个画矩形一个画圆形。这个具体的例子将有助于我们创建和理解交互式程序，像对象跟踪，图像分割等等。

```
import numpy as np
import cv2 as cv
drawing = False # 如果 True 是鼠标按下
mode = True # 如果 True，画矩形，按下‘m’切换到曲线
ix,iy = -1,-1
# 鼠标回调函数
def draw_circle(event,x,y,flags,param):
    global ix,iy,drawing,mode
    if event == cv.EVENT_LBUTTONDOWN:
        drawing = True
        ix,iy = x,y
    elif event == cv.EVENT_MOUSEMOVE:
        if drawing == True:
            if mode == True:
                cv.rectangle(img,(ix,iy),(x,y),(0,255,0),-1)
            else:
                cv.circle(img,(x,y),5,(0,0,255),-1)
    elif event == cv.EVENT_LBUTTONUP:
        drawing = False
        if mode == True:
            cv.rectangle(img,(ix,iy),(x,y),(0,255,0),-1)
        else:
            cv.circle(img,(x,y),5,(0,0,255),-1)
```

下面我们用鼠标回调函数和 OpenCV 窗口绑定。在主循环中，我们应该设置一个‘m‘按健绑定以在矩形和圆形之间切换。

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

其他资源

练习

1. 在我们的最后一个例子中，我们绘制了填充矩形。 您修改代码以绘制未填充的矩形。

作为调色板的跟踪栏

目标

- 学习将轨迹栏绑定到 OpenCV 窗口。
- 你将学习这些函数：**cv.getTrackbarPos()**, **cv.createTrackbar()** 等等。

代码演示

这里，我们将创建简单的程序，显示你指定颜色。 你有一个显示颜色的窗口和三个轨迹栏，用来指定 B，G，R 颜色的。你可以滑动轨迹栏改变窗口的颜色。默认情况下，初始颜色被设置为黑色。 Here we will create a simple application which shows the color you specify. You have a window which shows the color and three trackbars to specify each of B,G,R colors. You slide the trackbar and correspondingly window color changes. By default, initial color will be set to Black.

对于 **cv.getTrackbarPos()** 函数，第一个参数是轨迹栏名字，第二那个是被附上窗口名字，第三个参数是默认值，第四个是最大值，第五个是回调函数，滑条改变所执行的函数。这个回调函数也有一个默认参数，表示轨迹栏位置。我们并不关心函数做什么事，所以我们简单提一下。

轨迹栏的另一个重要应用是用作按钮或者开关。OpenCV，默认情况，是没有按钮功能的。因此我们能用轨迹栏做一些这样的功能。在我们的程序中，我门创建了一个开关，其中程序只会在开关打开时有效，否则屏幕始终是黑色。

```
import numpy as np
import cv2 as cv
def nothing(x):
    pass
# 创建一个黑色图像，一个窗口
img = np.zeros((300,512,3), np.uint8)
cv.namedWindow('image')
# 创建一个改变颜色的轨迹栏
cv.createTrackbar('R','image',0,255,nothing)
cv.createTrackbar('G','image',0,255,nothing)
cv.createTrackbar('B','image',0,255,nothing)
# 创建一个开关用来启用和关闭功能的
switch = '0 : OFF \n1 : ON'
cv.createTrackbar(switch, 'image',0,1,nothing)
while(1):
    cv.imshow('image',img)
    k = cv.waitKey(1) & 0xFF
    if k == 27:
        break
    # get current positions of four trackbars
    r = cv.getTrackbarPos('R','image')
    g = cv.getTrackbarPos('G','image')
    b = cv.getTrackbarPos('B','image')
    s = cv.getTrackbarPos(switch,'image')
    if s == 0:
        img[:] = 0
    else:
        img[:] = [b,g,r]
cv.destroyAllWindows()
```

程序的截图如下：

练习

1. 创建一个 Paint 程序，具有可调节颜色和笔刷半径的轨迹栏。关于绘画，参考之前的鼠标事件教程。

图像基本操作

目标

学习：

- 访问像素值并修改它们
- 访问像素属性
- 设置感兴趣区域（ROI）
- 拆分和合并图像

本节中的几乎所有操作都主要与 Numpy 有关，而不是 Opencv。熟悉 Numpy 后才能使用 Opencv 编写更好的优化代码。

（由于大多数代码都是单行代码，所以示例将在 Python 终端中显示）

访问和修改像素值

先来理解一下，图像与一般的矩阵或张量有何不同(不考虑图像的格式，元数据等信息)。首先，一张图像有自己的属性，宽，高，通道数。其中宽和高是我们肉眼可见的属性，而通道数则是图像能呈现色彩的属性。我们都知道，光学三原色是红色，绿色和蓝色，这三种颜色的混合可以形成任意的颜色。常见的图像的像素通道也是对应的R，G，B三个通道，在OpenCV中，每个通道的取值为0～255，。(注：还有RGBA，YCrCb，HSV等其他图像通道表示)。即，一般彩色图像读进内存之后是一个h *w* c的矩阵，其中h为图像高(相当于矩阵的行)，w为图像宽(相当于矩阵列)，c为通道数。

下面我们先加载一副彩色图像，更准确的说，是一副黄色图像，如图所示。

黄色为绿色和红色的混合，所以，该图像的所有像素值都应为R=255，G=255，B=0

```
>>> import numpy as np
>>> import cv2
>>> img = cv2.imread("img/yellow.jpg")
>>> h,w,c = img.shape
#图像为128*128*3的大小
>>> print(h,w,c)
128 128 3
```

从上面的代码中可以看到，您可以通过行和列坐标访问像素值。注意,对于 常见的 RGB 图像，OpenCV的imread函数返回的是一个蓝色(Blue)、绿色(Green)、红色(Red)值的数组，维度大小为3。而对于灰度图像，仅返回相应的强度。

```
>>> img[100,100]
#OpenCV的读取顺序为B，G，R，由于图像所有像素为黄色，因此，G=255，R=255
array([  0, 255, 255], dtype=uint8)

# 仅访问蓝色通道的像素
>>> blue = img[100,100,0]
>>> print(blue)
0
```

你也可以使用同样的方法来修改像素值

```
>>> img[100,100] = [255,255,255]
>>> print(img[100,100])
[255 255 255]
```

警告

Numpy 是一个用于快速阵列计算的优化库。因此，简单地访问每个像素值并对其进行修改将非常缓慢，并不鼓励这样做。

> **注意** 上述方法通常用于选择数组的某个区域，比如前 5 行和后 3 列。对于单个像素的访问，可以选择使用 Numpy 数组方法中的 array.item()和 array.itemset()，注意它们的返回值是一个标量。如果需要访问所有的 G、R、B 的值，则需要分别为所有的调用 array.item()。

更好的访问像素和编辑像素的方法：

```
#访问 红色通道 的值
>>>img.item(10,10,2)
59

#修改 红色通道 的值
>>>img.itemset((10,10,2),100)
>>>img.item(10,10,2)
100
```

访问图像属性

图像属性包括行数，列数和通道数，图像数据类型，像素数等。

与一般的numpy.array一样，可以通过 img.shape 访问图像的形状。它返回一组由图像的行、列和通道组成的元组（如果图象是彩色的）：

```
>>>print(img.shape)
(128,128,3)
```

> **注意** 如果图像是灰度图像，则返回的元组仅包含行数和列数，因此它是检查加载的图像是灰度图还是彩色图的一种很好的方法。

通过 img.size 访问图像的总像素数：

```
>>>print(img.size)
562248
```

图像数据类型可以由 img.dtype 获得：

```
>>>print(img.dtype)
UINT8
```

> **注意** img.dtype 在调试时非常重要，因为 OpenCV—Python 代码中的大量错误是由无效的数据类型引起的。

图像中的感兴趣区域

有时，您将不得不处理某些图像区域。对于图像中的眼部检测，在整个图像上进行第一次面部检测。当获得面部时，我们单独选择面部区域并在其内部搜索眼部而不是搜索整个图像。它提高了准确性（因为眼睛总是在脸上：D）和性能（因为我们在一个小区域搜索）。

使用 Numpy 索引再次获得 ROI（感兴趣区域）。在这里，我选择球并将其复制到图像中的另一个区域：

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

检查以下结果：

拆分和合并图像通道

有时您需要在 B，G，R 通道图像上单独工作。在这种情况下，您需要将 BGR 图像分割为单个通道。在其他情况下，您可能需要将这些单独的通道连接到 BGR 图像。您可以通过以下方式完成：

```
>>>b,g,r = cv.spilt(img)
>>>img = cv.merge((b,g,r))
```

或者使用numpy.array的切片方法

```
>>>b = img[:,:,0]
```

假设您要将所有红色像素设置为零，则无需先拆分通道。Numpy 索引更快：

```
>>> img[:,:,2] = 0
```

警告

cv.spilt()是一项代价高昂的操作（就时间而言）。所以只有在你需要时才这样做，否则就使用 Numpy 索引。

制作图像边界（填充）

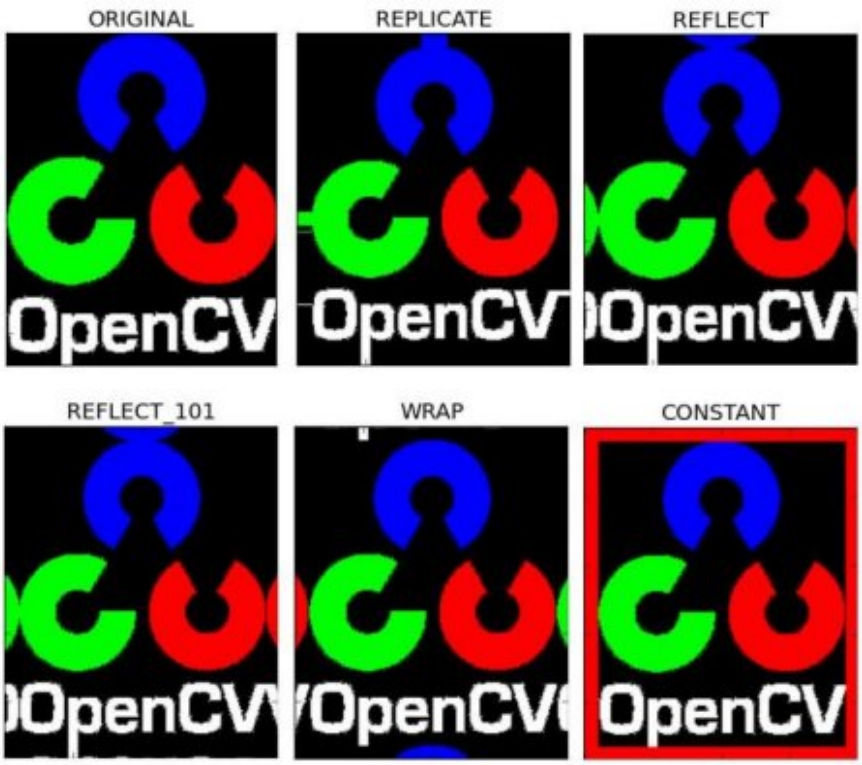
如果要在图像周围创建边框，比如相框，可以使用 cv.copyMakeBorder()。但它有更多卷积运算，零填充等应用。该函数采用以下参数：

- src-输入的图像
- top,bottom,left,right-上下左右四个方向上的边界拓宽的值
- borderType-定义要添加的边框类型的标志。它可以是以下类型：
    - cv.BORDER_CONSTANT- 添加一个恒定的彩色边框。该值应作为下一个参数value给出。
    - cv.BORDER_REFLECT-边框将是边框元素的镜像反射，如下所示：fedcba|abcdefgh|hgfedcb
    - cv.BORDER_REFLECT_101或者 cv.BORDER_DEFAULT-与上面相同，但略有改动，如下所示： gfedcb | abcdefgh | gfedcba
    - cv.BORDER_REPLICATE -最后一个元素被复制，如下所示：aaaaaa | abcdefgh | hhhhhhh
    - cv.BORDER_WRAP-不好解释，它看起来像这样： cdefgh | abcdefgh | abcdefg
- value- 如果边框类型为cv.BORDER_CONSTANT，则这个值即为要设置的边框颜色

下面是一个示例代码，演示了所有这些边框类型，以便更好地理解：

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
BLUE = [255,0,0]
img1 = cv.imread('opencv-logo.png')
replicate = cv.copyMakeBorder(img1,10,10,10,10,cv.BORDER_REPLICATE)
reflect = cv.copyMakeBorder(img1,10,10,10,10,cv.BORDER_REFLECT)
reflect101 = cv.copyMakeBorder(img1,10,10,10,10,cv.BORDER_REFLECT_101)
wrap = cv.copyMakeBorder(img1,10,10,10,10,cv.BORDER_WRAP)
constant= cv.copyMakeBorder(img1,10,10,10,10,cv.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()
```

请参阅下面的结果。（图像是通过 matplotlib 展示的。因此红色和蓝色通道将互换）：



其他资源

练习

图像的算术运算

目标

- 学习对图像的几种算术运算，如加法，减法，按位运算等。
- 您将学习以下函数：cv.add()，cv.addWeighted()等。

图像加法

您可以通过 OpenCV 函数，cv.add()或简单地通过 numpy 操作将两个图像相加，res = img1 + img2。两个图像应该具有相同的深度和类型，或者第二个图像可以是像素值，比如(255,255,255)，白色值。

> **注意** OpenCV 相加操作和 Numpy 相加操作之间存在差异。OpenCV 添加是饱和操作，而 Numpy 添加是模运算。要注意的是，两种加法对于结果溢出的数据，会通过某种方法使其在限定的数据范围内。

例如，请考虑以下示例：

```
>>> x = np.uint8([250])
>>> y = np.uint8([10])

>>> print(cv.add(x,y)) #250 + 10 =260 => 255
[[255]]

>>> print(x + y)
[4]
```

在将两个图象相加时会发现 OpenCV 函数能够提供更好的结果，所以尽可能地选择 OpenCV 函数。

图像混合

这也是将图像相加，但是对图像赋予不同的权重，从而给出混合感或透明感。图像按以下等式添加：

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

通过在(0,1)之间改变  的值, 可以用来对两幅图像或两段视频产生时间上的 *画面叠化* （cross-dissolve）效果，就像在幻灯片放映和电影制作中那样（很酷吧?）（译者注：在幻灯片翻页时可以设置为前后页缓慢过渡以产生叠加效果，电影中经常在情节过渡时出现画面叠加效果）。

在这里，我拍了两张图片将它们混合在一起。第一图像的权重为 0.7，第二图像的权重为 0.3。cv.addWeighted()在图像上应用以下等式。

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

```
img1= cv.imread('ml.png')
img2= cv.imread('opencv-logo.png')

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

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

观察以下结果：

按位操作

这包括按位 AND，OR，NOT 和 XOR 运算。它们在提取图像的某一部分（我们将在后面的章节中看到）、定义和使用非矩形 ROI 等方面非常有用。下面我们将看到如何更改图像的特定区域的示例：

假如我想加一个OpenCV的 logo在一个图像上，如果只是简单的将两张图像想加，则会改变叠加处的颜色。如果进行上面所说的混叠操作，则会得到一个有透明效应的结果，但我希望得到一个不透明的logo。如果logo是一个矩形logo，那可以用上节所讲的ROI来做。但是OpenCV的logo是不规则形状的，所以用下面的bitwise操作来进行。

```
#加载两张图片
img1 = cv.imread ('messi5.jpg')
img2 = cv.imread ('opencv-logo-white.png')
#我想在左上角放置一个logo，所以我创建了一个 ROI,并且这个ROI的宽和高为我想放置的logo的宽和高
rows, cols, channels = img2.shape
roi = img1 [0: rows, 0: cols]
#现在创建一个logo的掩码，通过对logo图像进行阈值，并对阈值结果并创建其反转掩码
img2gray = cv.cvtColor (img2, cv.COLOR_BGR2GRAY)
ret, mask = cv.threshold (img2gray, 10,255, cv.THRESH_BINARY)
mask_inv = cv.bitwise_not (mask)
#现在使 ROI 中的徽标区域变黑
img1_bg = cv.bitwise_and (roi, roi, mask = mask_inv)
#仅从徽标图像中获取徽标区域。
img2_fg = cv.bitwise_and (img2, img2, mask = mask)
#在 ROI 中放置徽标并修改主图像
dst = cv.add (img1_bg, img2_fg)
img1 [0: rows, 0: cols] = dst
cv.imshow ('res', img1)
cv.waitKey (0)
cv.destroyAllWindows ()
```

请参阅下面的结果。左图显示了我们创建的蒙版。右图显示最终结果。为了加深理解，请在上面的代码中显示所有中间图像，尤其是 img1_bg 和 img2_fg。

其他资源

练习

1. 使用cv.addWeighted()函数在文件夹中创建图像之间平滑过渡的幻灯片并放映。

性能测量和改进技术

目标

在图像处理中，由于我们每秒需要处理大量操作，因此我们的代码不仅要提供正确的解决方案，还要以最快的方式提供。所以，在本章中，我们将学习

- 衡量代码的性能
- 一些提高代码性能的技巧
- 我们将看到以下函数：cv.getTickCount,cv.getTickFrequency等。

除了 OpenCV 之外，Python 还提供了一个**时间**模块，有助于记录代码执行时间。另一个模块**配置文件**有助于获取有关代码的详细报告，例如代码中每个函数花费的时间，调用函数的次数等。但是，如果您使用的是 IPython，所有这些功能都集成在一个用户友好的方式中。我们将看到一些重要的内容，有关更多详细信息，请查看**其他资源**部分中的链接。

使用 OpenCV 测量性能

cv.getTickCount函数返回参考事件（如机器开启时刻）到调用此函数的时钟周期数。因此，如果在函数执行之前和之后调用它，则会获得用于执行函数的时钟周期数。

cv.getTickFrequency函数返回时钟周期的频率，或每秒钟的时钟周期数。因此，要在几秒钟内找到执行时间，您可以执行以下操作

```
e1 = cv.getTickCount()
#你的执行的代码
e2 = cv.getTickCount()
time = (e2 - e1)/cv.getTickFrequency()
```

我们将通过以下示例进行演示。下面的例子使用奇数大小从 5 到 49 的内核进行中值过滤。（不要担心结果会是什么样的，这不是我们的目标）：

```
img1 = cv.imread('messi5.jpg')
e1 = cv.getTickCount()
for i in xrange(5,49,2):
    img1 = cv.medianBlur(img1,i)
e2 = cv.getTickCount()
t = (e2 - e1)/cv.getTickFrequency()
print( t )
# 得到的结果是 0.521107655 秒
```

> **注意** 你可以用时间模块做同样的事情。使用 time.time()函数来替代 cv.getTickCount，然后取两次的差异，比如start=time.time(), end = time.time(), print(end-start)。

OpenCV 中的默认优化

许多 OpenCV 功能都使用 SSE2，AVX 等进行了优化。它还包含未经优化的代码。因此，如果我们的系统支持这些功能，我们应该利用它们（几乎所有现代处理器都支持它们）。优化功能在编译时是默认启用的，因此，OpenCV 在启用时运行优化代码，否则运行未优化代码。您可以使用cv.useOptimized()来检查它是否已启用/禁用，并使用cv.setUseOptimized()来启用/禁用它。让我们看一个简单的例子。

```
# 检查是否使用了优化
In [5]: cv.useOptimized()
Out[5]: True
In [6]: %timeit res = cv.medianBlur(img,49)
10 loops, best of 3: 34.9 ms per loop
# 禁用优化
In [7]: cv.setUseOptimized(False)
In [8]: cv.useOptimized()
Out[8]: False
In [9]: %timeit res = cv.medianBlur(img,49)
10 loops, best of 3: 64.1 ms per loop
```

请参阅，优化中值过滤比未优化版本快 2 倍。如果检查其来源，您可以看到中值过滤是 SIMD 优化的。因此，您可以使用它来在代码顶部启用优化（请记住它默认启用）。

测量 IPython 中的性能

有时您可能需要比较两个类似操作的性能。IPython 为您提供了一个神奇的命令时间来执行此操作。它运行代码几次以获得更准确的结果。注意，它们适用于测量单行代码。

例如，你知道以下哪个加法操作更好， x = 5; y = x*2, *x = 5; y = x*x, x = np.uint8([5]); y = x*x 或者 y = np.square(x) ?我们可以使用IPython中的timeit魔术指令方便的进行测试。

```
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
```

你可以看到，x = 5; y = x * x 是最快的，与 Numpy 相比快了约 20 倍。如果您也考虑创建矩阵，它可能会快达 100 倍。很酷，对吗？（Numpy 开发者正在研究这个问题）

> **注意** Python 标量操作比 Numpy 标量操作更快。因此对于包含一个或两个元素的操作，Python 标量比 Numpy 数组更好。当阵列的大小稍大时，Numpy 会占据优势。

我们将再尝试一个例子。这次，我们将比较同一图像的cv.countNonZero()和np.count_nonzero()的性能。

```
In [35]: %timeit z = cv.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，OpenCV 功能比 Numpy 功能快近 25 倍。

> **注意** 通常，OpenCV 函数比 Numpy 函数更快。因此，对于相同的操作，OpenCV 功能是首选。但是，可能有例外，尤其是当 Numpy 使用视图而不是副本时。

更多 IPython 魔术命令

还有其他一些魔术命令可以测量性能，分析，线性分析，内存测量等。它们都有很好的文档记录。因此，此处仅提供这些文档的链接。建议有兴趣的读者试用。

性能优化技术

有几种技术和编码方法可以利用 Python 和 Numpy 的最大性能。这里仅注明相关的内容，并提供重要来源的链接。这里要注意的主要是，首先尝试以简单的方式实现算法。一旦它正常工作，对其进行分析，找到瓶颈并对其进行优化。

1. 尽量避免在 Python 中使用循环，尤其是双循环/三循环等。它们本质上很慢。
2. 将算法/代码矢量化到最大可能范围，因为 Numpy 和 OpenCV 针对向量运算进行了优化。
3. 利用缓存一致性。
4. 除非需要，否则永远不要复制数组。尝试使用视图。复制多维数组是一项耗费时间和空间都很多的操作。(译者注：译者认为这儿的意思是除非必要，否在尽量不要使用深拷贝)

即使在完成所有这些操作之后，如果您的代码仍然很慢，或者使用大型循环是不可避免的，请使用其他库（如 Cython）来加快速度。

其他资源

1. Python 优化技术
2. Scipy 讲义-高级 Numpy
3. Ipython 中的时序和分析

练习

目标

在本教程中：

- 你会学到如何将图片从一个颜色空间转换到另一个，例如 BGR 到 Gray，BGR 到 HSV 等。
- 另外，我们会创建一个从视频中提取彩色对象的应用。
- 你会学到如下函数：**cv.cvtColor()**，**cv.inRange()**

改变颜色空间

在 OpenCV 中有超过 150 种颜色空间转换的方法。但我们仅需要研究两个最常使用的方法，他们是 BGR 到 Gray，BGR 到 HSV。

我们使用 cv.cvtColor(input_image, flag)函数进行颜色转换，其中 flag 决定了转换的类型。

对于 BGR 到 Gray 转换我们令 flag 为 **cv.COLOR_BGR2GRAY**。 同样，对于 BGR 到 HSV, 我们令 flag 为 **cv.COLOR_BGR2HSV**。如想得到其他 flag 值，只需要在 Python 终端中输入如下命令：

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

> **注意** 对于 HSV, 色调范围为 [0,179], 饱和度范围为 [0,255] ， 像素值为 [0,255]. 不同的软件使用不同的比例. 所以如果你想用 OpenCV 的值与别的软件的值作对比，你需要归一化这些范围。

目标追踪

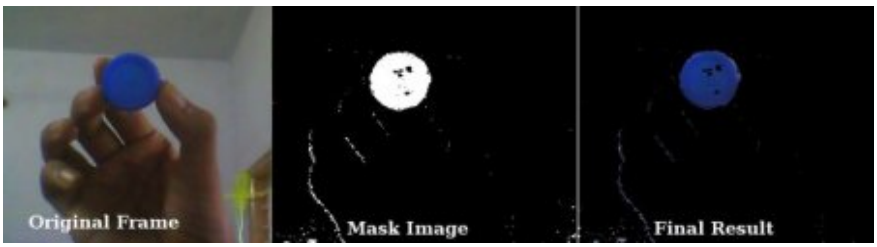
现在我们知道了如何将 BGR 图片转化为 HSV 图片，我们可以使用它去提取彩色对象。HSV 比 BGR 在颜色空间上更容易表示颜色。在我们的应用中，我们会尝试提取一个蓝色的彩色对象，方法为：

- 提取每一视频帧。
- 将 BGR 转化为 HSV 颜色空间。
- 我们将 HSV 图片的阈值设为蓝色范围。
- 现在提取出了蓝色对象，我们可以随意处理图片了 。

下面是代码:

```
import cv2 as cv
import numpy as np
cap = cv.VideoCapture(0)
while(1):
    # Take each frame
    _, frame = cap.read()
    # Convert BGR to HSV
    hsv = cv.cvtColor(frame, cv.COLOR_BGR2HSV)
    # define range of blue color in HSV
    lower_blue = np.array([110,50,50])
    upper_blue = np.array([130,255,255])
    # Threshold the HSV image to get only blue colors
    mask = cv.inRange(hsv, lower_blue, upper_blue)
    # Bitwise-AND mask and original image
    res = cv.bitwise_and(frame,frame, mask= mask)
    cv.imshow('frame',frame)
    cv.imshow('mask',mask)
    cv.imshow('res',res)
    k = cv.waitKey(5) & 0xFF
    if k == 27:
        break
cv.destroyAllWindows()
```

接下来的图片显示了追踪蓝色对象:



> **注意** 在图片中有一些噪声，在之后的章节中我们会学习如何消除。 这是目标追踪最简单的例子。当你学会了轮廓函数，你就可以做的更多，例如找到对象的质心并且使用它来追踪对象，只需在相机和前面移动你的手就可以绘制图表，还有许多其他有趣的东西。

如何找到 HSV 值去追踪?

这在 **stackoverflow.com** 中是经常见到的问题。 这个问题非常简单，你可以使用相同的函数：**cv.cvtColor()**。 不需要输入图片，你只需要输入你需要的 BGR 值即可. 例如, 为了找到绿色的 HSV 值, 可以在 Python 终端中输入以下代码:

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

现在你可以取 [H-10, 100,100] 和 [H+10, 255, 255] 分别作为上界和下界. 除此之外，你可以使用任何图像编辑工具（如 GIMP）或任何在线转换器来查找这些值，但不要忘记调整 HSV 范围。

其他资源

练习

1、尝试找到一种方法来提取多个彩色对象，例如，同时提取红色、蓝色、绿色对象。

目标

在本教程中：

- 你将会学到将不同的几何变换应用于图像，如平移、旋转、仿射变换等。
- 另外，我们会创建一个从视频中提取彩色对象的应用。
- 你会学到如下函数：**cv.getPerspectiveTransform**

变换

OpenCV 提供了两个变换函数， **cv.warpAffine** 和 **cv.getPerspectiveTransform** ，他们可以完成所有类型的变换。 **cv.warpAffine** 输入为 *23 的变换矩阵，* ***cv.getPerspectiveTransform*** *输入为* 23 的变换矩阵。

比例

比例是调整图片的大小。 OpenCV 使用 **cv.resize()** 函数进行调整。可以手动指定图像的大小，也可以指定比例因子。可以使用不同的插值方法。最好的插值方法是用于收缩的 **cv.INTER_AREA** 和 **cv.INTER_CUBIC** （慢）和快速方法 **cv.INTER_LINEAR** 。默认情况下，所使用的插值方法都是 **cv.INTER_AREA** 。你可以使用如下方法调整输入图片大小：

```
import numpy as np
import cv2 as cv
img = cv.imread('messi5.jpg')
res = cv.resize(img,None,fx=2, fy=2, interpolation = cv.INTER_CUBIC)
#OR
height, width = img.shape[:2]
res = cv.resize(img,(2*width, 2*height), interpolation = cv.INTER_CUBIC)
```

变换

变换是物体位置的移动。如果知道 **（x，y）** 方向的偏移量，假设为 **(t*{x},t*{y})**，则可以创建如下转换矩阵 **M**：

 图片

您可以将语句放入 np.float32 类型的 numpy 数组中，并将其传递到 **cv.warpAffine** 中。请参见以下转换（100,50）的示例：

```
import numpy as np
import cv2 as cv
img = cv.imread('messi5.jpg',0)
rows,cols = img.shape
M = np.float32([[1,0,100],[0,1,50]])
dst = cv.warpAffine(img,M,(cols,rows))
cv.imshow('img',dst)
cv.waitKey(0)
cv.destroyAllWindows()
```

警告

cv.warpAffine 函数的第三个参数是输出图像的大小，其形式应为（宽度、高度）。记住宽度=列数，高度=行数。

结果：

旋转

以 **\Theta** 角度旋转图片的转换矩阵形式为：

图片

但 Opencv 提供了可变旋转中心的比例变换，所以你可以在任意位置旋转图片，修改后的转换矩阵为：

图片

其中：

图片

为了找到这个转换矩阵，opencv 提供了一个函数， **cv.getRotationMatrix2D** 。请查看下面的示例，它将图像相对于中心旋转 90 度，而不进行任何缩放。

```
img = cv.imread('messi5.jpg',0)
rows,cols = img.shape
# cols-1 and rows-1 are the coordinate limits.
M = cv.getRotationMatrix2D(((cols-1)/2.0,(rows-1)/2.0),90,1)
dst = cv.warpAffine(img,M,(cols,rows))
```

结果：

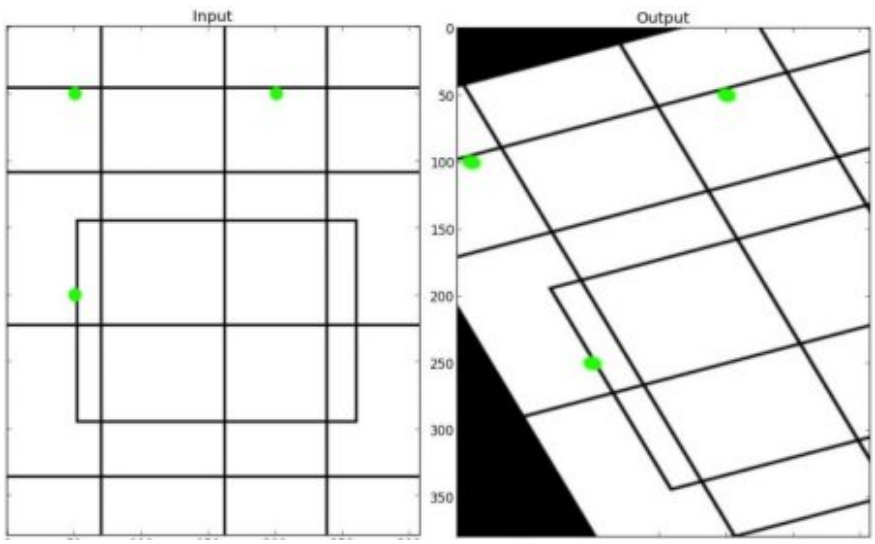
仿射变换

在仿射变换中，原始图像中的所有平行线在输出图像中仍然是平行的。为了找到变换矩阵，我们需要从输入图像中取三个点及其在输出图像中的对应位置。然后 **cv.getPerspectiveTransform** 将创建一个 2x3 矩阵，该矩阵将传递给 **cv.warpAffine** 。

查看下面的示例，并注意我选择的点（用绿色标记）：

```
img = cv.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 = cv.getAffineTransform(pts1,pts2)
dst = cv.warpAffine(img,M,(cols,rows))
plt.subplot(121),plt.imshow(img),plt.title('Input')
plt.subplot(122),plt.imshow(dst),plt.title('Output')
plt.show()
```

结果：



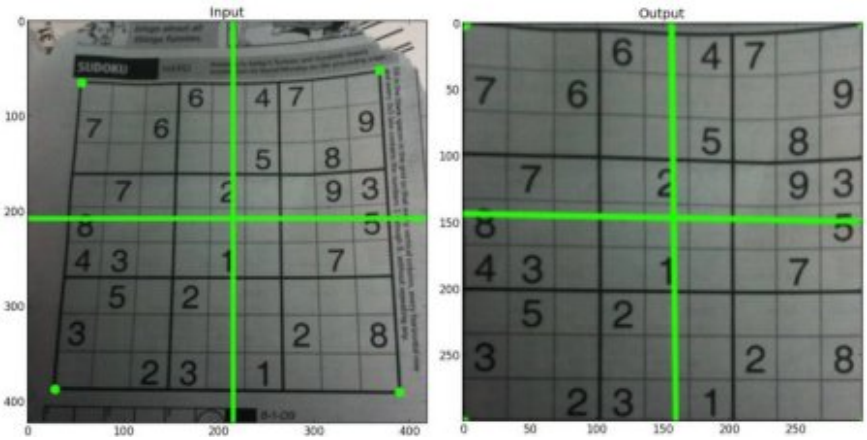
感知变换

对于感知转换，你需要一个 3x3 变换矩阵。即使在转换之后，直线也将保持直线。要找到这个变换矩阵，需要输入图像上的 4 个点和输出图像上的相应点。在这四点中，有三点不应该共线。然后通过 **cv.getPerspectiveTransform** 找到变换矩阵。然后对这个 3x3 变换矩阵使用 **cv.warpPerspective**。

请看代码：

```
img = cv.imread('sudoku.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 = cv.getPerspectiveTransform(pts1,pts2)
dst = cv.warpPerspective(img,M,(300,300))
plt.subplot(121),plt.imshow(img),plt.title('Input')
plt.subplot(122),plt.imshow(dst),plt.title('Output')
plt.show()
```

结果：



其他资源

1、"Computer Vision: Algorithms and Applications", Richard Szelisk

练习

目标

在本教程中：

- 你会学到简单阈值法，自使用阈值法，以及 Otsu 阈值法等。
- 你会学到如下函数：**cv.threshold**，**cv.adaptiveThreshold** 等。

简单阈值法

此方法是直截了当的。如果像素值大于阈值，则会指定一个值（可能为白色），否则会指定另一个值（可能为黑色）。使用的函数是 cv.threshold。第一个参数是源图像，它应该是灰度图像。第二个参数是阈值，用于对像素值进行分类。第三个参数是 maxval，它表示像素值大于（有时小于）阈值时要给定的值。opencv 提供了不同类型的阈值，由函数的第四个参数决定。不同的类型有：

- **cv.THRESH_BINARY**
- **cv.THRESH_BINARY_INV**
- **cv.THRESH_TRUNC**
- **cv.THRESH_TOZERO**
- **cv.THRESH_TOZERO_INV**

文档清楚地解释了每种类型的含义。请查看文档。

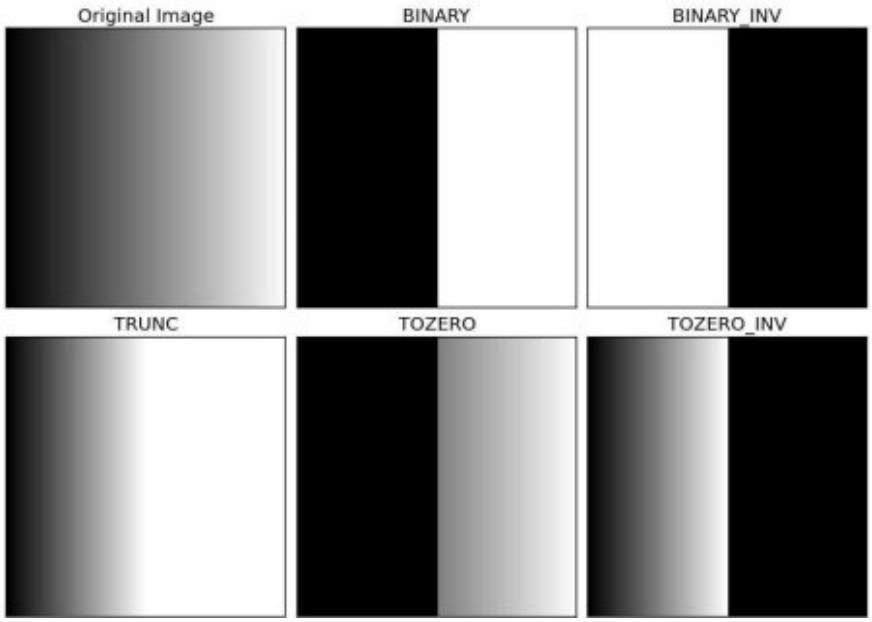
获得两个输出。第一个是 retval，稍后将解释。第二个输出是我们的阈值图像。

代码如下：

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
img = cv.imread('gradient.png',0)
ret,thresh1 = cv.threshold(img,127,255,cv.THRESH_BINARY)
ret,thresh2 = cv.threshold(img,127,255,cv.THRESH_BINARY_INV)
ret,thresh3 = cv.threshold(img,127,255,cv.THRESH_TRUNC)
ret,thresh4 = cv.threshold(img,127,255,cv.THRESH_TOZERO)
ret,thresh5 = cv.threshold(img,127,255,cv.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** 绘制多个图像，我们使用 plt.subplot()函数。有关详细信息，请查看 Matplotlib 文档。

结果如下所示：



自适应阈值

在前一节中，我们使用一个全局变量作为阈值。但在图像在不同区域具有不同照明条件的条件下，这可能不是很好。在这种情况下，我们采用自适应阈值。在此，算法计算图像的一个小区域的阈值。因此，我们得到了同一图像不同区域的不同阈值，对于不同光照下的图像，得到了更好的结果。

它有三个“特殊”输入参数，只有一个输出参数。

Adaptive Method-它决定如何计算阈值。

- **cv.ADAPTIVE_THRESH_MEAN_C** 阈值是指邻近地区的平均值。
- **cv.ADAPTIVE_THRESH_GAUSSIAN_C** 阈值是权重为高斯窗的邻域值的加权和。

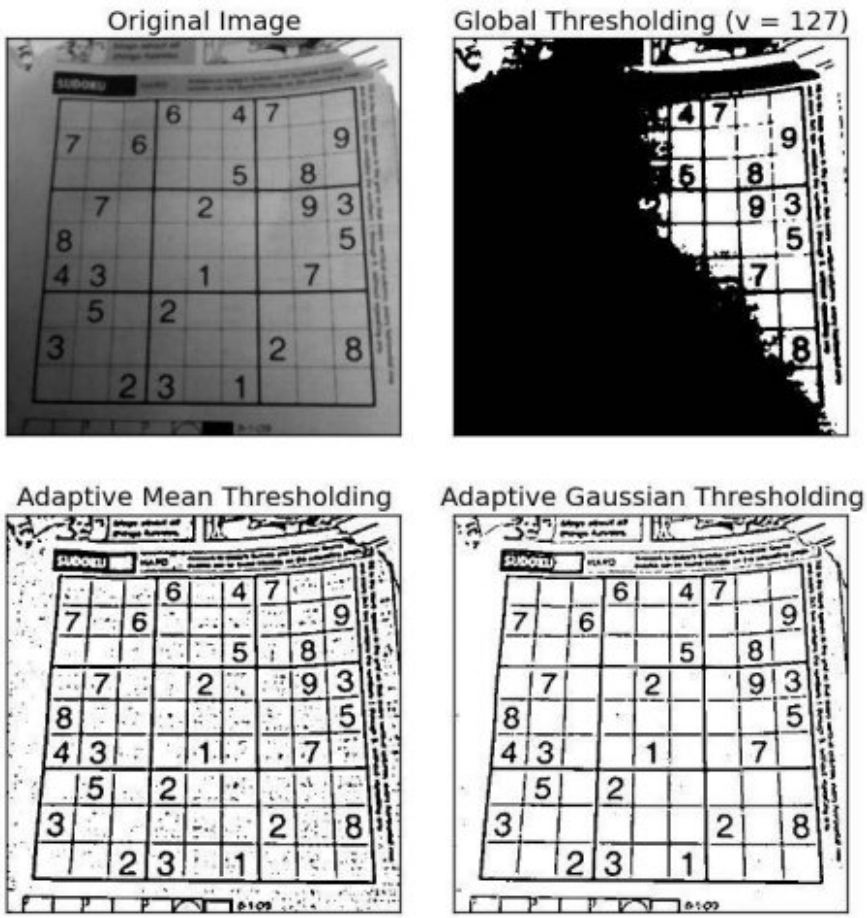
Block Size-它决定了附近区域的大小。

C-它只是一个常数，从平均值或加权平均值中减去。

下面的代码比较了具有不同照明的图像的全局阈值和自适应阈值：

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
img = cv.imread('sudoku.png',0)
img = cv.medianBlur(img,5)
ret,th1 = cv.threshold(img,127,255,cv.THRESH_BINARY)
th2 = cv.adaptiveThreshold(img,255,cv.ADAPTIVE_THRESH_MEAN_C,\
            cv.THRESH_BINARY,11,2)
th3 = cv.adaptiveThreshold(img,255,cv.ADAPTIVE_THRESH_GAUSSIAN_C,\
            cv.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()
```

结果如下所示：



Otsu 二值化

在第一部分中，我告诉过您有一个参数 retval。当我们进行 Otsu 二值化时，它的用途就来了。那是什么？

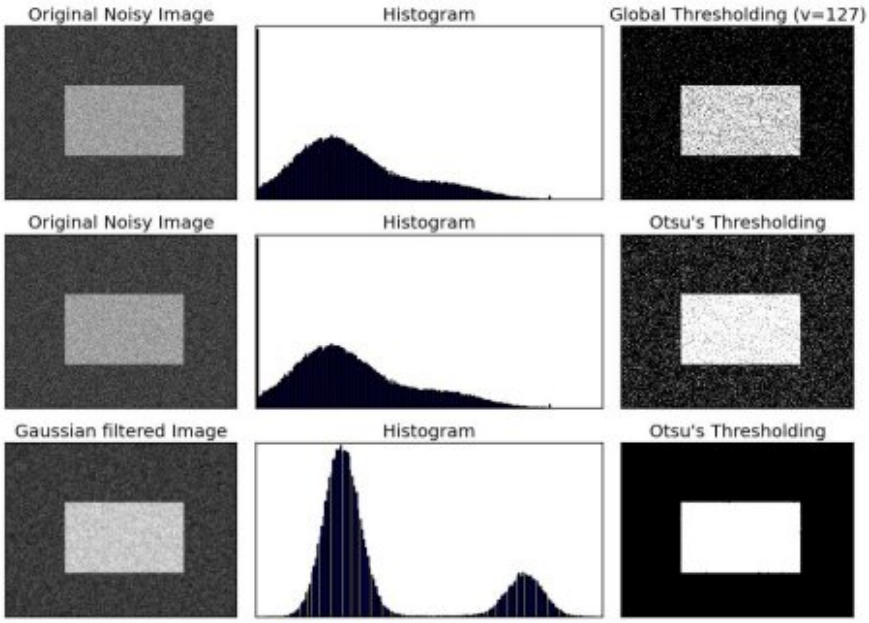
在全局阈值化中，我们使用一个任意的阈值，对吗？那么，我们如何知道我们选择的值是好的还是不好的呢？答案是，试错法。但是考虑一个双峰图像（简单来说，双峰图像是一个直方图有两个峰值的图像）。对于那个图像，我们可以近似地取这些峰值中间的一个值作为阈值，对吗？这就是 Otsu 二值化所做的。所以简单来说，它会自动从双峰图像的图像直方图中计算出阈值。（对于非双峰图像，二值化将不准确。）

为此，我们使用了 **cv.threshold** 函数，但传递了一个额外的符号 **cv.THRESH_OTSU** 。对于阈值，只需通过零。然后，该算法找到最佳阈值，并作为第二个输出返回 retval。如果不使用 otsu 阈值，则 retval 与你使用的阈值相同。

查看下面的示例。输入图像是噪声图像。在第一种情况下，我应用了值为 127 的全局阈值。在第二种情况下，我直接应用 otsu 阈值。在第三种情况下，我使用 5x5 高斯核过滤图像以去除噪声，然后应用 otsu 阈值。查看噪声过滤如何改进结果。

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
img = cv.imread('noisy2.png',0)
# 全局阈值
ret1,th1 = cv.threshold(img,127,255,cv.THRESH_BINARY)
# Otsu 阈值
ret2,th2 = cv.threshold(img,0,255,cv.THRESH_BINARY+cv.THRESH_OTSU)
# 经过高斯滤波的 Otsu 阈值
blur = cv.GaussianBlur(img,(5,5),0)
ret3,th3 = cv.threshold(blur,0,255,cv.THRESH_BINARY+cv.THRESH_OTSU)
# 画出所有的图像和他们的直方图
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"]
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()
```

结果如下：



Otsu 二值化原理

本节演示了 otsu 二值化的 python 实现，以展示它的实际工作方式。如果你不感兴趣，可以跳过这个。

由于我们使用的是双峰图像，因此 Otsu 的算法试图找到一个阈值（t），该阈值将由关系给出的类内加权方差最小化：

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

$$ 其中：

$$ \begin{eqnarray} q1(t) = \sum{i=1}^{t} P(i) \quad & \quad q2(t) = \sum{i=t+1}^{I} P(i) \ \mu1(t) = \sum{i=1}^{t} \frac{iP(i)}{q1(t)} \quad & \quad \mu_2(t) = \sum{i=t+1}^{I} \frac{iP(i)}{q2(t)} \ \sigma_1^2(t) = \sum{i=1}^{t} [i-\mu1(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{eqnarray}

$$

它实际上找到一个 T 值，它位于两个峰值之间，这样两个类的方差最小。它可以简单地在 python 中实现，如下所示：

```
img = cv.imread('noisy2.png',0)
blur = cv.GaussianBlur(img,(5,5),0)
# 找到归一化直方图还有累计分布函数
hist = cv.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]) # 概率
    q1,q2 = Q[i],Q[255]-Q[i] # 类别总和
    b1,b2 = np.hsplit(bins,[i]) # 权重
    # f 找到均值与方差
    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
    # 计算最小函数
    fn = v1*q1 + v2*q2
    if fn < fn_min:
        fn_min = fn
        thresh = i
# 用 OpenCV 函数的 otsu'阈值
ret, otsu = cv.threshold(blur,0,255,cv.THRESH_BINARY+cv.THRESH_OTSU)
print( "{} {}".format(thresh,ret) )
```

> **注意** 这其中有很多新的函数，我们会在之后章节讲述。

其他资源

1、Digital Image Processing, Rafael C. Gonzalez

练习

1、Otsu 二值化有很多的优化方法，你可以尝试搜索实现。

目标

在本教程中：

- 用各种低通滤波器模糊图像。
- 对图像应用自定义过滤器（二维卷积）。

二维卷积(图像滤波)

与一维信号一样，图像也可以通过各种低通滤波器（LPF）、高通滤波器（HPF）等进行过滤。LPF 有助于消除噪音、模糊图像等。HPF 滤波器有助于在图像中找到边缘。

opencv 提供了函数 **cv.filter2D()**，用于将内核与图像卷积起来。作为一个例子，我们将尝试对图像进行平均过滤。5x5 平均过滤内核如下：

$$ 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}

$$

 图片

操作如下：将该内核保持在一个像素之上，将该内核下面的所有 25 个像素相加，取其平均值，并用新的平均值替换中心像素。它继续对图像中的所有像素执行此操作。尝试此代码并检查结果：

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('opencv_logo.png')
kernel = np.ones((5,5),np.float32)/25
dst = cv.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()
```

结果如下：

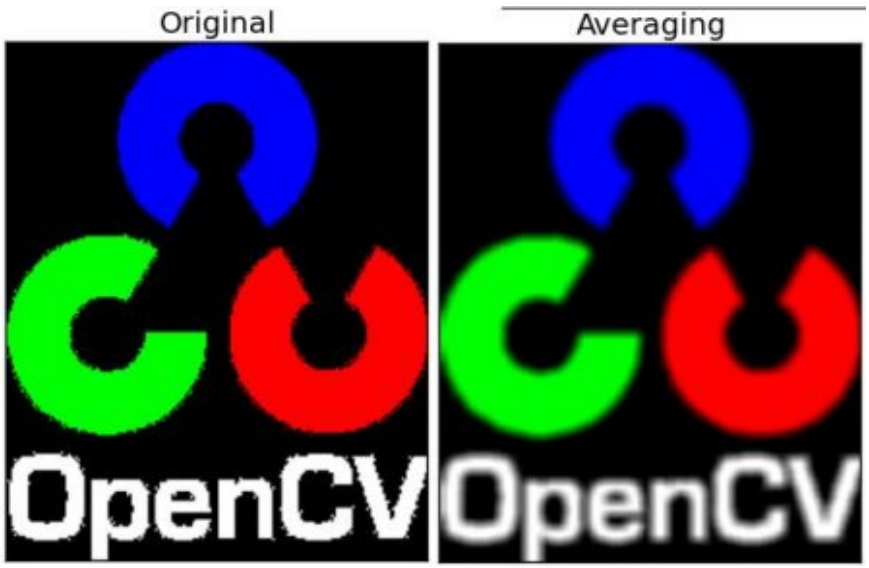
 图片

图像模糊（图像平滑）

图像模糊是通过将图像与低通滤波核卷积来实现的。它有助于消除噪音。它实际上从图像中删除高频内容（例如：噪声、边缘）。所以在这个操作中边缘有点模糊。（好吧，有一些模糊技术不会使边缘太模糊）。OpenCV 主要提供四种模糊技术。

1、平均化

这是通过用一个标准化的框过滤器卷积图像来完成的。它只需取内核区域下所有像素的平均值并替换中心元素。这是通过函数 **cv.blur()**或 **cv.boxFilter()**完成的。有关内核的更多详细信息，请查看文档。我们应该指定内核的宽度和高度。3x3 标准化框过滤器如下所示：



> **注意** 如果你不用标准化滤波，使用 **cv.boxFilter()**，传入 normalize=False 参数。

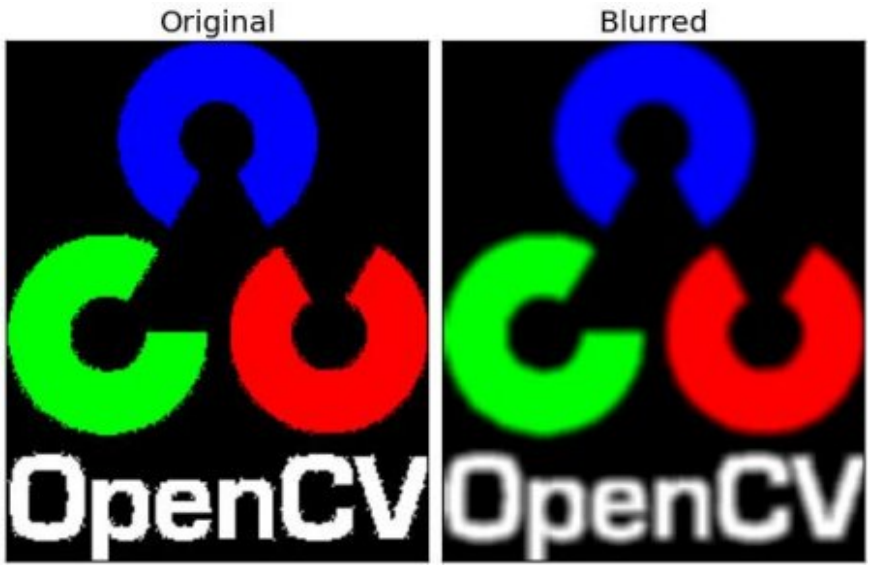
5*5 核的简单应用如下所示:

```
mport cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
img = cv.imread('opencv-logo-white.png')
blur = cv.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:

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

$$



2、高斯模糊

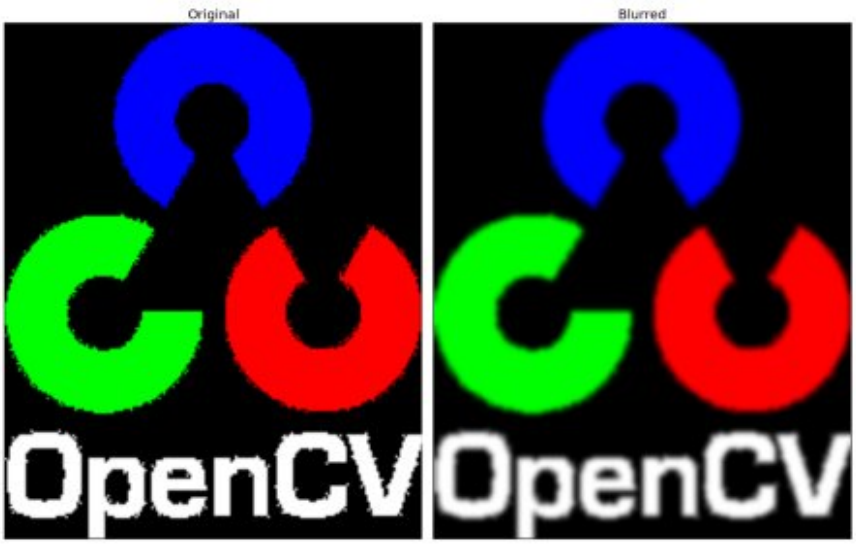
在这种情况下，使用高斯核代替了核滤波器。它是通过函数 **cv.GaussianBlur()**完成的。我们应该指定内核的宽度和高度，它应该是正数和奇数。我们还应该分别指定 x 和 y 方向的标准偏差、sigmax 和 sigmay。如果只指定 sigmax，则 sigmay 与 sigmax 相同。如果这两个值都是 0，那么它们是根据内核大小计算出来的。高斯模糊是消除图像高斯噪声的有效方法。

如果需要，可以使用函数 **cv.getGaussianKernel()**创建高斯内核。

上述代码可以修改为高斯模糊：

```
blur = cv.GaussianBlur(img,(5,5),0)
```

结果:



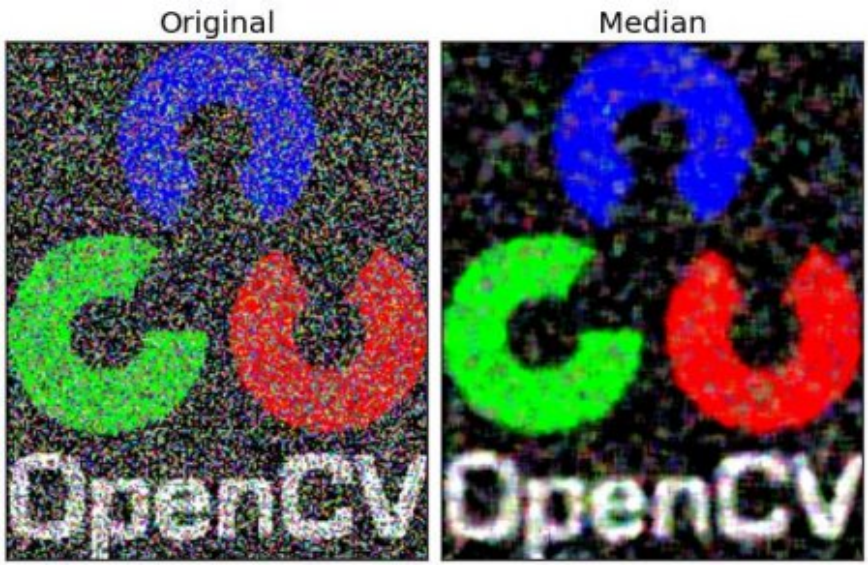
3、中值滤波

在这里，函数 cv.medianBlur()取内核区域下所有像素的中值，中央元素替换为该中值。这对图像中的椒盐噪声非常有效。有趣的是，在上面的过滤器中，中心元素是一个新计算的值，它可能是图像中的像素值，也可能是一个新值。但在中值模糊中，中心元素总是被图像中的一些像素值所取代。有效降低噪音。它的内核大小应该是一个正的奇数整数。

在这个演示中，我在原始图像中添加了 50%的噪声，并应用了中间模糊。结果如下：

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

结果:



4、双边滤波

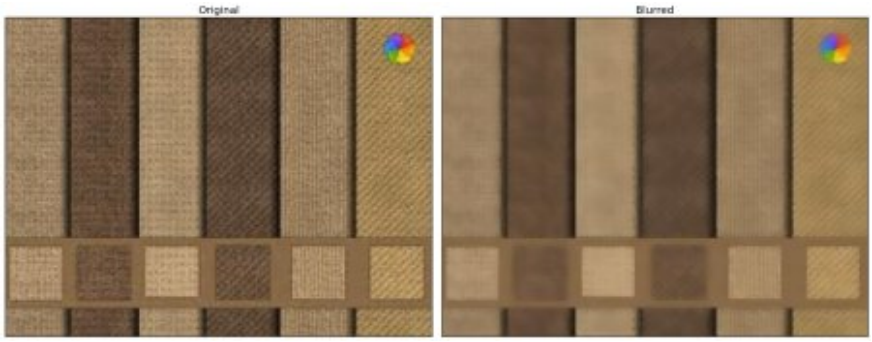
cv.bilateralFilter()在保持边缘锐利的同时，对噪声去除非常有效。但与其他过滤器相比，操作速度较慢。我们已经看到高斯滤波器取像素周围的邻域并找到其高斯加权平均值。该高斯滤波器是一个空间函数，即在滤波时考虑相邻像素。它不考虑像素是否具有几乎相同的强度。它不考虑像素是否是边缘像素。所以它也会模糊边缘，这是我们不想做的。

双边滤波器在空间上也采用高斯滤波器，而另一个高斯滤波器则是像素差的函数。空间的高斯函数确保模糊只考虑邻近像素，而强度差的高斯函数确保模糊只考虑与中心像素强度相似的像素。所以它保留了边缘，因为边缘的像素会有很大的强度变化。

下面的示例显示使用双边滤波（有关参数的详细信息，请访问文档）。

```
blur = cv.bilateralFilter(img,9,75,75)
```

结果:



其他资源

更多关于 双边滤波

练习

目标

在本教程中：

- 我们将学习不同的形态操作，如腐蚀、扩张、开、闭等。
- 我们将看到不同的函数，如： cv.erode()、 cv.dilate()、 cv.morphologyEx() 等。

理论

形态转换是基于图像形状的一些简单操作。它通常在二进制图像上执行。它需要两个输入，一个是我们的原始图像，第二个是决定操作性质的结构元素或内核。两个基本的形态学操作是腐蚀和扩张。接下来如开，闭，梯度等也会介绍。在下图的帮助下，我们将逐一看到它们：

1、腐蚀

侵蚀的基本概念就像土壤侵蚀一样，只侵蚀前景对象的边界（总是尽量保持前景为白色）。那它有什么作用呢？内核在图像中滑动（如二维卷积）。只有当内核下的所有像素都为 1 时，原始图像中的像素（1 或 0）才会被视为 1，否则会被侵蚀（变为零）。

所以根据内核的大小，边界附近的所有像素都将被丢弃。因此，前景对象的厚度或大小在图像中减少或只是白色区域减少。它有助于消除小的白色噪音（如我们在“颜色空间”一章中所看到的），分离两个连接的对象等。

作为一个例子，我将使用一个 5x5 内核，里面有很多内核。让我们看看它是如何工作的：

```
mport cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
img = cv.imread('opencv-logo-white.png')
blur = cv.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：

2、扩张

它与侵蚀正好相反。这里，如果内核下至少有一个像素为“1”，则像素元素为“1”。所以它会增加图像中的白色区域，或者增加前景对象的大小。通常情况下，在去除噪音的情况下，腐蚀后会膨胀。因为，腐蚀消除了白噪声，但它也缩小了我们的对象。所以我们扩大它。由于噪音消失了，它们不会再回来，但我们的目标区域会增加。它还可用于连接对象的断开部分。

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

结果:

3、开

开只是腐蚀的另一个名称，随后是膨胀。正如我们上面所解释的，它对消除噪音很有用。在这里，我们使用 **cv.morphologyEx()**。

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

结果:

4、闭

关闭与打开相反，膨胀后腐蚀。它在关闭前景对象内的小孔或对象上的小黑点时很有用。

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

结果:

5、形态梯度

它与图像的膨胀和腐蚀相反。

结果将类似于对象的轮廓。

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

结果:

6、顶帽

它与原图像和原图像的开相反。下面是 9*9 核的例子。

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

结果:

7、黑帽

它与原图像和原图像的闭相反。

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

结果:

结构参量

在前面的例子中，我们在 numpy 的帮助下手工创建了一个结构参量。它是长方形的。但在某些情况下，您可能需要椭圆/圆形的内核。因此，opencv 有一个函数，**cv.getStructuringElement()**。只要传递内核的形状和大小，就可以得到所需的内核。

```
# Rectangular Kernel
>>> cv.getStructuringElement(cv.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
>>> cv.getStructuringElement(cv.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
>>> cv.getStructuringElement(cv.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)
```

其他资源

在 HIPR2 的 形态学操作

练习

目标

在本教程中：

- 你会学到如何找到图像的梯度，边缘等。
- 你会学到如下函数：**cv.Sobel()**，**cv.Scharr()**，**cv.Laplacian()**等。

理论

OpenCv 提供三种类型的梯度滤波器或高通滤波器，Sobel、Scharr 和 Laplacian。我们会逐步介绍。

1、Sobel 和 Scharr 微分

Sobel 算子是一种联合高斯平滑加微分运算，因此对噪声的抵抗能力更强。可以指定要采用的导数的方向，垂直或水平（分别由参数、Yorder 和 Xorder 指定）。还可以通过参数 ksize 指定内核的大小。如果 ksize=-1，则使用 3x3 Scharr 过滤器，这比 3x3 Sobel 过滤器效果更好。请参阅所用内核的文档。

2、Laplacian 微分

它计算由关系式$\Delta src = \frac{\partial ^2{src}}{\partial x^2} + \frac{\partial ^2{src}}{\partial y^2}$给出的图像的拉普拉斯式，其中使用 Sobel 微分计算每个导数。如果 ksize=1，则使用以下内核进行筛选：

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

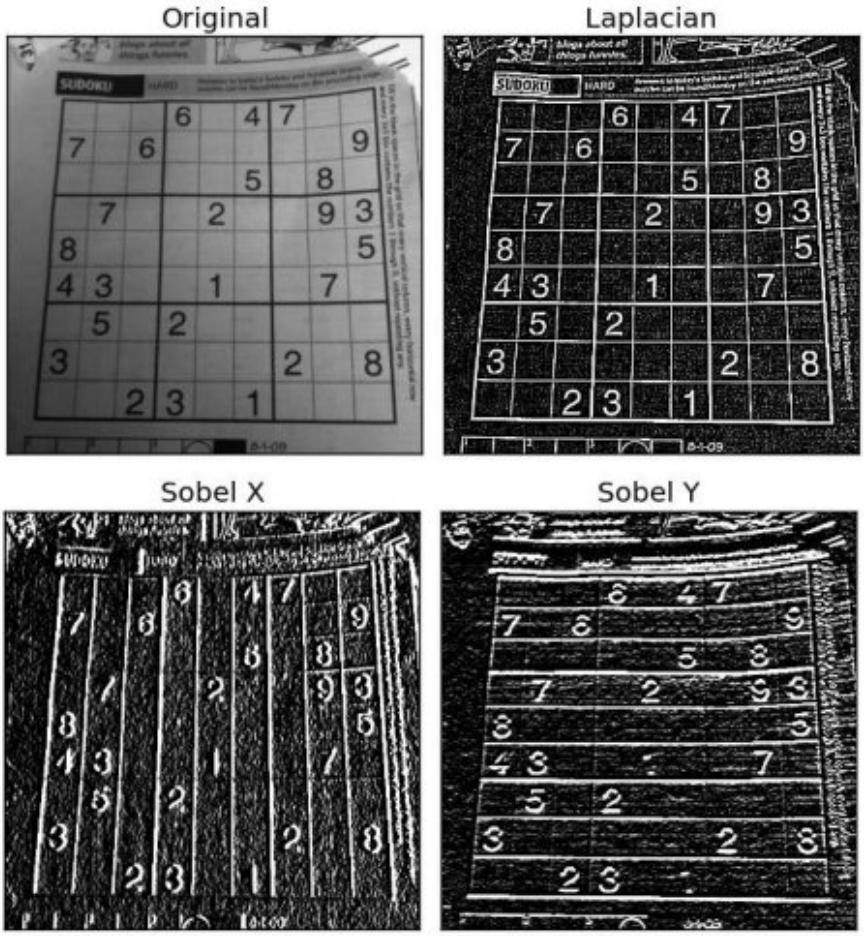
$$

代码

下面的代码显示了一个图表中的所有运算符。所有的内核都是 5x5 大小。输出图像的深度通过-1 得到 np.uint8 类型的结果。

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('dave.jpg',0)
laplacian = cv.Laplacian(img,cv.CV_64F)
sobelx = cv.Sobel(img,cv.CV_64F,1,0,ksize=5)
sobely = cv.Sobel(img,cv.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:



一件重要的事!

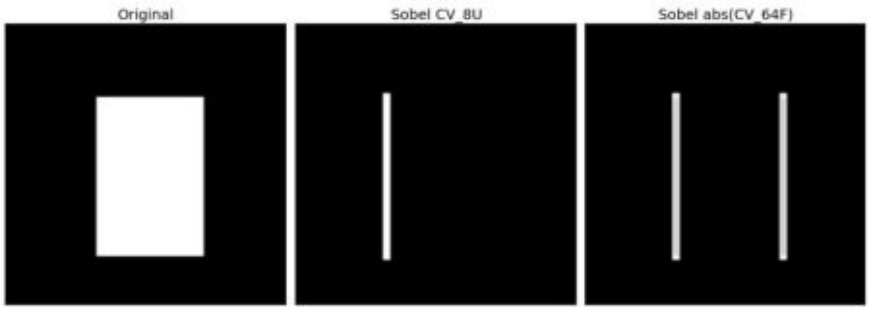
在上一个示例中，输出数据类型是 cv.cv u 或 np.uint8。但这有一个小问题。黑白过渡为正斜率（有正值），而黑白过渡为负斜率（有负值）。所以当你把数据转换成 np.uint8 时，所有的负斜率都变成零。简单来说，你失去了边缘。

如果要检测两条边，更好的选择是将输出数据类型保留为更高的格式，如 cv.cv_16s、cv.cv_64f 等，取其绝对值，然后转换回 cv.cv_8u。下面的代码演示了水平 Sobel 过滤器的此过程以及结果差异。

下面是代码:

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('box.png',0)
# Output dtype = cv.CV_8U
sobelx8u = cv.Sobel(img,cv.CV_8U,1,0,ksize=5)
# Output dtype = cv.CV_64F. Then take its absolute and convert to cv.CV_8U
sobelx64f = cv.Sobel(img,cv.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()
```

结果为：



其他资源

练习

Canny 边缘检测

目标

在本章中，我们将了解

- Canny 边缘检测的概念
- OpenCV 的功能： **cv.Canny（）**

理论

Canny 边缘检测是一种流行的边缘检测算法。它是由 John F. Canny 在 1986 年提出。

1. 这是一个多阶段算法，我们将介绍算法的每一个步骤。
2. **降噪**

   由于边缘检测易受图像中的噪声影响，因此第一步是使用 5x5 高斯滤波器去除图像中的噪声。我们在前面的章节中已经介绍到了这一点。
3. **寻找图像的强度梯度**

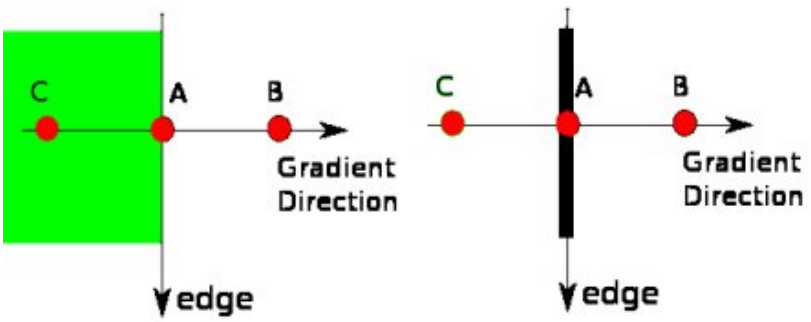
   然后在水平和垂直方向上用 Sobel 内核对平滑后的图像进行滤波，以获得水平方向（G_x）和垂直方向（G_y）的一阶导数。从这两个图像中，我们可以找到每个像素的边缘梯度和方向，如下所示：

   $$ Edge_Gradient \; (G) = \sqrt{G_x^2 + G_y^2} \ Angle \; (\theta) = \tan^{-1} \bigg(\frac{G_y}{G_x}\bigg) $$

> 渐变方向始终垂直于边缘。它被四舍五入到表示垂直，水平和两个对角线方向的四个角度中的一个。

1. **非最大抑制**

   在获得梯度幅度和方向之后，完成图像的全扫描以去除可能不构成边缘的任何不需要的像素。为此，在每个像素处，检查像素是否是其在梯度方向上的邻域中的局部最大值。检查下图：

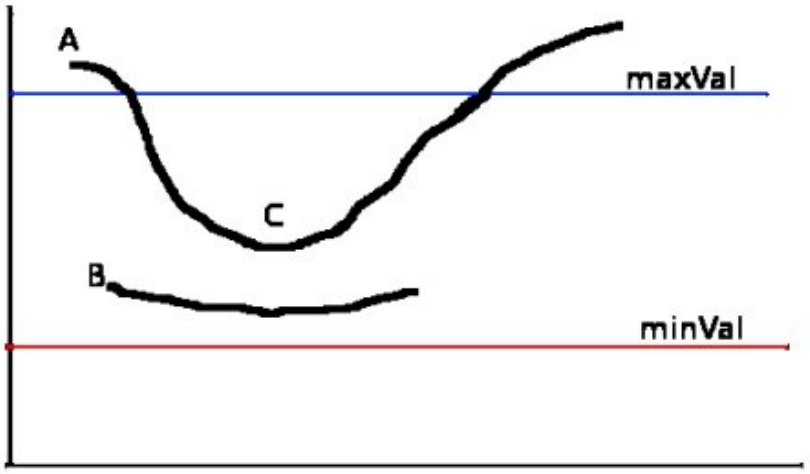


A 点位于边缘（垂直方向）。渐变方向与边缘垂直。 B 点和 C 点处于梯度方向。因此，用点 B 和 C 检查点 A，看它是否形成局部最大值。如果是这样，则考虑下一阶段，否则，它被抑制（归零）。

简而言之，您得到的结果是具有“细边”的二进制图像。

2. **滞后阈值**

这个阶段决定哪些边缘都是边缘，哪些边缘不是边缘。为此，我们需要两个阈值，minVal 和 maxVal。强度梯度大于 maxVal 的任何边缘肯定是边缘，而 minVal 以下的边缘肯定是非边缘，因此被丢弃。位于这两个阈值之间的人是基于其连通性的分类边缘或非边缘。如果它们连接到“可靠边缘”像素，则它们被视为边缘的一部分。否则，他们也被丢弃。见下图：



边缘 A 高于 maxVal，因此被视为“确定边缘”。虽然边 C 低于 maxVal，但它连接到边 A，因此也被视为有效边，我们得到完整的曲线。但是边缘 B 虽然高于 minVal 并且与边缘 C 的区域相同，但它没有连接到任何“可靠边缘”，因此被丢弃。因此，我们必须相应地选择 minVal 和 maxVal 才能获得正确的结果。

在假设边是长线的情况下，该阶段也消除了小像素噪声。

所以我们最终得到的是图像中的强边缘。

OpenCV 中的 Canny 边缘检测

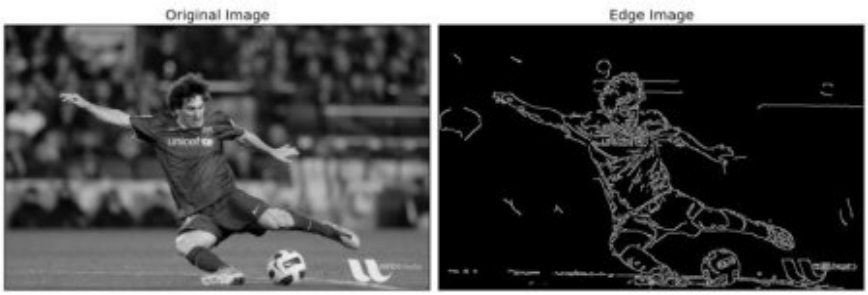
OpenCV 将以上所有内容放在单个函数中， **cv.Canny（）** 。我们将看到如何使用它。第一个参数是我们的输入图像。第二个和第三个参数分别是我们的 minVal 和 maxVal。第三个参数是 aperture_size。它是用于查找图像渐变的 Sobel 内核的大小。默认情况下，它是 3.最后一个参数是 L2gradient，它指定用于查找梯度幅度的等式。如果它是 True，它使用上面提到的更准确的等式，否则它使用这个函数：

$$ Edge_Gradient \; (G) = |G_x| + |G_y|

$$ 默认情况下，它为 False。

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('messi5.jpg',0)
edges = cv.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()
```

结果如图所示：



其他资源

1. 维基百科的 Canny 边缘探测器
2. Canny 边缘检测教程，作者 Bill Green，2002 年。

练习

1. 编写一个小应用程序来查找 Canny 边缘检测，其阈值可以使用两个轨道栏进行更改。这样，您就可以了解阈值的影响。

图像金字塔

目标

在这一章当中，

- 我们将了解图像金字塔。
- 我们将使用图像金字塔创建一个新的水果，“Orapple”
- 我们将看到这些功能： **cv.pyrUp（）** ， **cv.pyrDown（）**

理论

在通常情况下我们使用大小恒定的图像。但在某些情况下，我们需要使用同一副图像的不同分辨率的部分。例如，在搜索图像中的某些内容时，如脸部信息，并不确定该对象在所述图像中的大小。在这种情况下，我们需要创建一组具有不同分辨率的相同图像，并在所有图像中搜索对象。这些具有不同分辨率的图像被称为**图像金字塔**（因为当它们保持在堆叠中，底部具有最高分辨率图像而顶部具有最低分辨率图像时，它看起来像金字塔）。

图像金字塔有两种。 1）**高斯金字塔**和 2）**拉普拉斯金字塔**

通过去除较低级别（较高分辨率）图像中的连续行和列来形成高斯金字塔中的较高级别（低分辨率）。然后，较高级别的每个像素由来自基础级别中的 5 个像素的贡献形成，具有高斯权重。通过这样做，  图像变为  图像。因此面积减少到原始面积的四分之一。它被称为 Octave。当我们在金字塔中上升时（即分辨率降低），相同的模式继续。同样，在扩展时，每个级别的区域变为 4 次。我们可以使用 **cv.pyrDown（）** 和 **cv.pyrUp（）** 函数找到高斯金字塔。

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

以下是图像金字塔中的 4 个级别。

现在你可以用 **cv.pyrUp（）** 函数处理图像金字塔。

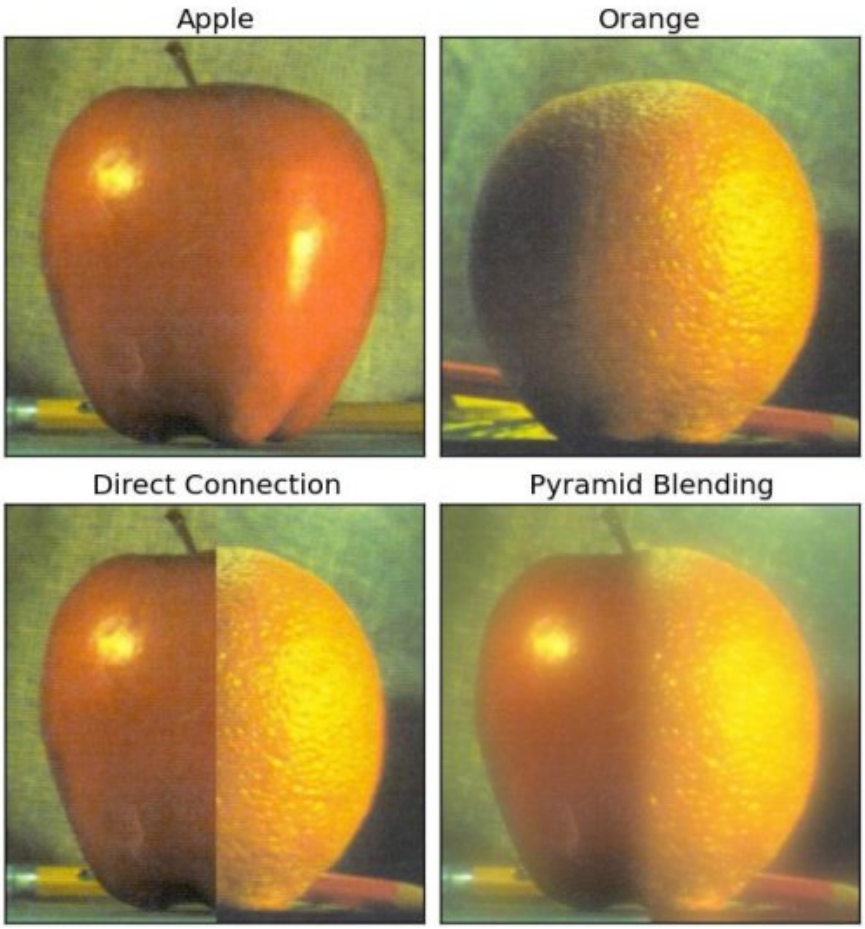
```
higher_reso2 = cv.pyrUp(lower_reso)
```

请记住，`higher_reso2` 不等于 `higher_reso`，因为一旦降低了分辨率，就会丢失信息。下图是从前一种情况下的最小图像创建的金字塔下 3 级。将其与原始图像进行比较：

拉普拉斯金字塔由高斯金字塔形成。没有专属功能。拉普拉斯金字塔图像仅与边缘图像相似。它的大部分元素都是零。它们用于图像压缩。拉普拉斯金字塔中的一个层次由高斯金字塔中的该层次与高斯金字塔中的上层的扩展版本之间的差异形成。拉普拉斯级别的三个级别如下所示（调整对比度以增强内容）：

使用金字塔的图像混合

金字塔的一个应用是图像混合。例如，在图像拼接中，您需要将两个图像堆叠在一起，但由于图像之间的不连续性，它可能看起来不太好。在这种情况下，使用金字塔进行图像混合可以实现无缝混合，而不会在图像中留下太多数据。其中一个典型的例子是混合了两种水果，橙子和苹果。现在查看结果以了解我在说什么：



请在其他资源中查看第一个参考资料，它有关于图像混合，拉普拉斯金字塔等的完整图表细节。简单的表述如下：

1. 加载苹果和橙色的两个图像
2. 找到苹果和橙色的高斯金字塔（在这个例子中，级别数是 6）
3. 从高斯金字塔，找到他们的拉普拉斯金字塔
4. 现在加入左半部分的苹果和右半部分的拉普拉斯金字塔
5. 最后，从这个联合图像金字塔，重建原始图像。

以下是完整的代码。（为简单起见，每个步骤都单独完成，由此可能需要更多内存。读者也可以对其进行优化）。

```
import cv2 as cv
import numpy as np,sys
A = cv.imread('apple.jpg')
B = cv.imread('orange.jpg')
# generate Gaussian pyramid for A
G = A.copy()
gpA = [G]
for i in xrange(6):
    G = cv.pyrDown(G)
    gpA.append(G)
# generate Gaussian pyramid for B
G = B.copy()
gpB = [G]
for i in xrange(6):
    G = cv.pyrDown(G)
    gpB.append(G)
# generate Laplacian Pyramid for A
lpA = [gpA[5]]
for i in xrange(5,0,-1):
    GE = cv.pyrUp(gpA[i])
    L = cv.subtract(gpA[i-1],GE)
    lpA.append(L)
# generate Laplacian Pyramid for B
lpB = [gpB[5]]
for i in xrange(5,0,-1):
    GE = cv.pyrUp(gpB[i])
    L = cv.subtract(gpB[i-1],GE)
    lpB.append(L)
# Now add left and right halves of images in each level
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_ = cv.pyrUp(ls_)
    ls_ = cv.add(ls_, LS[i])
# image with direct connecting each half
real = np.hstack((A[:,:cols/2],B[:,cols/2:]))
cv.imwrite('Pyramid_blending2.jpg',ls_)
cv.imwrite('Direct_blending.jpg',real)
```

其他资源

1. 图像混合

练习

OpenCV 中的轮廓

此章节在Opencv 文档中 属于大章节, 有细分章节,为了不调整结构, 将章节内容全部加入此章节内容中

轮廓：入门

学习查找和绘制轮廓

目标

- 了解轮廓是什么。
- 学习寻找轮廓，绘制轮廓等
- 您将看到以下功能：**cv.findContours()** ，**cv.drawContours()**

什么是轮廓?

轮廓可以简单地解释为连接具有相同颜色或强度的所有连续点(沿边界)的曲线。轮廓是用于形状分析以及对象检测和识别的有用工具。

- 为了获得更高的准确性，请使用二进制图像。因此，在找到轮廓之前，请应用阈值或 **Canny** 边缘检测。
- 从OpenCV 3.2开始，**findContours()** 不再修改源图像。
- 在OpenCV中，找到轮廓就像从黑色背景中找到白色物体。因此请记住，要找到的对象应该是白色，背景应该是黑色。

让我们看看如何找到二进制图像的轮廓：

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

看到，在 **cv.findContours()** 函数中有三个参数，第一个是源图像，第二个是轮廓检索模式，第三个是轮廓逼近方法。并输出轮廓和层次。轮廓是图像中所有轮廓的Python列表。每个单独的轮廓都是对象边界点的(x，y)坐标的Numpy数组。

> **注意** 稍后我们将详细讨论第二和第三论点以及有关层次结构。在此之前，代码示例中赋予它们的值将对所有图像都适用。

如何绘制轮廓?

要绘制轮廓，请使用 **cv.drawContours()** 函数。只要有边界点，它也可以用来绘制任何形状。它的第一个参数是源图像，第二个参数是应该作为Python列表传递的轮廓，第三个参数是轮廓的索引(在绘制单个轮廓时很有用。要绘制所有轮廓，请传递-1)，其余参数是颜色，厚度等等

- 要在图像中绘制所有轮廓：

```
cv.drawContours(img, contours, -1, (0,255,0), 3)
```

- 要绘制单个轮廓，请说第四个轮廓：

```
cv.drawContours(img, contours, 3, (0,255,0), 3)
```

- 但是在大多数情况下，以下方法会很有用：

```
cnt = contours[4]
cv.drawContours(img, [cnt], 0, (0,255,0), 3)
```

> **注意** 最后两种方法相同，但是前进时，您会发现最后一种更有用。

轮廓近似法

这是 **cv.findContours()** 函数中的第三个参数。它实际上表示什么？

上面我们告诉我们轮廓是强度相同的形状的边界。它存储形状边界的(x，y)坐标。但是它存储所有坐标吗？这是通过这种轮廓近似方法指定的。

如果传递 **cv.CHAIN_APPROX_NONE** ，则会存储所有边界点。但是实际上我们需要所有这些要点吗？例如，您找到了一条直线的轮廓。您是否需要线上的所有点来代表该线？不，我们只需要该线的两个端点即可。这就是 **cv.CHAIN_APPROX_SIMPLE** 所做的。它删除所有冗余点并压缩轮廓，从而节省内存。

下面的矩形图像演示了此技术。只需在轮廓数组中的所有坐标上绘制一个圆(以蓝色绘制)。第一个图像显示点I与得到 **cv.CHAIN_APPROX_NONE** (734点)和第二图像显示了一个与 **cv.CHAIN_APPROX_SIMPLE** (仅4个点)。看，它可以节省多少内存！！！

轮廓特征

学习找到轮廓的不同特征，如区域，周长，边界矩形等。

目标

在本文中，我们将学习

- 查找轮廓的不同特征，例如面积，周长，质心，边界框等
- 您将看到大量与轮廓有关的功能。

1. 矩

图像矩可帮助您计算某些特征，例如物体的重心，物体的面积等。请查看“图像矩”上的Wikipedia页面

函数 **cv.moments()** 提供了所有计算出的矩值的列表。见下文：

```
import numpy as np
import cv2 as cv
img = cv.imread('star.jpg',0)
ret,thresh = cv.threshold(img,127,255,0)
contours,hierarchy = cv.findContours(thresh, 1, 2)
cnt = contours[0]
M = cv.moments(cnt)
print( M )
```

在图像矩中，您可以提取有用的数据，例如面积，质心等。质心由关系C给出 $C*x* = *frac{M*{10}}{M*{00}}$ *和 $C_y* = *frac{M*{01}}{M_{00}}$。可以按照以下步骤进行：

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

2.轮廓面积

轮廓区域由函数 **cv.contourArea()** 或从力矩 M_{00}中给出。

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

3.轮廓周长

也称为弧长。可以使用 **cv.arcLength()** 函数找到它。第二个参数指定形状是闭合轮廓(如果通过True)还是曲线。

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

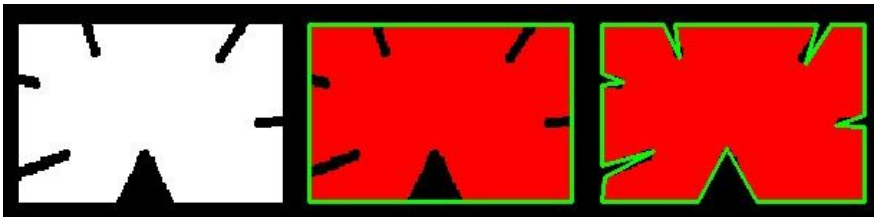
4.轮廓近似

根据我们指定的精度，它可以将轮廓形状近似为顶点数量较少的其他形状。它是Douglas-Peucker算法的实现。检查维基百科页面上的算法和演示。

为了理解这一点，假设您试图在图像中找到一个正方形，但是由于图像中的某些问题，您没有得到一个完美的正方形，而是一个“坏形状”(如下图所示)。现在，您可以使用此功能来近似形状。在这种情况下，第二个参数称为epsilon，它是从轮廓到近似轮廓的最大距离。它是一个精度参数。需要正确选择 epsilon 才能获得正确的输出。

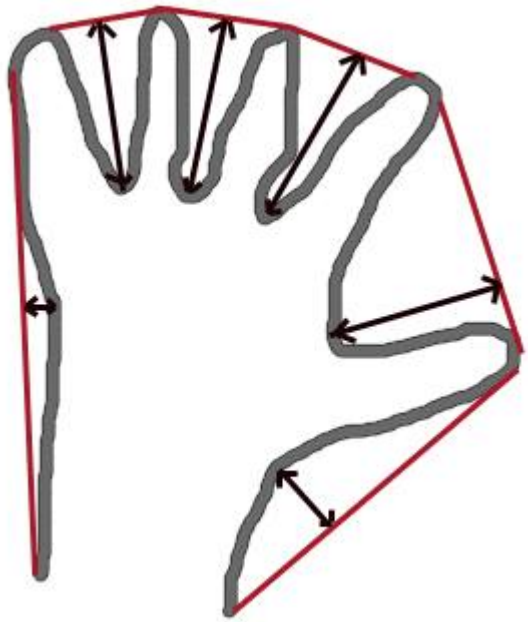
```
epsilon = 0.1*cv.arcLength(cnt,True)
approx = cv.approxPolyDP(cnt,epsilon,True)
```

下面，在第二张图片中，绿线显示了 精度 epsilon = 10% 时的近似曲线。第三幅图显示了精度 epsilon = 1% 时的情况。第三个参数指定曲线是否闭合。



5.凸包

凸包外观看起来与轮廓逼近相似，但并非如此(在某些情况下两者可能提供相同的结果)。在这里，**cv.convexHull()** 函数检查曲线是否存在凸凹缺陷并对其进行校正。一般而言，凸曲线是始终凸出或至少平坦的曲线。如果在内部凸出，则称为凸度缺陷。例如，检查下面的手的图像。红线显示手的凸包。双向箭头标记显示凸度缺陷，这是船体与轮廓线之间的局部最大偏差。



关于它的语法，有一些事情需要讨论：

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

参数详细信息：

- points: 就是我们传入的轮廓。
- hull: 是输出，通常我们避免它。
- clockwise：方向标记。如果为True，则输出凸包为顺时针方向。否则，其方向为逆时针方向。
- returnPoints：默认情况下为True。然后返回船体点的坐标。如果为False，则返回与船体点相对应的轮廓点的索引。

因此，要获得如上图所示的凸包，以下内容就足够了：

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

但是，如果要查找凸度缺陷，则需要传递 returnPoints = False。为了理解它，我们将拍摄上面的矩形图像。首先，我发现它的轮廓为cnt。现在，我发现它的带有returnPoints = True的凸包，得到以下值：[[[234 202]]，[[51 202]]，[[51 79]]，[[234 79]]]，它们是四个角矩形的点。现在，如果对returnPoints = False执行相同的操作，则会得到以下结果：[[129]，[67]，[0]，[142]]。这些是轮廓中相应点的索引。例如，检查第一个值：cnt [129] = [[234，202]]与第一个结果相同(对于其他结果依此类推)。

当我们讨论凸度缺陷时，您将再次看到它。

6.检查凸度

cv.isContourConvex() 是一个函数用来检查曲线是否为凸多边形。它只是返回True还是False。

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

7.边界矩形

有两种类型的边界矩形。

7.a. 直角矩形

它是一个直角矩形，不考虑对象的旋转。因此，边界矩形的面积将不会最小。它可以通过函数 **cv.boundingRect()** 找到。

令(x，y)为矩形的左上角坐标，而(w，h)为矩形的宽度和高度。

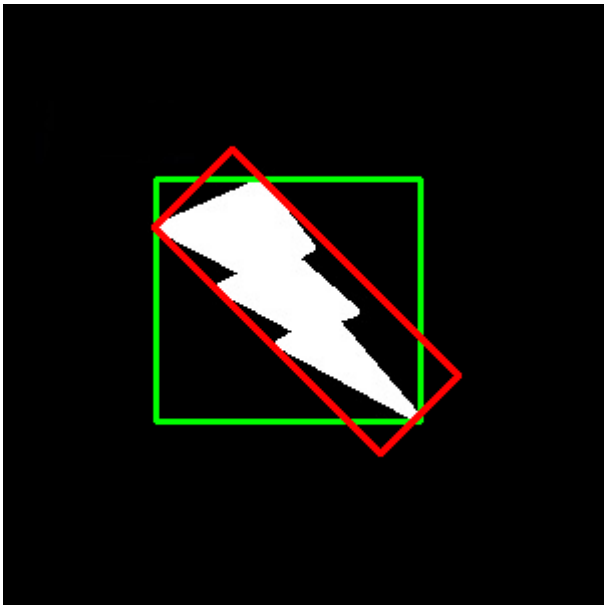
```
x,y,w,h = cv.boundingRect(cnt)
cv.rectangle(img,(x,y),(x+w,y+h),(0,255,0),2)
```

7.b. 旋转矩形

在这里，边界矩形是用最小面积绘制的，因此它也考虑了旋转。使用的函数是 **cv.minAreaRect()** 。它返回一个Box2D结构，其中包含以下细节-(中心(x，y)，(宽度，高度)，旋转角度)。但是要绘制此矩形，我们需要矩形的4个角。它是通过函数 **cv.boxPoints()** 获得的

```
rect = cv.minAreaRect(cnt)
box = cv.boxPoints(rect)
box = np.int0(box)
cv.drawContours(img,[box],0,(0,0,255),2)
```

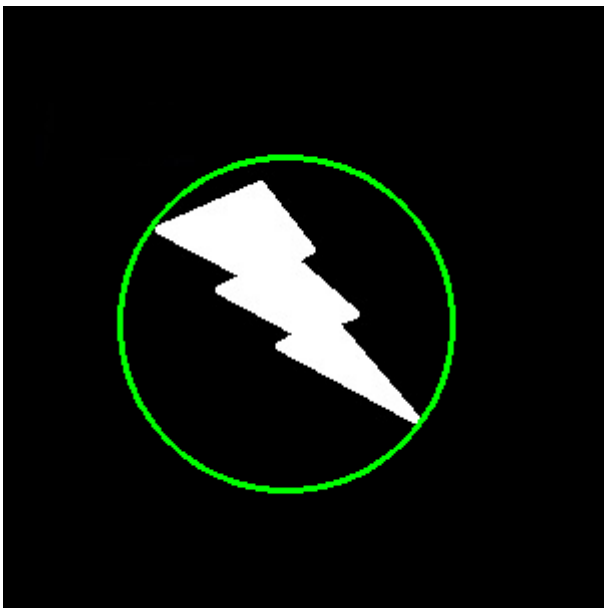
两个矩形都显示在单个图像中。绿色矩形显示法线边界矩形。红色矩形是旋转的矩形。



8.最小外圆

接下来，我们使用函数 **cv.minEnclosingCircle()** 找到对象的外接圆。它是一个以最小面积完全覆盖对象的圆圈。

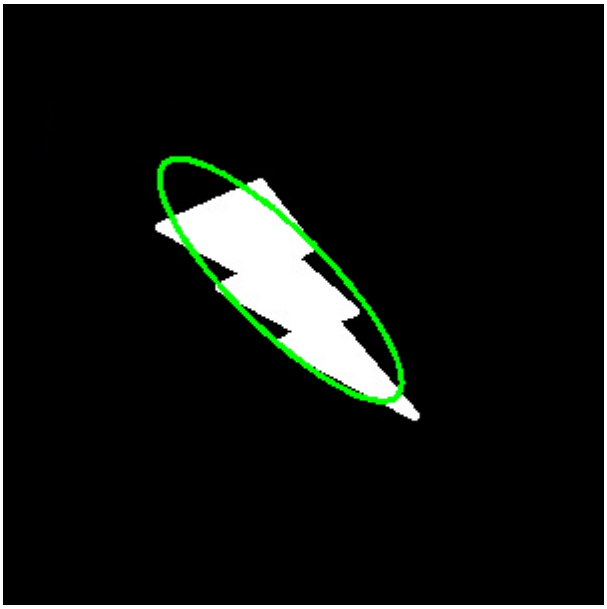
```
(x,y),radius = cv.minEnclosingCircle(cnt)
center = (int(x),int(y))
radius = int(radius)
cv.circle(img,center,radius,(0,255,0),2)
```



9.拟合椭圆

下一步是使椭圆适合对象。它返回椭圆所在的旋转矩形。

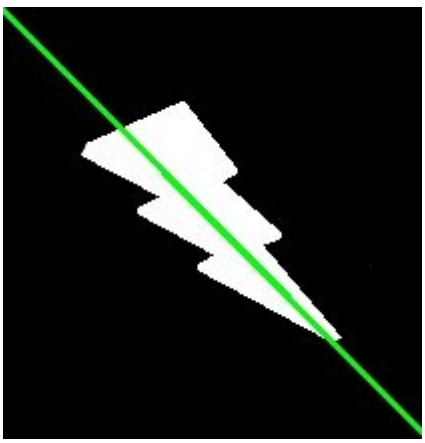
```
ellipse = cv.fitEllipse(cnt)
cv.ellipse(img,ellipse,(0,255,0),2)
```



10.拟合线

同样，我们可以将一条直线拟合到一组点。下图包含一组白点。我们可以近似一条直线。

```
rows,cols = img.shape[:2]
[vx,vy,x,y] = cv.fitLine(cnt, cv.DIST_L2,0,0.01,0.01)
lefty = int((-x*vy/vx) + y)
righty = int(((cols-x)*vy/vx)+y)
cv.line(img,(cols-1,righty),(0,lefty),(0,255,0),2)
```



轮廓属性

学习找到轮廓的不同属性，如 Solidity，Mean Intensity 等。

在这里，我们将学习提取对象的一些常用属性，例如实体，等效直径，蒙版图像，平均强度等。更多功能可以在 Matlab regionprops文档中找到。

> *(注意：质心，面积，周长等也属于此类，但我们在上一章中已经看到了)*

1.长宽比

它是对象边界矩形的宽度与高度的比率。

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

$$

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

2.范围

范围是轮廓区域与边界矩形区域的比率。

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

$$

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

3.固实性

固实性是轮廓面积与其凸包面积的比率。

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

$$

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

4.等效直径

等效直径是面积与轮廓面积相同的圆的直径。

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

$$

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

5.方向

方向是物体指向的角度。以下方法还给出了主轴和副轴的长度。

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

6.遮罩和像素点

在某些情况下，我们可能需要构成该对象的所有点。可以按照以下步骤完成：

```
mask = np.zeros(imgray.shape,np.uint8)
cv.drawContours(mask,[cnt],0,255,-1)
pixelpoints = np.transpose(np.nonzero(mask))
# pixelpoints = cv.findNonZero(mask)
```

在这里，给出了两种方法，一种使用Numpy函数，另一种使用OpenCV函数(最后注释的行)执行相同的操作。结果也相同，但略有不同。Numpy以(行，列)格式给出坐标，而OpenCV以(x，y)格式给出坐标。因此，基本上答案是可以互换的。注意，row = x，column = y。

7.最大值，最小值及其位置

我们可以使用遮罩图像找到这些参数。

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

8.平均颜色或平均强度

在这里，我们可以找到对象的平均颜色。或者可以是灰度模式下物体的平均强度。我们再次使用相同的蒙版进行此操作。

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

9.极端点

极点是指对象的最顶部，最底部，最右侧和最左侧的点。

```
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])
```

例如，如果将其应用于印度地图，则会得到以下结果：

轮廓：更多功能

学习找到凸性缺陷，pointPolygonTest，匹配不同的形状等。

目标

在本章中，我们将学习

- 凸性缺陷以及如何找到它们。
- 查找点到多边形的最短距离
- 匹配不同的形状

理论和代码

1.凸包缺陷

在第二章中，我们看到了关于轮廓的凸包。物体与该船体的任何偏离都可以视为凸包缺陷。

OpenCV带有一个现成的函数 **cv.convexityDefect()** 来查找该函数。基本的函数调用如下所示：

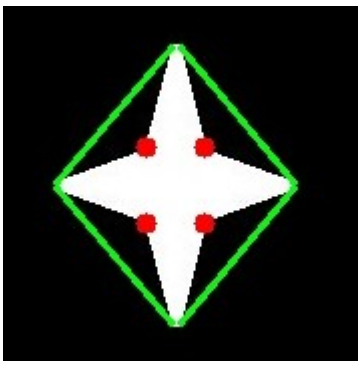
```
hull = cv.convexHull(cnt,returnPoints = False)
defects = cv.convexityDefects(cnt,hull)
```

> **注意** 请记住，在寻找凸包时，我们必须传递returnPoints = False，以便寻找凸缺陷。

它返回一个数组，其中每行包含这些值- [起点，终点，最远点，到最远点的近似距离]。我们可以使用图像对其进行可视化。我们画一条连接起点和终点的线，然后在最远的点画一个圆。请记住，返回的前三个值是cnt的索引。因此，我们必须从cnt带来这些值。

```
import cv2 as cv
import numpy as np
img = cv.imread('star.jpg')
img_gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
ret,thresh = cv.threshold(img_gray, 127, 255,0)
contours,hierarchy = cv.findContours(thresh,2,1)
cnt = contours[0]
hull = cv.convexHull(cnt,returnPoints = False)
defects = cv.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])
    cv.line(img,start,end,[0,255,0],2)
    cv.circle(img,far,5,[0,0,255],-1)
cv.imshow('img',img)
cv.waitKey(0)
cv.destroyAllWindows()
```

并查看结果：



2.点多边形测试

此功能查找图像中的点与轮廓之间的最短距离。它返回的距离为：当点在轮廓外时为负；当点在轮廓内时为正；如果点在轮廓上，则返回零。

例如，我们可以如下检查点(50,50)：

```
dist = cv.pointPolygonTest(cnt,(50,50),True)
```

在函数中，第三个参数是measureDist。如果为True，则找到带符号的距离。如果为False，它将查找该点是在轮廓内部还是外部或轮廓上(它分别返回+ 1，-1、0)。

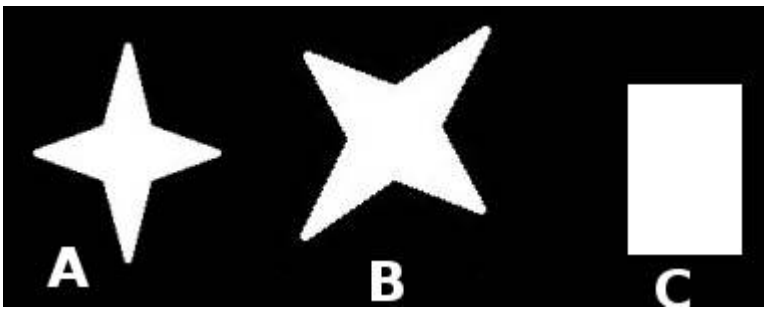
> **注意** 如果您不想查找距离，请确保第三个参数为False，因为这是一个耗时的过程。因此，将其设置为False可使速度提高2-3倍。

3.匹配形状

OpenCV带有函数 **cv.matchShapes()**，使我们能够比较两个形状或两个轮廓，并返回显示相似性的度量。结果越低，匹配越好。它是基于 **hu-moment** 值计算的。文档中介绍了不同的测量方法。

```
import cv2 as cv
import numpy as np
img1 = cv.imread('star.jpg',0)
img2 = cv.imread('star2.jpg',0)
ret, thresh = cv.threshold(img1, 127, 255,0)
ret, thresh2 = cv.threshold(img2, 127, 255,0)
contours,hierarchy = cv.findContours(thresh,2,1)
cnt1 = contours[0]
contours,hierarchy = cv.findContours(thresh2,2,1)
cnt2 = contours[0]
ret = cv.matchShapes(cnt1,cnt2,1,0.0)
print( ret )
```

我尝试匹配以下给出的不同形状的形状：



我得到以下结果：

- 匹配图像A本身= 0.0
- 将图像A与图像B匹配= 0.001946
- 将图像A与图像C匹配= 0.326911 看，即使图像旋转也不会对该比较产生太大影响。

也可以看看 Hu-Moments是平移，旋转和缩放不变的七个矩。第七个是偏斜不变的。这些值可以使用 **cv.HuMoments()** 函数找到。

练习题

1. 检查文档 **cv.pointPolygonTest()** ，您可以找到红色和蓝色的漂亮图像。它表示从所有像素到其上的白色曲线的距离。曲线内的所有像素均为蓝色，具体取决于距离。同样，外部点为红色。等高线边缘用白色标记。所以问题很简单。编写代码以创建距离的这种表示。
2. 使用 **cv.matchShapes()** 比较数字或字母的图像。(那将是迈向OCR的简单步骤)

轮廓层次

了解轮廓层次结构

目标

这次，我们了解轮廓的层次结构，即Contours中的父子关系。

理论

在有关轮廓的最后几篇文章中，我们使用了与OpenCV提供的轮廓相关的一些功能。但是，当我们使用 cv.findContours() 函数在图像中找到轮廓时，我们传递了一个参数，即 **Contour Retrieval Mode** 。我们通常通过 **cv.RETR_LIST** 或 **cv.RETR_TREE** ，效果很好。但这实际上是什么意思？

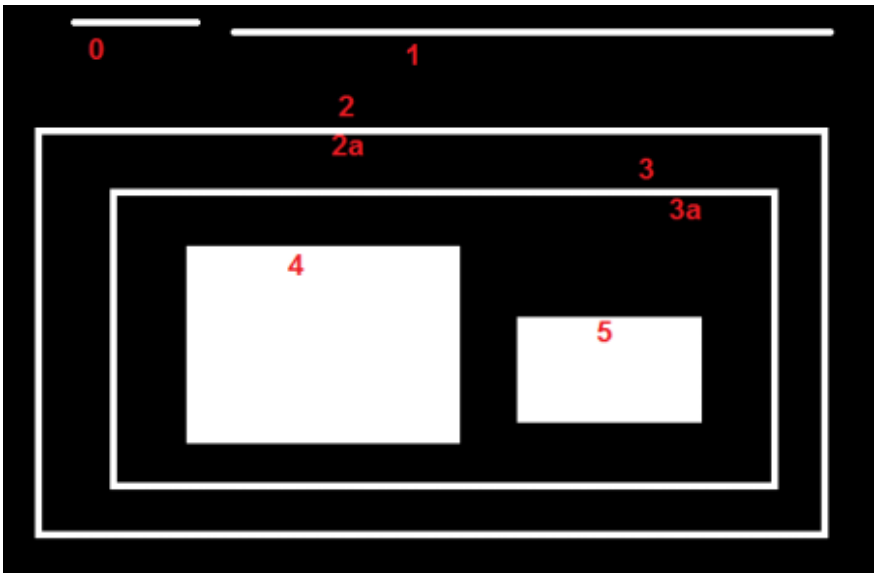
另外，在输出中，我们得到了三个数组，第一个是图像，第二个是轮廓，另一个是我们命名为层次结构的输出(请检查上一篇文章中的代码)。但是，我们从未在任何地方使用此层次结构。那么，这个层次结构是什么呢？它与前面提到的函数参数有什么关系？

这就是本文要处理的内容。

什么是层次结构?

通常我们使用 cv.findContours() 函数来检测图像中的对象，对吗？有时对象位于不同的位置。但是在某些情况下，某些形状位于其他形状内。就像嵌套的数字一样。在这种情况下，我们将外部的一个称为 **父级** ，将内部的一个称为 **子级** 。这样，图像中的轮廓彼此之间就具有某种关系。并且我们可以指定一个轮廓如何相互连接，例如是其他轮廓的子轮廓，还是父轮廓等。这种关系的表示称为 **层次结构** 。

考虑下面的示例图像：



在此图像中，我从 **0-5** 编号了一些形状。2和2a表示最外面的盒子的外部和内部轮廓。

在此，轮廓0,1,2在 **外部或最外部** 。我们可以说，它们处于 **0层次结构** 中，或者只是处于 **相同的层次结构级别** 中。

接下来是 **轮廓2a** 。可以将其视为 **轮廓2的子级** (或者相反，轮廓2是轮廓2a的父级)。因此，将其设置为 **hierarchy-1** 。同样，contour-3是contour-2的子级，位于下一个层次结构中。最后，轮廓4,5是轮廓3a的子级，它们位于最后的层次结构级别。从编号方式上来说，轮廓4是轮廓3a的第一个子元素(也可以是轮廓5)。

我提到这些东西是为了理解诸如相同的层次结构级别，外部轮廓，子轮廓，父轮廓，第一个孩子等术语。现在让我们进入OpenCV。

OpenCV中的层次结构表示

因此，每个轮廓都有关于其层次结构，其子级，其父级等的信息。OpenCV将其表示为四个值的数组：**[Next，Previous，First_Child，Parent]**

Next 表示相同等级的下一个轮廓 例如，在我们的图片中选择轮廓0。谁是同一级别的下一个轮廓？它是轮廓1。因此，只需将Next = 1放进去。同样对于Contour-1，下一个就是轮廓线2。所以下一个= 2。

那轮廓2呢？在同一层中没有下一个轮廓。简而言之，将Next = -1。那轮廓4呢？与轮廓5处于同一水平。所以它的下一个轮廓是轮廓5，所以Next = 5。

Previous 表示相同轮廓级别的上一个轮廓 和上面一样。轮廓1的先前轮廓是同一级别的轮廓0。同样对于轮廓2，它是轮廓1。对于轮廓0，没有先前值，因此将其设为-1。

First_Child 表示其第一个子轮廓 无需任何解释。对于轮廓2，子级是轮廓2a。这样就得到了轮廓2a的相应索引值。那轮廓3a呢？它有两个孩子。但是我们只带第一个孩子。它是轮廓4。因此，轮廓3a的First_Child = 4。

Parent 代表示其父代轮廓的索引 它与First_Child相反。轮廓4和轮廓5的父轮廓均为轮廓3a。对于轮廓3a，它是轮廓3，依此类推。

> **注意** 如果没有孩子或父母，则该字段为-1

因此，现在我们知道了OpenCV中使用的层次结构样式，我们可以借助上面给出的相同图像来检查OpenCV中的轮廓检索模式。即像 **cv.RETR_LIST** ，**cv.RETR_TREE** ，**cv.RETR_CCOMP**，**cv.RETR_EXTERNAL** 等标志是什么意思?

轮廓检索模式

1. RETR_LIST

这是四个标志中最简单的一个(从解释的角度来看)。它仅检索所有轮廓，但不创建任何父子关系。在这个规则下，**父母和孩子是平等的**，他们只是轮廓。即它们都属于同一层次结构级别。

因此，在这里，层次结构数组中的第3和第4项始终为-1。但是很明显，下一个和上一个术语将具有其相应的值。只需自己检查并验证即可。

以下是我得到的结果，每行是相应轮廓的层次结构详细信息。例如，第一行对应于轮廓0。下一个轮廓为轮廓1。因此Next =1。没有先前的轮廓，因此Previous = -1。如前所述，其余两个为-1。

```
>>> 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]]])
```

如果不使用任何层次结构功能，这是在代码中使用的不错选择。

2. RETR_EXTERNAL

如果使用此标志，则仅返回极端的外部标志。保留所有子轮廓。可以说，根据这项法律，只有每个家庭中的老大才能得到照顾。它不在乎家庭其他成员 :)。

那么，在我们的图像中，有多少个极端的外部轮廓？即在等级0级别？只有3个，即轮廓0,1,2，对吗？现在尝试使用该标志查找轮廓。在此，赋予每个元素的值也与上述相同。与上面的结果进行比较。以下是我得到的：

```
>>> 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]]])
```

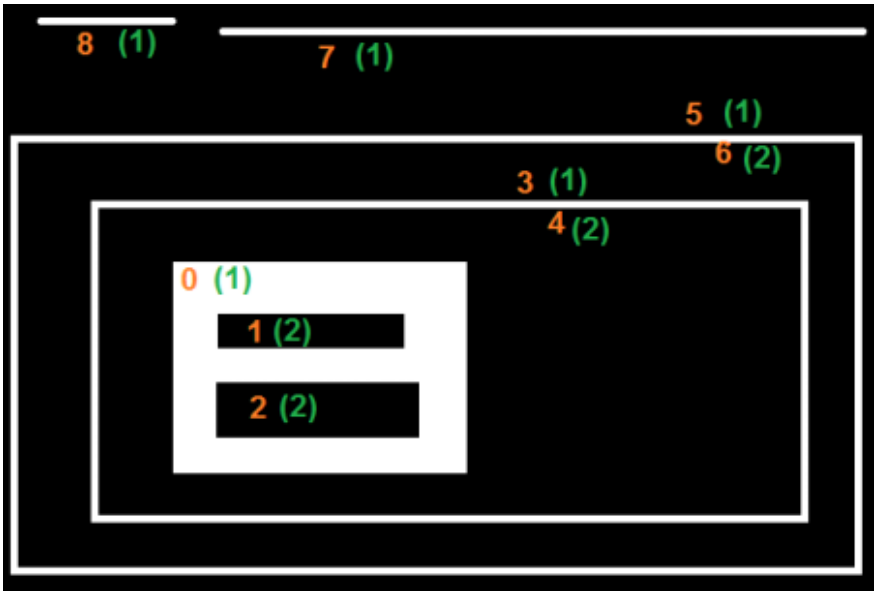
如果只想提取外部轮廓，则可以使用此标志。在某些情况下可能有用。

3. RETR_CCOMP

该标志检索所有轮廓并将它们排列为2级层次结构。即，对象的外部轮廓(即其边界)位于层次1中。然后，将对象(如果有)中的孔的轮廓放置在层次2中。如果其中有任何对象，则其轮廓将仅再次放置在等级1中。以及它在等级2中的漏洞等等。

只需考虑黑色背景上的“白色大零”图像即可。零外圈属于第一层级，零内圈属于第二层级。

我们可以用一个简单的图像来解释它。在这里，我用红色标记了轮廓的顺序，并用绿色(1或2)标记了它们所属的层次。该顺序与OpenCV检测轮廓的顺序相同。



因此考虑第一个轮廓，即轮廓0。它是等级1。它有两个孔，轮廓1和2，它们属于层次2。因此，对于轮廓0，相同层次结构级别中的下一个轮廓为轮廓3。而且没有以前的。它的第一个子对象是层次结构2中的轮廓1。它没有父级，因为它位于1层级中。因此其层次结构数组为 [3,-1,1,-1]

现在取轮廓1。它在等级2中。在同一层次结构中(轮廓1的父项下)下一个是轮廓2。没有上一个。没有孩子，但父母的轮廓为0。因此数组为 [2,-1,-1,0]。

同样的轮廓2：它位于层次2中。在轮廓-0下的相同层次结构中没有下一个轮廓。所以没有下一步。上一个是轮廓1。没有孩子，父母的轮廓为0。因此数组为 [-1,1,-1,0]。

轮廓-3：等级1中的下一个是轮廓5。上一个是轮廓0。孩子是轮廓4，没有父母。因此数组为 [5,0,4,-1]。

轮廓-4：位于轮廓3下的层次2中，并且没有同级。所以没有下一个，没有以前的，没有孩子，父母是轮廓3。因此数组为 [-1,-1,-1,3]。

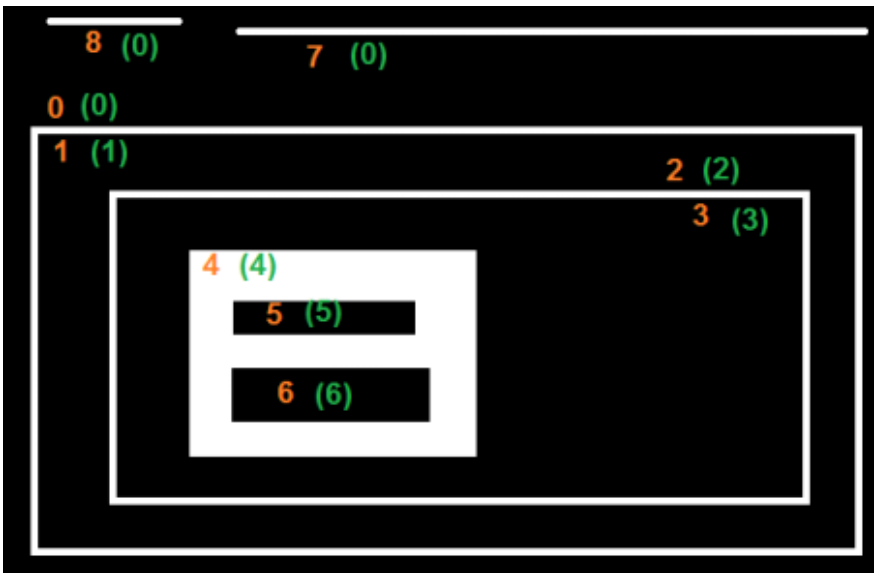
剩下的可以填满。这是我得到的最终答案：

```
>>> 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]]])
```

4. RETR_TREE

这是最后一个家伙，Perfect先生。它检索所有轮廓并创建完整的族层次列表。它甚至告诉，谁是爷爷，父亲，儿子，孙子甚至更远... :)。

例如，我拍摄了上面的图片，重写了 **cv.RETR_TREE** 的代码，根据OpenCV给定的结果对轮廓进行重新排序并对其进行分析。同样，红色字母表示轮廓编号，绿色字母表示层次结构顺序。



取轮廓0：在层次0中。同一层次结构中的下一个轮廓是轮廓7。没有先前的轮廓。孩子是轮廓1。而且没有父母。因此数组为 [7,-1,1,-1]。

取轮廓2：在等级1中。同一级别无轮廓。没有上一个。孩子是轮廓3。父级是轮廓1。因此数组为 [-1,-1,3,1]。

还有，尝试一下。以下是完整答案：

```
>>> 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]]])
```

OpenCV 中的直方图

直方图 - 1：查找，绘图，分析!

学习直方图的基础知识

目标

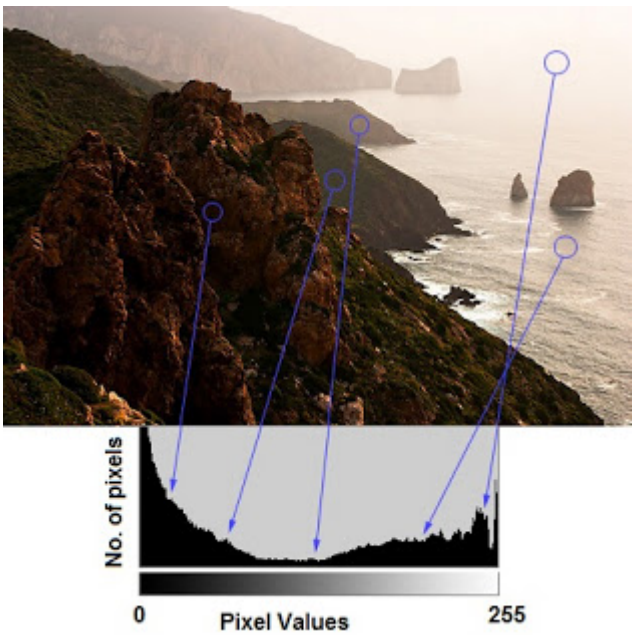
学会

- 使用OpenCV和Numpy函数查找直方图
- 使用OpenCV和Matplotlib函数绘制直方图
- 您将看到以下功能：**cv.calcHist()** ，**np.histogram()** 等。

理论

那么直方图是什么？您可以将直方图视为图形或曲线图，从而使您对图像的强度分布有一个整体的了解。它是在X轴上具有像素值(不总是从0到255的范围)，在Y轴上具有图像中相应像素数的图。

这只是理解图像的另一种方式。通过查看图像的直方图，您可以直观地了解该图像的对比度，亮度，强度分布等。当今几乎所有图像处理工具都提供直方图功能。以下是 剑桥彩色网站 上的图片，建议您访问该网站以获取更多详细信息。



您可以看到图像及其直方图。(请记住，此直方图是针对灰度图像而非彩色图像绘制的)。直方图的左侧区域显示图像中较暗像素的数量，而右侧区域则显示较亮像素的数量。从直方图中，您可以看到暗区域多于亮区域，中间值的数量(中间值的像素值，例如127附近)非常少。

查找直方图

现在我们有了一个关于直方图的想法，我们可以研究如何找到它。OpenCV和Numpy都为此内置了功能。在使用这些功能之前，我们需要了解一些与直方图有关的术语。

BINS： 上面的直方图显示每个像素值的像素数，即从0到255。即，您需要256个值来显示上面的直方图。但是考虑一下，如果您不需要分别找到所有像素值的像素数，而是找到像素值间隔中的像素数怎么办？例如，您需要找到介于0到15之间，然后16到31之间，...，240到255之间的像素数。您只需要16个值即可表示直方图。这就是在 OpenCV直方图教程 中给出的示例中所显示的内容。

因此，您要做的就是将整个直方图分成16个子部分，每个子部分的值就是其中所有像素数的总和。每个子部分都称为“BIN”。在第一种情况下，bin的数量为256个(每个像素一个)，而在第二种情况下，bin的数量仅为16个。BINS由OpenCV文档中的**histSize** 术语表示。

DIMS： 这是我们为其收集数据的参数的数量。在这种情况下，我们仅收集关于强度值的一件事的数据。所以这里是1。

范围： 这是您要测量的强度值的范围。通常，它是[0,256]，即所有强度值。

1. OpenCV中的直方图计算

因此，现在我们使用 **cv.calcHist()** 函数查找直方图。让我们熟悉一下函数及其参数：

$$ cv.calcHist(images, channels, mask, histSize, ranges[, hist[, accumulate]])

$$

1. images：它是uint8或float32类型的源图像。它应该放在方括号中，即“[img]”。
2. channels：也以方括号给出。它是我们计算直方图的通道的索引。例如，如果输入为灰度图像，则其值为[0]。对于彩色图像，您可以传递[0]，[1]或[2]分别计算蓝色，绿色或红色通道的直方图。
3. mask：遮罩图像。为了找到完整图像的直方图，将其指定为“无”。但是，如果要查找图像特定区域的直方图，则必须为此创建一个遮罩图像并将其作为遮罩。(我将在后面显示一个示例。)
4. histSize：这表示我们的BIN计数。需要放在方括号中。对于全尺寸，我们通过[256]。
5. ranges：这是我们的RANGE。通常为[0,256]。 因此，让我们从示例图像开始。只需在灰度模式下加载图像并找到其完整的直方图即可。

```
img = cv.imread('home.jpg',0)
hist = cv.calcHist([img],[0],None,[256],[0,256])
```

hist是256x1的数组，每个值对应于该图像中具有相应像素值的像素数。

2. Numpy中的直方图计算

Numpy还为您提供了一个函数 **np.histogram()** 。因此，您可以在下面的行尝试代替 **calcHist()** 函数：

```
hist,bins = np.histogram(img.ravel(),256,[0,256])
```

hist与我们之前计算的相同。但是bin将具有257个元素，因为Numpy计算出bin的范围为0-0.99、1-1.99、2-2.99等。因此最终范围为255-255.99。为了表示这一点，他们还在料箱末端添加了256。但是我们不需要256。最多255就足够了。

也可以看看 Numpy还有另一个函数 **np.bincount()** ，它比np.histogram()快10倍左右。因此，对于一维直方图，您可以更好地尝试一下。不要忘记在np.bincount中设置minlength = 256。例如，hist = np.bincount(img.ravel()，minlength = 256)

> **注意** OpenCV函数比 np.histogram() 快(大约40倍)。因此，请坚持使用OpenCV功能。

现在我们应该绘制直方图，但是如何?

绘制直方图

有两种方法，

1. 简短方法：使用Matplotlib绘图功能
2. 很长的路要走：使用OpenCV绘图功能

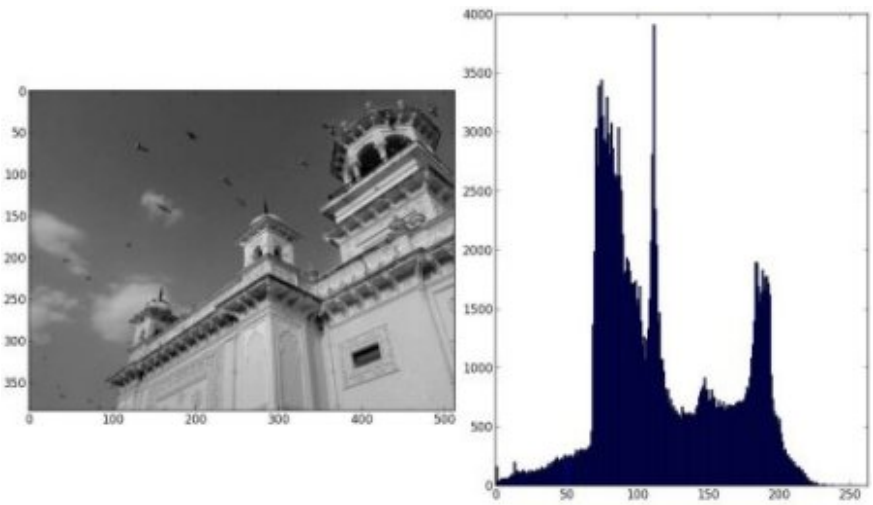
1.使用Matplotlib

Matplotlib带有直方图绘图功能：matplotlib.pyplot.hist()

它直接找到直方图并将其绘制。您无需使用 **calcHist()** 或 np.histogram() 函数来查找直方图。请参见下面的代码：

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('home.jpg',0)
plt.hist(img.ravel(),256,[0,256]); plt.show()
```

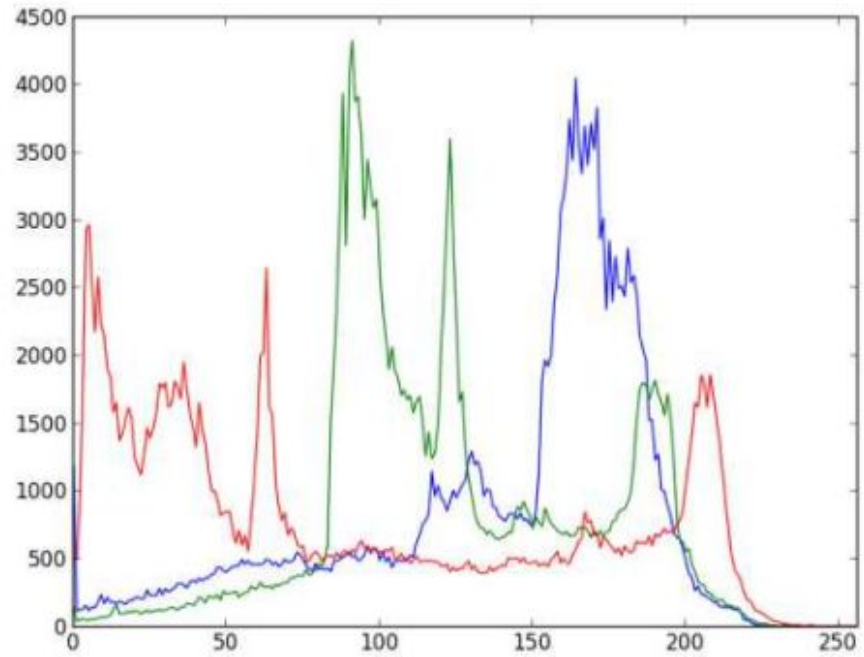
您将得到如下图：



或者，您可以使用matplotlib的法线图，这对于BGR图是很好的。为此，您需要首先找到直方图数据。试试下面的代码：

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('home.jpg')
color = ('b','g','r')
for i,col in enumerate(color):
    histr = cv.calcHist([img],[i],None,[256],[0,256])
    plt.plot(histr,color = col)
    plt.xlim([0,256])
plt.show()
```

结果：



]

您可以从上图中得出，蓝色在图像中具有一些高值区域(显然这应该是由于天空)

2.使用OpenCV

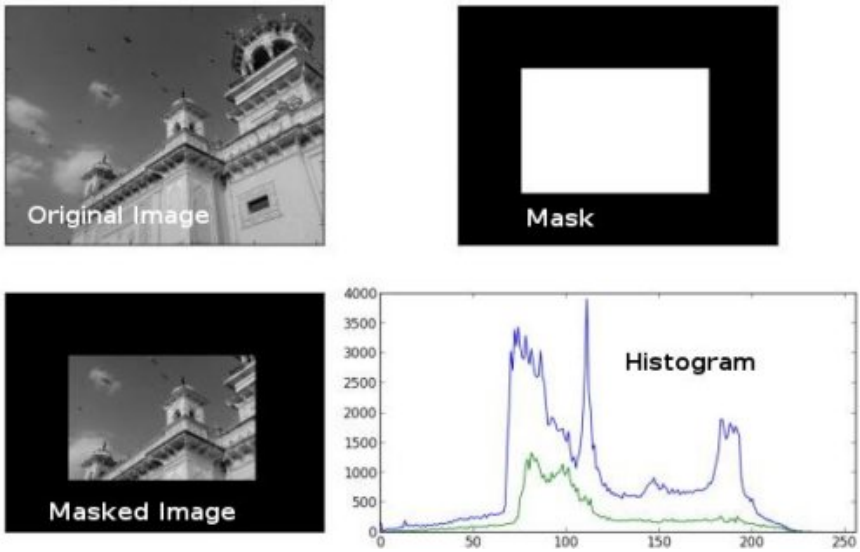
好吧，在这里您可以调整直方图的值及其bin值，使其看起来像x，y坐标，以便可以使用 **cv.line()** 或 cv.polyline() 函数绘制它以生成与上述相同的图像。OpenCV-Python2官方示例已经提供了此功能。检查 samples/python/hist.py 代码

遮罩的应用

我们使用 **cv.calcHist()** 查找完整图像的直方图。如果要查找图像某些区域的直方图怎么办？只需在要查找直方图的区域上创建白色的蒙版图像，否则创建黑色。然后通过这个作为面具。

```
img = cv.imread('home.jpg',0)
# create a mask
mask = np.zeros(img.shape[:2], np.uint8)
mask[100:300, 100:400] = 255
masked_img = cv.bitwise_and(img,img,mask = mask)
# Calculate histogram with mask and without mask
# Check third argument for mask
hist_full = cv.calcHist([img],[0],None,[256],[0,256])
hist_mask = cv.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()
```

查看结果。在直方图中，蓝线表示完整图像的直方图，绿线表示遮蔽区域的直方图。



其他资源

1. 剑桥彩色网站

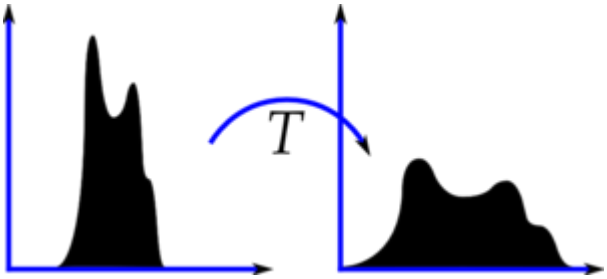
直方图 - 2：直方图均衡

学会均衡直方图以获得更好的图像对比度

目标

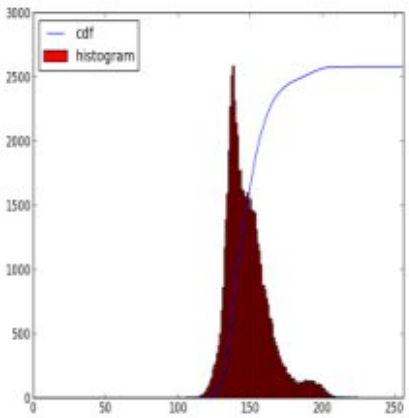
在这个部分，

我们将学习直方图均衡化的概念，并将其用于改善图像的对比度。 理论 考虑一个图像，其像素值仅限于特定的值范围。例如，较亮的图像会将所有像素限制在较高的值。但是，好的图像将具有来自图像所有区域的像素。因此，您需要将此直方图拉伸到两端(如下图所示，来自维基百科)，这就是直方图均衡化的作用(简单来说)。通常，这可以提高图像的对比度。



我建议您阅读 直方图均衡化 上的Wikipedia页面，以获取有关它的更多详细信息。它很好地解释了示例，使您在阅读完之后几乎可以理解所有内容。相反，在这里我们将看到其Numpy实现。之后，我们将看到OpenCV功能。

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('wiki.jpg',0)
hist,bins = np.histogram(img.flatten(),256,[0,256])
cdf = hist.cumsum()
cdf_normalized = cdf * float(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()
```



您可以看到直方图位于较亮的区域。我们需要全方位的服务。为此，我们需要一个转换函数，该函数将较亮区域中的输入像素映射到整个区域中的输出像素。这就是直方图均衡化的作用。

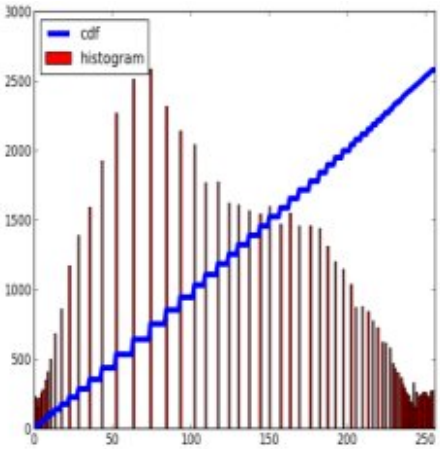
现在，我们找到最小的直方图值(不包括0)并应用Wiki页面中给出的直方图均衡方程。但是我在这里使用了Numpy的masked array概念数组。对于掩码数组，所有操作都在非掩码元素上执行。您可以从有关屏蔽数组的Numpy文档中了解有关此内容的更多信息。

```
cdf_m = np.ma.masked_equal(cdf,0)
cdf_m = (cdf_m - cdf_m.min())*255/(cdf_m.max()-cdf_m.min())
cdf = np.ma.filled(cdf_m,0).astype('uint8')
```

现在我们有了查找表，该表为我们提供了有关每个输入像素值的输出像素值是什么的信息。因此，我们仅应用变换。

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

现在我们像以前一样计算它的直方图和cdf(您这样做)，结果如下所示：



另一个重要特征是，即使图像是较暗的图像(而不是我们使用的较亮的图像)，在均衡后，我们将获得与获得的图像几乎相同的图像。结果，它被用作“参考工具”，以使所有图像具有相同的照明条件。在许多情况下这很有用。例如，在人脸识别中，在训练人脸数据之前，将人脸图像进行直方图均衡，以使它们全部具有相同的光照条件。

OpenCV中的直方图均衡

OpenCV具有执行此操作的功能 **cv.equalizeHist()** 。它的输入只是灰度图像，输出是我们的直方图均衡图像。

下面是一个简单的代码片段，显示了它与我们使用的同一图像的用法：

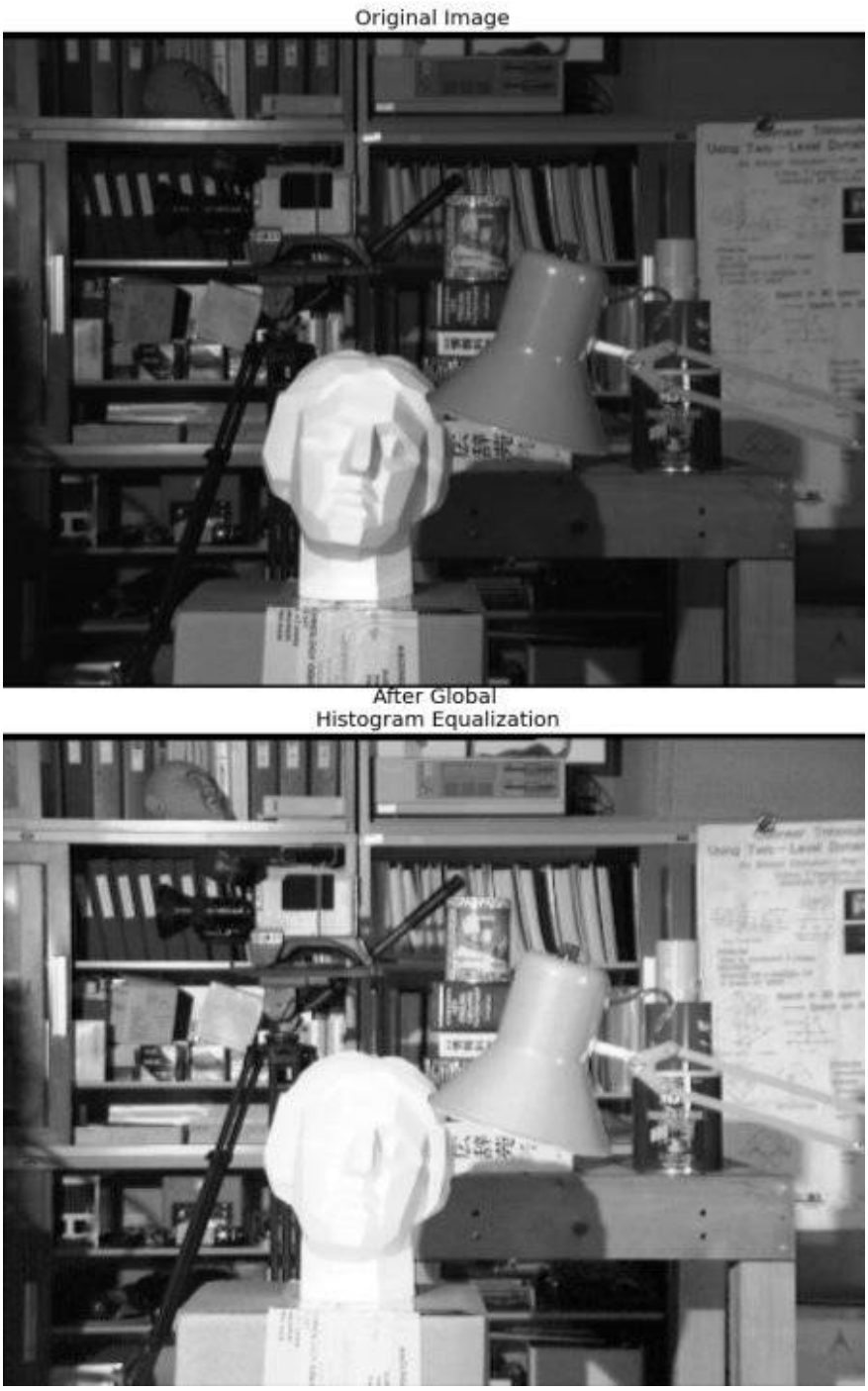
```
img = cv.imread('wiki.jpg',0)
equ = cv.equalizeHist(img)
res = np.hstack((img,equ)) #stacking images side-by-side
cv.imwrite('res.png',res)
```

因此，现在您可以在不同的光照条件下拍摄不同的图像，对其进行均衡并检查结果。

当图像的直方图限制在特定区域时，直方图均衡化效果很好。在直方图覆盖较大区域(即同时存在亮像素和暗像素)的强度变化较大的地方，效果不好。请检查其他资源中的SOF链接。

CLAHE(对比度受限的自适应直方图均衡)

我们刚刚看到的第一个直方图均衡化考虑了图像的整体对比度。在许多情况下，这不是一个好主意。例如，下图显示了输入图像及其在全局直方图均衡后的结果。



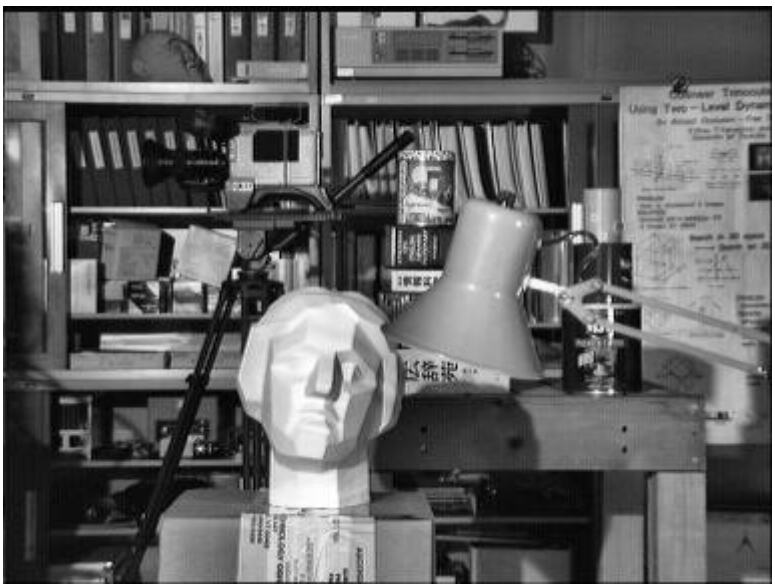
直方图均衡后，背景对比度确实得到了改善。但是在两个图像中比较雕像的脸。由于亮度过高，我们在那里丢失了大多数信息。这是因为它的直方图不像我们在前面的案例中所看到的那样局限于特定区域(尝试绘制输入图像的直方图，您将获得更多的直觉)。

因此，为了解决这个问题，使用了 **自适应直方图均衡** 。在这种情况下，图像被分成称为“ tiles”的小块(在OpenCV中，tileSize默认为8x8)。然后，像往常一样对这些块中的每一个进行直方图均衡。因此，在较小的区域中，直方图将局限于一个较小的区域(除非有噪声)。如果有噪音，它将被放大。为了避免这种情况，应用了 **对比度限制** 。如果任何直方图bin超过指定的对比度限制(在OpenCV中默认为40)，则在应用直方图均衡之前，将这些像素裁剪并均匀地分布到其他bin。均衡后，要消除图块边界中的伪影，请应用双线性插值。

下面的代码片段显示了如何在OpenCV中应用CLAHE：

```
import numpy as np
import cv2 as cv
img = cv.imread('tsukuba_l.png',0)
# create a CLAHE object (Arguments are optional).
clahe = cv.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
cl1 = clahe.apply(img)
cv.imwrite('clahe_2.jpg',cl1)
```

查看下面的结果，并将其与上面的结果进行比较，尤其是雕像区域：



其他资源

1. 直方图均衡化的维基百科页面
2. Numpy中的模版数组

还要检查有关对比度调整的以下SOF问题：

1. 如何在C中的OpenCV中调整对比度?
2. 如何使用opencv均衡图像的对比度和亮度?

直方图 - 3：2D 直方图

学习查找和绘制 2D 直方图

目标

在本章中，我们将学习查找和绘制2D直方图。这将在以后的章节中有所帮助。

介绍

在第一篇文章中，我们计算并绘制了一维直方图。之所以称为一维，是因为我们仅考虑一个特征，即像素的灰度强度值。但是在二维直方图中，您要考虑两个特征。通常，它用于查找颜色直方图，其中两个特征是每个像素的色相和饱和度值。

已经有一个python样本(*samples/python/color_histogram.py*)用于查找颜色直方图。我们将尝试了解如何创建这种颜色直方图，这对于理解诸如直方图反投影之类的更多主题将很有用。

OpenCV中的2D直方图

它非常简单，并且使用相同的函数 **cv.calcHist()** 进行计算。对于颜色直方图，我们需要将图像从BGR转换为HSV。(请记住，对于一维直方图，我们从BGR转换为灰度)。对于2D直方图，其参数将进行如下修改：

- channels = [0,1]， 因为我们需要同时处理H和S平面。
- bins = [180,256] 对于H平面为180，对于S平面为256。
- range= [0,180,0,256] 色相值介于0和180之间，饱和度介于0和256之间。 现在检查以下代码：

```
import numpy as np
import cv2 as cv
img = cv.imread('home.jpg')
hsv = cv.cvtColor(img,cv.COLOR_BGR2HSV)
hist = cv.calcHist([hsv], [0, 1], None, [180, 256], [0, 180, 0, 256])
```

而已。

Numpy中的2D直方图

Numpy还为此提供了一个特定功能：**np.histogram2d()** 。(请记住，对于一维直方图，我们使用了 **np.histogram()**)。

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('home.jpg')
hsv = cv.cvtColor(img,cv.COLOR_BGR2HSV)
hist, xbins, ybins = np.histogram2d(h.ravel(),s.ravel(),[180,256],[[0,180],[0,256]])
```

第一个参数是H平面，第二个参数是S平面，第三个参数是每个仓的数量，第四个是它们的范围。

现在我们可以检查如何绘制此颜色直方图。

绘制2D直方图

方法-1：使用 cv.imshow()

我们得到的结果是尺寸为180x256的二维数组。因此，可以使用 **cv.imshow()** 函数像平常一样显示它们。它将是一幅灰度图像，除非您知道不同颜色的色相值，否则不会对其中的颜色有太多了解。

方法-2：使用Matplotlib

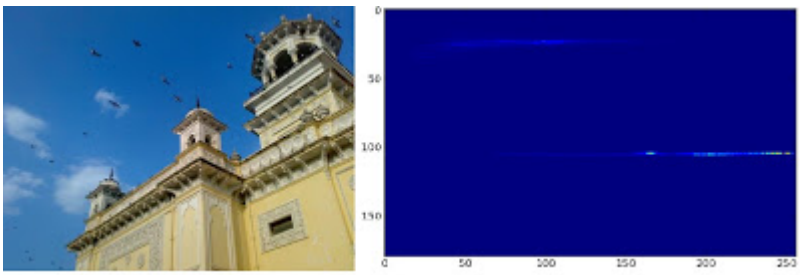
我们可以使用 **matplotlib.pyplot.imshow()** 函数绘制具有不同颜色图的2D直方图。它使我们对不同的像素密度有了更好的了解。但是，这也并不能使我们一眼就能知道是什么颜色，除非您知道不同颜色的色相值。我还是更喜欢这种方法。它简单而更好。

> **注意** 使用此功能时，请记住，插值标记应最接近以获得更好的结果。

考虑代码：

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('home.jpg')
hsv = cv.cvtColor(img,cv.COLOR_BGR2HSV)
hist = cv.calcHist( [hsv], [0, 1], None, [180, 256], [0, 180, 0, 256] )
plt.imshow(hist,interpolation = 'nearest')
plt.show()
```

下面是输入图像及其颜色直方图。X轴显示S值，Y轴显示色相。



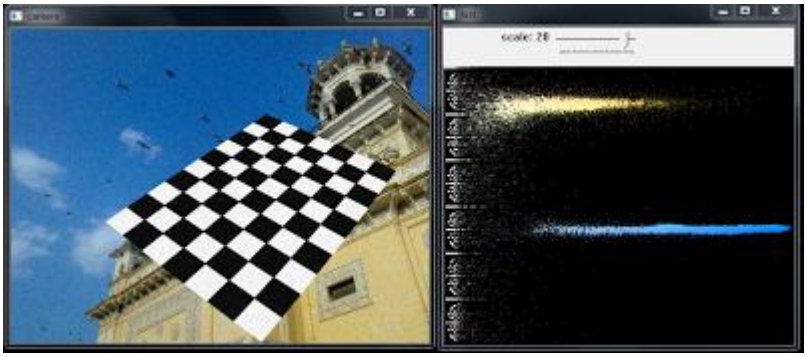
在直方图中，您可以在H = 100和S = 200附近看到一些较高的值。它对应于天空的蓝色。同样，在H = 25和S = 100附近可以看到另一个峰值。它对应于宫殿的黄色。您可以使用GIMP等任何图像编辑工具进行验证。

方法3：OpenCV示例样式！

OpenCV-Python2示例中有一个颜色直方图的示例代码(*samples/python/color_histogram.py*)。如果运行代码，则可以看到直方图也显示了相应的颜色。或者简单地，它输出颜色编码的直方图。其结果非常好(尽管您需要添加额外的线束)。

在该代码中，作者在HSV中创建了一个颜色图。然后将其转换为BGR。将所得的直方图图像与此颜色图相乘。他还使用一些预处理步骤来删除小的孤立像素，从而获得良好的直方图。

我将它留给读者来运行代码，对其进行分析并拥有自己的解决方法。下面是与上面相同的图像的代码输出：



您可以在直方图中清楚地看到存在什么颜色，那里是蓝色，那里是黄色，并且由于棋盘而有些白色。不错！

直方图 - 4：直方图反投影

了解直方图反投影以分割彩色对象

目标

在本章中，我们将学习直方图反投影。

理论

它是由 **Michael J. Swain** 和 **Dana H. Ballard** 在他们的论文“ 通过颜色直方图索引”中提出的。

简单来说到底是什么？它用于图像分割或在图像中查找感兴趣的对象。简而言之，它创建的图像大小与输入图像相同(但只有一个通道)，其中每个像素对应于该像素属于我们物体的概率。在更简单的环境中，与其余部分相比，输出图像将使我们感兴趣的对象具有更多的白色。好吧，这是一个直观的解释。(我无法使其更简单)。直方图反投影与camshift算法等配合使用。

我们该怎么做呢？ 我们创建一个图像的直方图，其中包含我们感兴趣的对象(在我们的示例中是地面，离开播放器等)。对象应尽可能填充图像以获得更好的效果。而且颜色直方图比灰度直方图更可取，因为对象的颜色比其灰度强度是定义对象的更好方法。然后，我们将该直方图“反向投影”到需要找到对象的测试图像上，换句话说，我们计算出属于地面的每个像素的概率并将其显示出来。在适当的阈值下产生的输出仅使我们有基础。

Numpy中的算法

1. 首先，我们需要计算我们要查找的对象(使其为“ M”)和要搜索的图像(使其为“ I”)的颜色直方图。

```
import numpy as np
import cv2 as cvfrom matplotlib import pyplot as plt
#roi is the object or region of object we need to find
roi = cv.imread('rose_red.png')
hsv = cv.cvtColor(roi,cv.COLOR_BGR2HSV)
#target is the image we search in
target = cv.imread('rose.png')
hsvt = cv.cvtColor(target,cv.COLOR_BGR2HSV)
# Find the histograms using calcHist. Can be done with np.histogram2d also
M = cv.calcHist([hsv],[0, 1], None, [180, 256], [0, 180, 0, 256] )
I = cv.calcHist([hsvt],[0, 1], None, [180, 256], [0, 180, 0, 256] )
```

2. 求出比率 $R=\frac{M}{I}$。然后反向投影R，即使用R作为调色板，并以每个像素作为其对应的目标概率创建一个新图像。即 $B(x，y)= R [h(x，y)，s(x，y)]$ 其中h是色调，s是像素在(x，y)的饱和度。之后，应用条件 $B(x,y)= min[B(x,y),1]$ 。

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

3. 现在对圆盘应用卷积，$B=D∗B$ ，其中D是光盘内核。

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

4. 现在最大强度的位置给了我们物体的位置。如果我们期望图像中有一个区域，则对合适的值进行阈值处理会得到不错的结果。

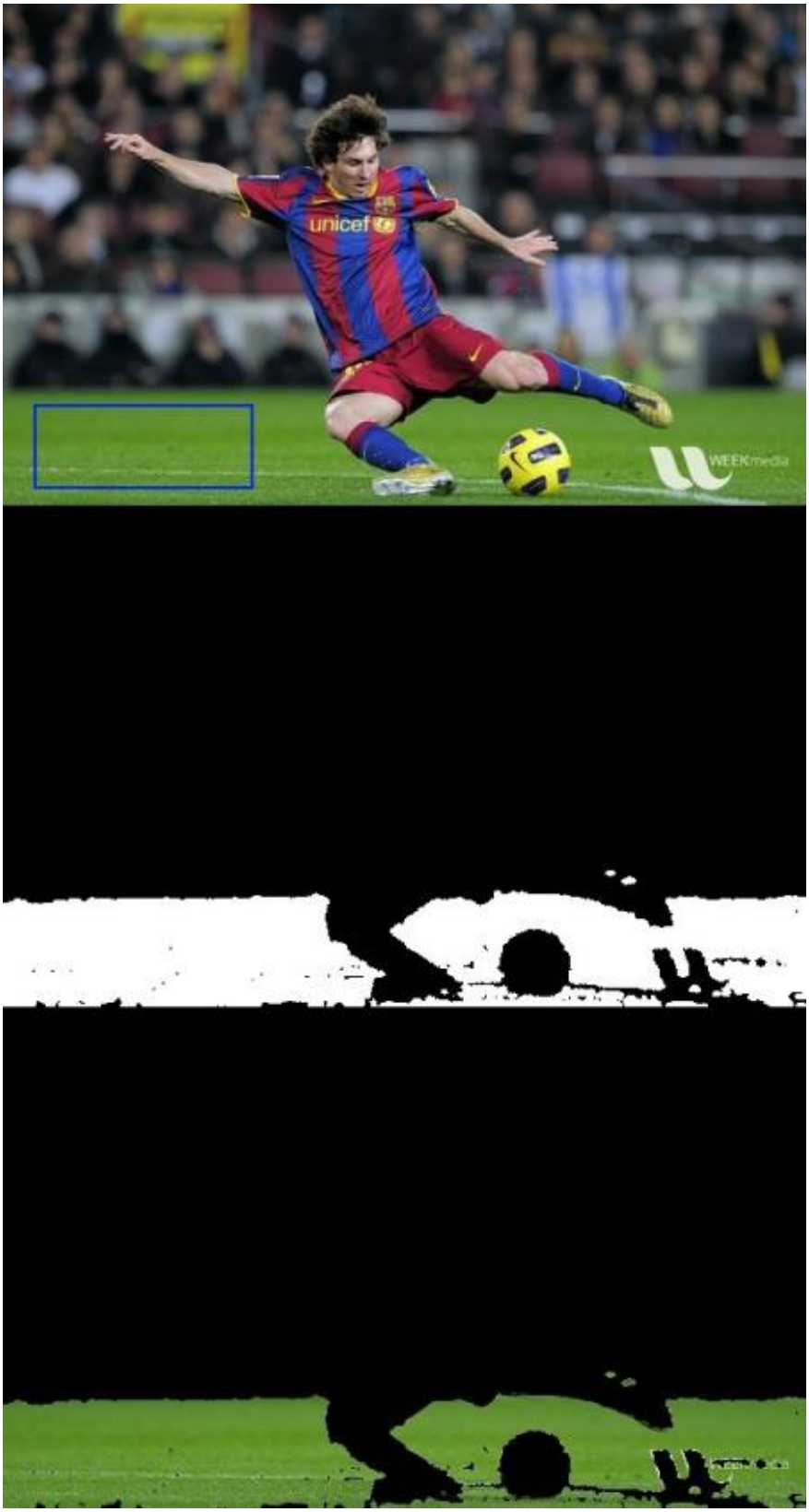
```
ret,thresh = cv.threshold(B,50,255,
```

OpenCV中的反投影

OpenCV提供了一个内置函数 **cv.calcBackProject()** 。它的参数与 **cv.calcHist()** 函数几乎相同。它的参数之一是直方图，它是对象的直方图，我们必须找到它。另外，在传递给backproject函数之前，应对对象直方图进行标准化。它返回概率图像。然后，我们将图像与磁盘内核卷积并应用阈值。下面是我的代码和输出：

```
import numpy as np
import cv2 as cv
roi = cv.imread('rose_red.png')
hsv = cv.cvtColor(roi,cv.COLOR_BGR2HSV)
target = cv.imread('rose.png')
hsvt = cv.cvtColor(target,cv.COLOR_BGR2HSV)
# calculating object histogram
roihist = cv.calcHist([hsv],[0, 1], None, [180, 256], [0, 180, 0, 256] )
# normalize histogram and apply backprojection
cv.normalize(roihist,roihist,0,255,cv.NORM_MINMAX)
dst = cv.calcBackProject([hsvt],[0,1],roihist,[0,180,0,256],1)
# Now convolute with circular disc
disc = cv.getStructuringElement(cv.MORPH_ELLIPSE,(5,5))
cv.filter2D(dst,-1,disc,dst)
# threshold and binary AND
ret,thresh = cv.threshold(dst,50,255,0)
thresh = cv.merge((thresh,thresh,thresh))
res = cv.bitwise_and(target,thresh)
res = np.vstack((target,thresh,res))
cv.imwrite('res.jpg',res)
```

以下是我处理过的一个示例。我将蓝色矩形内的区域用作示例对象，我想提取整个地面。



其他资源

1. “通过颜色直方图建立索引”，Swain，Michael J.，第三届国际计算机视觉会议，1990年。

OpenCV 中的图像转换

此章节文档只给出了傅里叶变换, 直接将其列出本问内

傅里叶变换： 学习找到图像的傅里叶变换

目标

在本节中，我们将学习

- 使用OpenCV查找图像的傅立叶变换
- 利用Numpy中可用的FFT功能
- 傅立叶变换的一些应用
- 我们将看到以下函数：**cv.dft()** ， **cv.idft()** 等

理论

傅立叶变换用于分析各种滤波器的频率特性。对于图像，使用 **2D离散傅里叶变换（DFT）** 查找频域。快速算法称为 **快速傅立叶变换（FFT）** 用于计算DFT。关于这些的详细信息可以在任何图像处理或信号处理教科书中找到。请参阅其他资源_部分。

对于正弦信号，$x(t)=Asin(2\pi ft)$ ，我们可以说 f 是信号的频率，如果采用其频域，我们可以在 f 处看到一个尖峰。如果信号进行采样，以形成离散信号，我们得到了相同的频域，但在范围周期性 $[-π，π]$ 或 $[0,2\pi]$（或 $[0,N]$ 用于N点DFT）。您可以将图像视为在两个方向上采样的信号。因此，在X和Y方向都进行傅立叶变换，可以得到图像的频率表示。

更直观地说，对于正弦信号，如果振幅在短时间内变化如此之快，则可以说它是高频信号。如果变化缓慢，则为低频信号。您可以将相同的想法扩展到图像。图像中的振幅在哪里急剧变化？在边缘点或噪音。因此，可以说边缘和噪声是图像中的高频内容。如果幅度没有太大变化，则它是低频分量。（一些链接已添加到“其他资源”，其中通过示例直观地说明了频率变换）。

现在，我们将看到如何找到傅立叶变换。

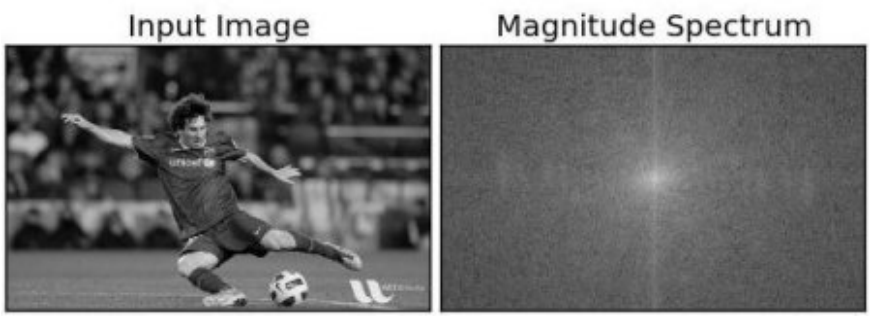
Numpy中的傅立叶变换

首先，我们将看到如何使用Numpy查找傅立叶变换。Numpy具有FFT软件包来执行此操作。**np.fft.fft2()** 为我们提供了频率转换，它将是一个复杂的数组。它的第一个参数是输入图像，即灰度图像。第二个参数是可选的，它决定输出数组的大小。如果它大于输入图像的大小，则在计算FFT之前用零填充输入图像。如果小于输入图像，将裁切输入图像。如果未传递任何参数，则输出数组的大小将与输入的大小相同。

现在，一旦获得结果，零频率分量（DC分量）将位于左上角。如果要使其居中，则需要将结果偏移 $\frac{N}{2}$ 在两个方向上。只需通过函数 **np.fft.fftshift()** 即可完成。（它更容易分析）。找到频率变换后，就可以找到幅度谱。

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
img = cv.imread('messi5.jpg',0)
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
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：

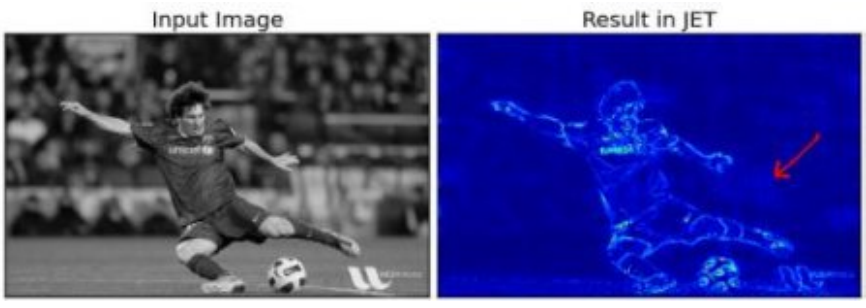


看，您可以在中心看到更多白色区域，这表明低频内容更多。

因此，您已经进行了频率变换，您可以在频域中执行一些操作，例如高通滤波和重建图像，若进行逆DFT。为此，您需用尺寸为60x60的矩形窗口遮罩来消除低频。然后，使用 **np.fft.ifftshift()** 应用反向移位，以使DC分量再次出现在左上角。然后使用 **np.ifft2()** 函数找到逆FFT 。同样，结果将是一个复数。您可以采用其绝对值来进行

```
rows, cols = img.shape
crow,ccol = rows//2 , cols//2
fshift[crow-30:crow+31, ccol-30:ccol+31] = 0
f_ishift = np.fft.ifftshift(fshift)
img_back = np.fft.ifft2(f_ishift)
img_back = np.real(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：



结果表明高通滤波是边缘检测操作。这就是我们在“图像渐变”一章中看到的。这也表明大多数图像数据都存在于频谱的低频区域。无论如何，我们已经看到了如何在Numpy中找到DFT，IDFT等。现在，让我们看看如何在OpenCV中进行操作。

如果您仔细观察结果，尤其是最后一张JET颜色的图像，您会看到一些伪像（我用红色箭头标记的一个实例）。它在那里显示出一些波纹状结构，称为 **振铃效应** 。这是由我们用于遮罩的矩形窗口引起的。此蒙版转换为正弦形状，从而导致此问题。因此，矩形窗口不用于过滤。更好的选择是高斯窗口。

OpenCV中的傅立叶变换

OpenCV 为此提供了功能 **cv.dft()** 和 **cv.idft()** 。它返回与以前相同的结果，但是有两个通道。第一个通道将具有结果的实部，第二个通道将具有结果的虚部。输入的图像应首先转换为**np.float32** 。我们将看到如何做。

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('messi5.jpg',0)
dft = cv.dft(np.float32(img),flags = cv.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
magnitude_spectrum = 20*np.log(cv.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** 您还可以使用 **cv.cartToPolar()** 一次返回大小和相位

因此，现在我们必须进行逆DFT。在上一部分中，我们创建了一个HPF，这次我们将看到如何去除图像中的高频内容，即我们将LPF应用于图像。实际上会使图像模糊。为此，我们首先创建一个在低频时具有高值（1）的蒙版，即，我们传递LF含量，并在HF区域传递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 = cv.idft(f_ishift)
img_back = cv.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()
```

查看结果：

> **注意** 像往常一样，OpenCV函数 **cv.dft()** 和 **cv.idft()** 比Numpy对应函数要快。但是Numpy功能更加人性化。有关性能问题的更多详细信息，请参阅以下部分。

DFT的性能优化

对于某些阵列大小，DFT计算的性能更好。当阵列大小为2的幂时，它是最快的。大小为2、3和5的乘积的数组也得到了有效处理。因此，如果您担心代码的性能，可以在找到DFT之前将数组的大小修改为任何最佳大小（通过填充零）。对于OpenCV，您必须手动填充零。但是对于Numpy，您可以指定FFT计算的新大小，它将自动为您填充零。

那么我们如何找到这个最佳尺寸呢？OpenCV 为此提供了一个函数 **cv.getOptimalDFTSize()** 。它适用于 **cv.dft()** 和 **np.fft.fft2()** 。让我们使用IPython magic命令timeit检查它们的性能。

```
In [16]: img = cv.imread('messi5.jpg',0)
In [17]: rows,cols = img.shape
In [18]: print("{} {}".format(rows,cols))
342 548
In [19]: nrows = cv.getOptimalDFTSize(rows)
In [20]: ncols = cv.getOptimalDFTSize(cols)
In [21]: print("{} {}".format(nrows,ncols))
360 576
```

参见，将大小（342,548）修改为（360，576）。现在让我们用零填充（对于OpenCV），并找到其DFT计算性能。您可以通过创建一个新的大零数组并将数据复制到其中来完成此操作，或者使用 **cv.copyMakeBorder()** 。

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

要么：

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

现在，我们计算Numpy函数的DFT性能比较：

```
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
```

它显示了4倍的加速。现在，我们将尝试使用OpenCV函数。

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

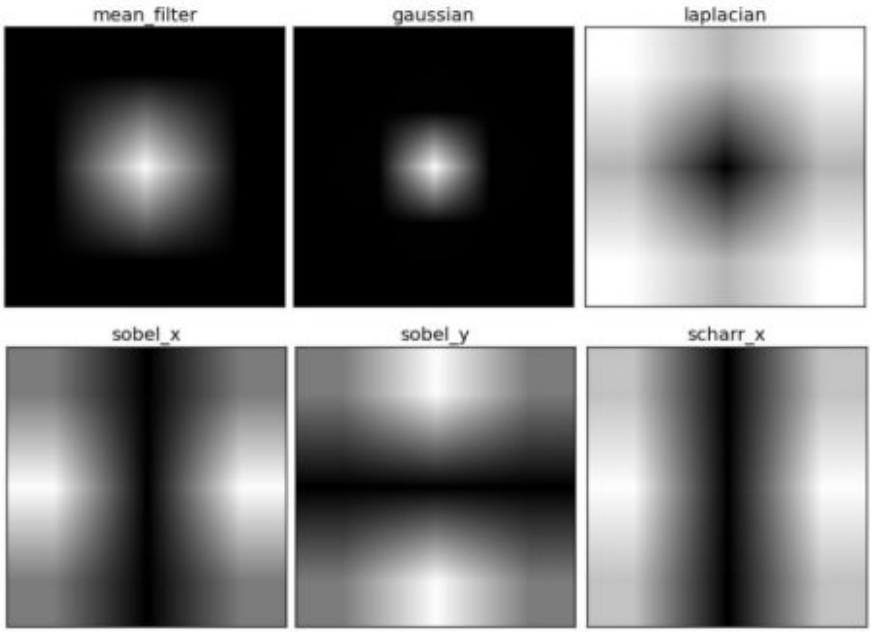
它还显示了4倍的加速。您还可以看到OpenCV函数比Numpy函数快3倍左右。也可以对逆FFT进行测试，这留给您练习。

为什么拉普拉斯算子是高通滤波器?

在论坛上提出了类似的问题。问题是，为什么拉普拉斯算子是高通滤波器？为什么Sobel是HPF？等等。第一个得到的答案是傅里叶变换。只需对Laplacian进行傅立叶变换，以获得更大的FFT大小。分析一下：

```
import cv2 as cv
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 = cv.getGaussianKernel(5,10)
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()
```

查看结果：



从图像中，您可以看到每个内核阻止的频率区域以及它经过的区域。从这些信息中，我们可以说出为什么每个内核都是HPF或LPF

其他资源

史蒂文·雷哈尔（Steven Lehar）对傅立叶理论的直观解释 HIPR的傅立叶变换 对于图像，频域表示什么?

模板匹配

目标

在本章中，您将学习

- 使用模板匹配查找图像中的对象
- 您将了解这些函数： **cv.matchTemplate（）** ， **cv.minMaxLoc（）**

理论

模板匹配是一种在较大图像中搜索和查找模板图像位置的方法。为此，OpenCV 带有一个函数 **cv.matchTemplate（）** 。它只是在输入图像上滑动模板图像（如在 2D 卷积中），并比较模板图像下的模板和输入图像的补丁。在 OpenCV 中实现了几种比较方法。（您可以查看文档以获取更多详细信息）。它返回一个灰度图像，其中每个像素表示该像素的邻域与模板匹配的程度。

如果输入图像的大小（WxH）且模板图像的大小（wxh），则输出图像的大小为（W-w + 1，H-h + 1）。得到结果后，可以使用 **cv.minMaxLoc（）** 函数查找最大/最小值的位置。将其作为矩形的左上角，取（w，h）作为矩形的宽度和高度。那个矩形是你的模板区域。

Note

如果你使用cv.TM_SQDIFF函数作为比较的方法, 最小值作为匹配值。

OpenCV 中的模板匹配

在这里，作为一个例子，我们将在梅西的照片中搜索他的面部，因此我创建了一个如下的模板：

我们将尝试所有的比较方法，看看它们的结果如何：

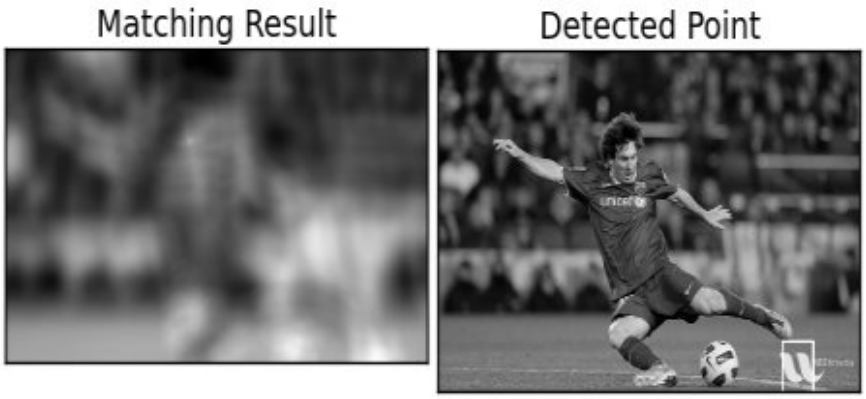
```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
img = cv.imread('messi5.jpg',0)
img2 = img.copy()
template = cv.imread('template.jpg',0)
w, h = template.shape[::-1]
# All the 6 methods for comparison in a list
methods = ['cv.TM_CCOEFF', 'cv.TM_CCOEFF_NORMED', 'cv.TM_CCORR',
            'cv.TM_CCORR_NORMED', 'cv.TM_SQDIFF', 'cv.TM_SQDIFF_NORMED']
for meth in methods:
    img = img2.copy()
    method = eval(meth)
    # Apply template Matching
    res = cv.matchTemplate(img,template,method)
    min_val, max_val, min_loc, max_loc = cv.minMaxLoc(res)
    # If the method is TM_SQDIFF or TM_SQDIFF_NORMED, take minimum
    if method in [cv.TM_SQDIFF, cv.TM_SQDIFF_NORMED]:
        top_left = min_loc
    else:
        top_left = max_loc
    bottom_right = (top_left[0] + w, top_left[1] + h)
    cv.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()
```

请参阅以下结果：

- cv.TM_CCOEFF

- cv.TM_CCOEFF_NORMED

- cv.TM_CCORR



- cv.TM_CCORR_NORMED

- cv.TM_SQDIFF

- cv.TM_SQDIFF_NORMED

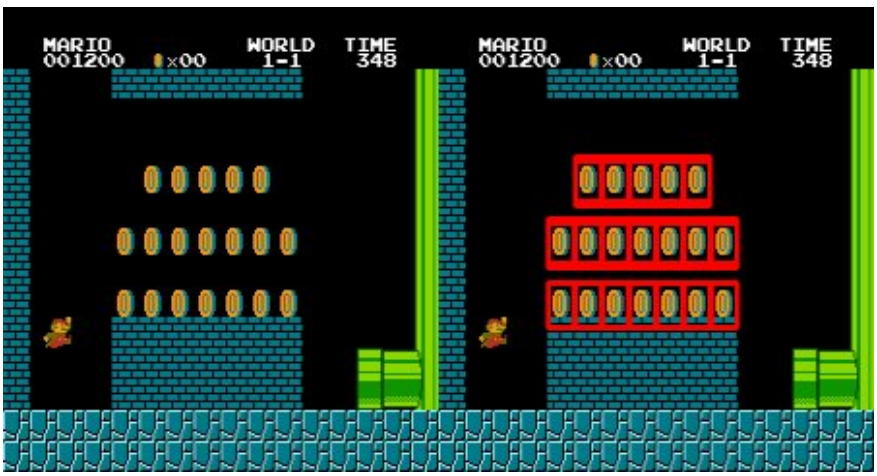
你可以看到使用 **cv.TM_CCORR**的结果并不像我们预期的那样好。

模板与多个对象匹配

在上一节中，我们搜索了梅西的脸部图像，该图像仅在图中出现一次。假设您正在搜索的对象在图中出现了多次， **cv.minMaxLoc（）** 将不会为你提供所有的匹配点。在这种情况下，我们将使用阈值。所以在这个例子中，我们将使用着名游戏 **Mario** 的截图，并在其中找到硬币。

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
img_rgb = cv.imread('mario.png')
img_gray = cv.cvtColor(img_rgb, cv.COLOR_BGR2GRAY)
template = cv.imread('mario_coin.png',0)
w, h = template.shape[::-1]
res = cv.matchTemplate(img_gray,template,cv.TM_CCOEFF_NORMED)
threshold = 0.8
loc = np.where( res >= threshold)
for pt in zip(*loc[::-1]):
    cv.rectangle(img_rgb, pt, (pt[0] + w, pt[1] + h), (0,0,255), 2)
cv.imwrite('res.png',img_rgb)
```

结果：



其他资源

练习

霍夫线变换

目标

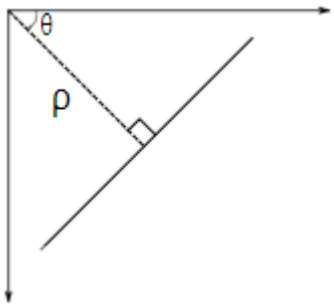
在这一章当中，

- 我们将理解霍夫变换的概念。
- 我们将看到如何使用它来检测图像中的线条。
- 我们将了解以下函数： **cv.HoughLines（）** ， **cv.HoughLinesP（）**

理论

如果您能够以数学形式表示该形状，则霍夫变换是一种用于检测任何形状的流行技术。它可以检测到形状，即使它被破坏或扭曲一点点。我们将看到它如何适用于生产线。

一条线可以表示为$y = mx+c$或以极坐标形式 $\rho = x \cos \theta + y \sin \theta$表示为其中$\rho$是垂线从原点到直线的距离。$\theta$是这条垂直线形成的角度，水平轴是逆时针测量的（该方向因你代表坐标系而变化。这种表示用于 OpenCV ）。检查下图：

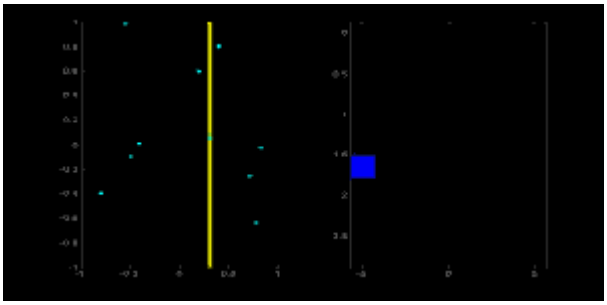


所以如果这条线经过原点以下，它会有一个小于 180 度的正角度。如果它在原点上方，我们不是取一个大于 180 的角，而是取一个小于 180 的负角度。任何垂直线都是 0 度，水平线是 90 度。

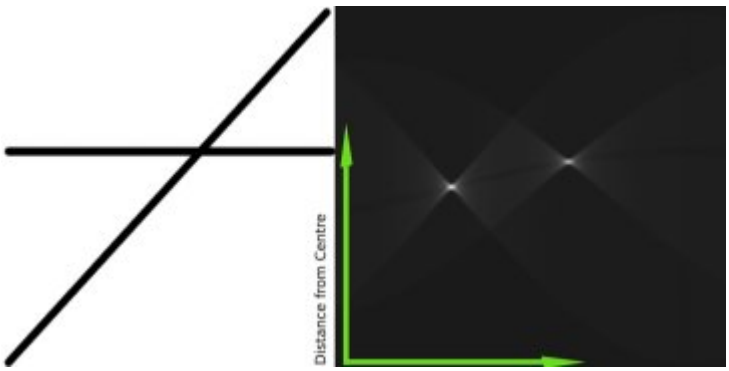
现在我们来看看霍夫变换是怎么作用于直线的。任何线都可以被表示成这两项。所以首先它创建了一个二维数组，或者是累加器(来保存这两个参数的值)然后他设置作为初始值。 令(二维数组的)行表示，令列表示 。数组的大小取决于你需要的精准度。假设说你想要角度的精度是精确到 1 度，那你就需要 180 列。而对于来说，最大可能的距离是图像的对角线长度。因此，要精确到一个像素的程度，行数应该是图像的对角长度。

想象有一张 100x100 的图像，它上面有一条水平的线条在图像正中间。取这条线的第一个点，你知道它的坐标 (x,y)值。现在，按照这条直线的等式，把值$\theta = 0, 1, 2, ..., 180$带入，并且查看你获取到的值。对每一对，我们都把它们在累加器中对应的值计数增加 1。而此时，在这个计数器中，单元 (50,90) 和其他单元一样计数等于 1。

现在去这条线上的第二个点，重复上面的步骤。增加单元中你拿到的对应的值(rho, theta)。这一次，单元(50,90)的计数增到到了 2。你实际上做的是投票投出值。你不断为这条直线上所有的点继续这个过程。在每一个点，单元(50,90)都会得票，而其他的单元有可能会得票，也有可能不会。按这个方案，最终单元 (50,90) 会获得最高的投票(译者注：在 100X100 的图像正中水平的直线到原点距离为 50，垂角 90 度)。所以最终搜索我们的累加器来找最高得票的单元，我们就会取到 (50,90)，这就说明图像中有一条线距离原点垂距 50，它过原点的垂线和水平线夹角为 90 度。这个过程很好的显示在了以下的动画中。（图片提供： Amos Storkey ）



这就是霍夫变化针对直线工作的原理。非常简单，也许你可以用 Numpy 自己来实现它。以下是一张显示了累加器的图像。在某些地方的亮点说明它们是图像中可能线条的参数。（图片提供：维基百科）

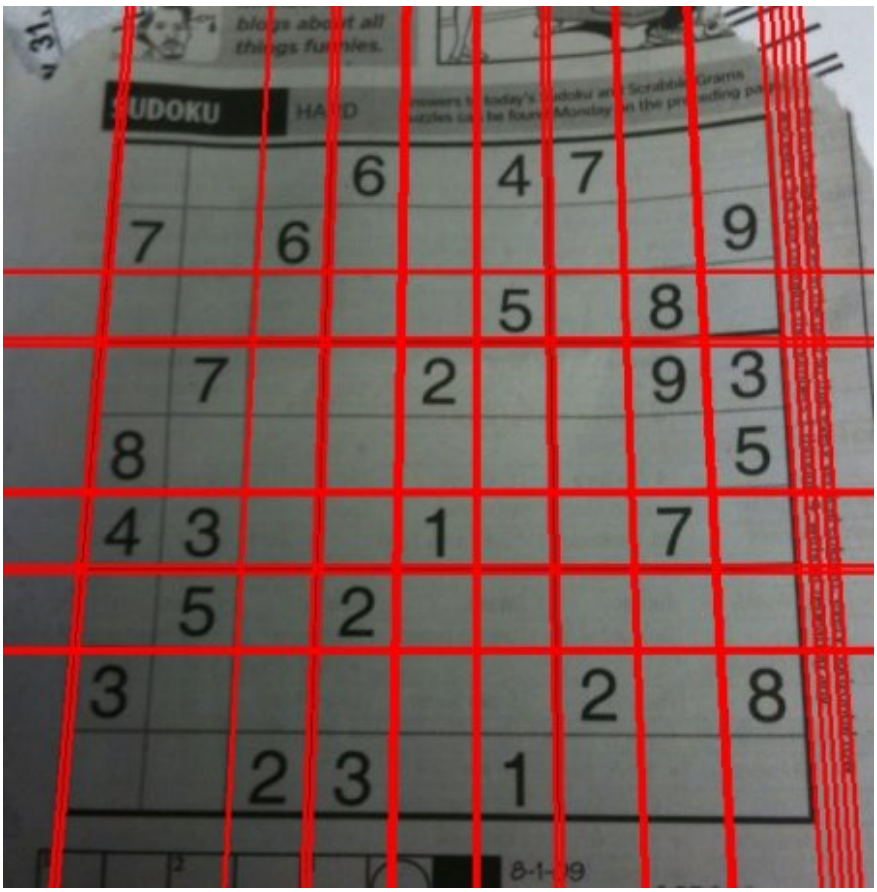


OpenCV 中的霍夫变换

上面解释的这一堆东西，在 OpenCV 里都封装起来成为**cv.HoughLines（）** 函数了。它简单的返回了一个:math:(rho, theta)`值得数组。ρ 的单位是像素，θ的单位是弧度。第一个参数，输入图像应该是个二元图像，所以在应用霍夫线性变换之前先来个阈值法或者坎尼边缘检测。第二、第三参数分别是 ρ 和 θ 的精度。第四个参数则是一个阈值，它代表了一个(ρ,θ)单元被认为是一条直线需要获得的最低票数。要记住的是，得票数其实取决于这条直线穿过了多少个点。所以它也代表了应被检测出的线条最少有多长。

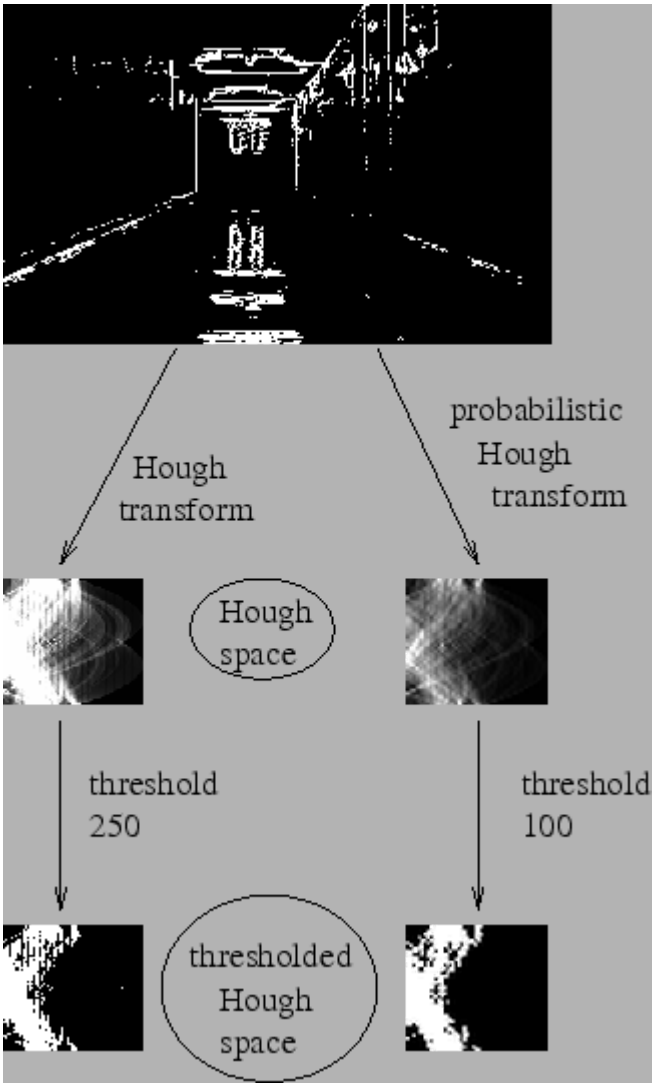
```
import cv2 as cv
import numpy as np
img = cv.imread(cv.samples.findFile('sudoku.png'))
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
edges = cv.Canny(gray,50,150,apertureSize = 3)
lines = cv.HoughLines(edges,1,np.pi/180,200)
for line in lines:
    rho,theta = line[0]
    a = np.cos(theta)
    b = np.sin(theta)
    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))
    cv.line(img,(x1,y1),(x2,y2),(0,0,255),2)
cv.imwrite('houghlines3.jpg',img)
```

检查以下结果：



概率 Hough 变换

在霍夫变换中，你可以发现即使是一条仅有两个参数的直线，也需要用到大量的计算。概率霍夫变换是我们已知的，针对霍夫变换的优化方案。它不去取所有的点来列入考虑，取而代之的是取足够完成直线检测的这些点的随机子集。只要我们把阈值下调一点。下图在霍夫空间中比较了霍夫变换与概率霍夫变换。（图片提供：Franck Bettinger 的主页）



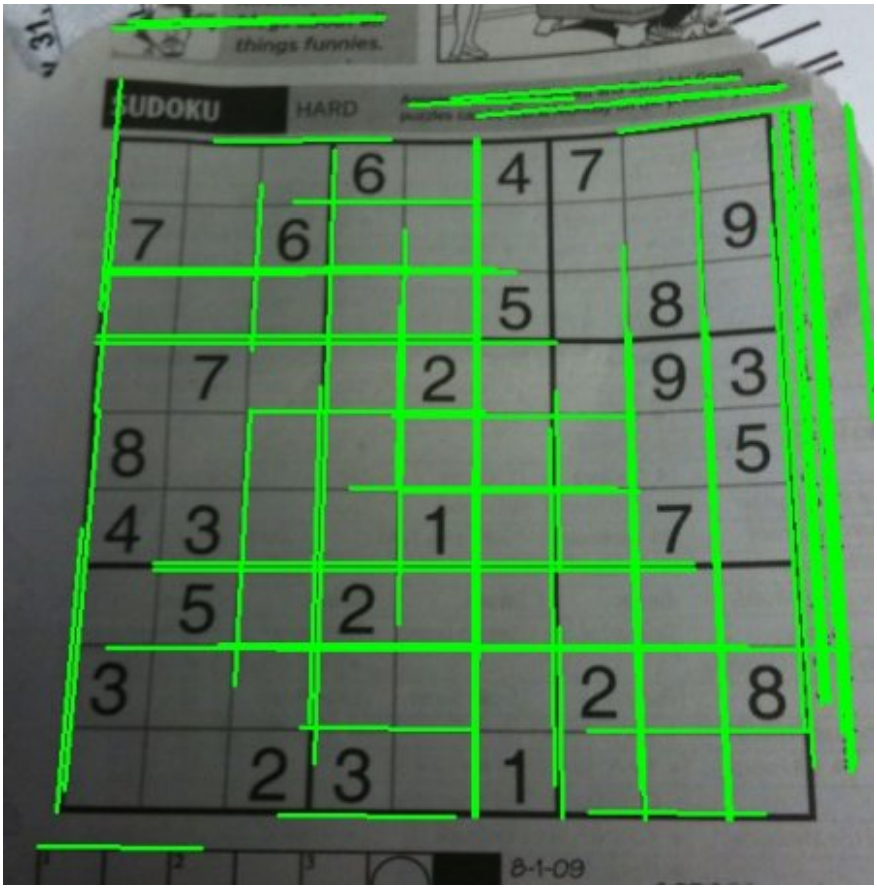
OpenCV 实现基于使用 Matas，J。和 Galambos，C。和 Kittler，J.V。 [133] 的渐进概率 Hough 变换的线的鲁棒检测。使用的函数是 **cv.HoughLinesP（）** 。它比之前介绍的函数多出来两个参数：

- **minLineLength** - 最小线长。比这个值小的线条会被丢弃。
- **maxLineGap** - 允许线段之间的最大间隙，以便将(在同一条直线上的)线段视为同一条。

最好的是，它直接返回直线的两个端点。在前面的例子中，你只得到直线的参数，你必须找到所有的点。而这个方法，一切都是那么的直接和简单。

```
import cv2 as cv
import numpy as np
img = cv.imread(cv.samples.findFile('sudoku.png'))
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
edges = cv.Canny(gray,50,150,apertureSize = 3)
lines = cv.HoughLinesP(edges,1,np.pi/180,100,minLineLength=100,maxLineGap=10)
for line in lines:
    x1,y1,x2,y2 = line[0]
    cv.line(img,(x1,y1),(x2,y2),(0,255,0),2)
cv.imwrite('houghlines5.jpg',img)
```

请参阅以下结果：



其他资源

1. 维基百科的 Hough 变换

练习

霍夫圆变换

目标

在这一章当中，

- 我们将学习使用霍夫变换来查找图像中的圆圈。
- 我们将了解这些函数： **cv.HoughCircles（）**

理论

圆圈在数学上表示为$(x-x{*center}*)^2 + *(y - y*{center})^2 = r^2$其中 $(x*{center}*,y{center})$是圆的中心，r 是圆的半径。从这个公式来看，得知我们有三个参数，这样我们就需要一个三维度的累加器来做霍夫变换了，这样效率是非常低的。所以 OpenCV 用了更巧妙的方法**Hough Gradient Method** ，它利用了边缘的梯度信息。 我们在这里使用的函数是 **cv.HoughCircles（）** 。它的参数非常的多，这些参数在文档中都有详细的解释。所以我们直接上代码吧。

```
import numpy as np
import cv2 as cv
img = cv.imread('opencv-logo-white.png',0)
img = cv.medianBlur(img,5)
cimg = cv.cvtColor(img,cv.COLOR_GRAY2BGR)
circles = cv.HoughCircles(img,cv.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
    cv.circle(cimg,(i[0],i[1]),i[2],(0,255,0),2)
    # draw the center of the circle
    cv.circle(cimg,(i[0],i[1]),2,(0,0,255),3)
cv.imshow('detected circles',cimg)
cv.waitKey(0)
cv.destroyAllWindows()
```

结果如下所示：

其他资源

练习

基于分水岭算法的图像分割

目标

- 在这一章当中，
  - 我们将学习使用分水岭算法使用基于标记的图像分割
  - 我们将看到：**cv.watershed()**

理论

任何灰度图像都可以看作是地形表面，其中高强度表示峰和丘陵，而低强度表示山谷。您开始用不同颜色的水（标签）填充每个孤立的山谷（局部最小值）。随着水的上升，取决于附近的峰值（梯度），来自不同山谷的水，明显具有不同的颜色将开始融合。为避免这种情况，您需要在水合并的位置建立障碍。你继续填补水和建筑障碍的工作，直到所有的山峰都在水下。然后，您创建的障碍为您提供分割结果。这是分水岭背后的“哲学”。您可以访问分水岭上的CMM 网页，以便在某些动画的帮助下了解它。

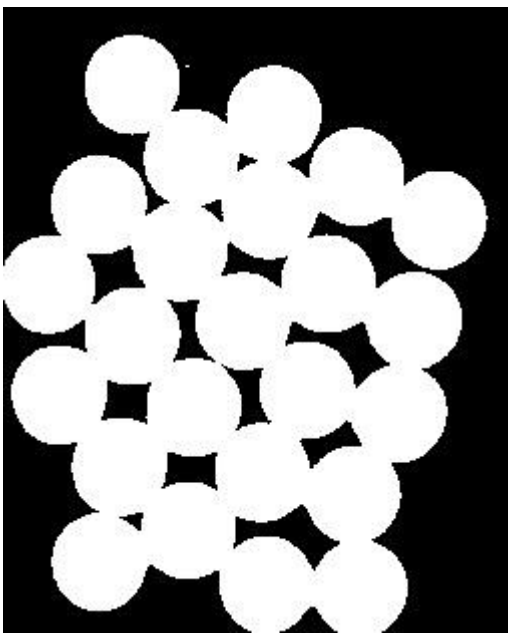
但是这种方法会因图像中的噪声或任何其他不规则性而给出过度调整结果。因此，OpenCV 实施了一个基于标记的分水岭算法，您可以在其中指定要合并的所有谷点，哪些不合并。这是一种交互式图像分割。我们所做的是为我们知道的对象提供不同的标签。用一种颜色（或强度）标记我们确定为前景或对象的区域，用另一种颜色标记我们确定为背景或非对象的区域，最后标记我们不确定任何内容的区域，用 0 标记它。这是我们的标记。然后应用分水岭算法。然后我们的标记将使用我们给出的标签进行更新，对象的边界将具有-1 的值。

代码

下面我们将看到一个如何使用距离变换和分水岭来分割相互接触的物体的示例。考虑下面的硬币图像，硬币互相接触。即使你达到阈值，它也会相互接触。

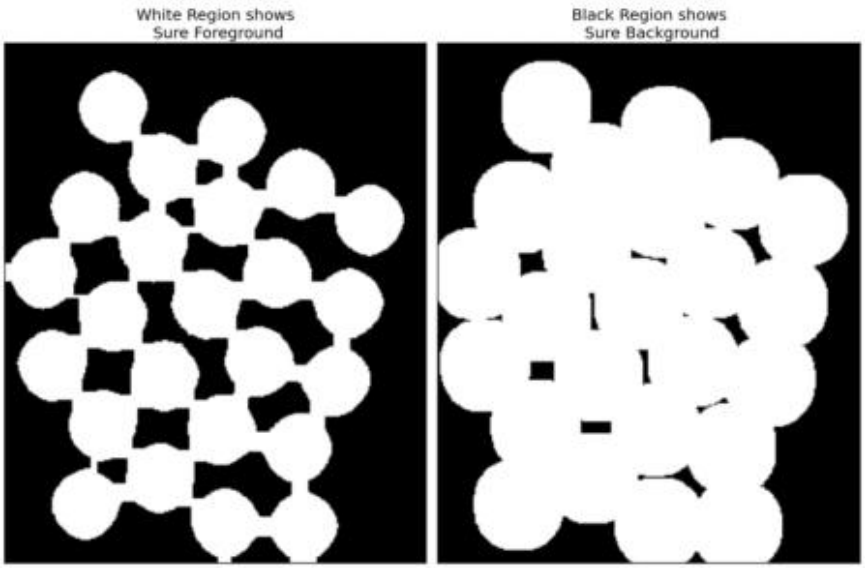
我们首先找到硬币的近似估计值。为此，我们可以使用 Otsu 的二值化。

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('coins.png')
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
ret, thresh = cv.threshold(gray,0,255,cv.THRESH_BINARY_INV+cv.THRESH_OTSU)
```



现在我们需要去除图像中的任何小白噪声。为此，我们可以使用形态开放。要移除对象中的任何小孔，我们可以使用形态学闭合。所以，现在我们确切地知道靠近物体中心的区域是前景，而远离物体的区域是背景。只有我们不确定的区域是硬币的边界区域。

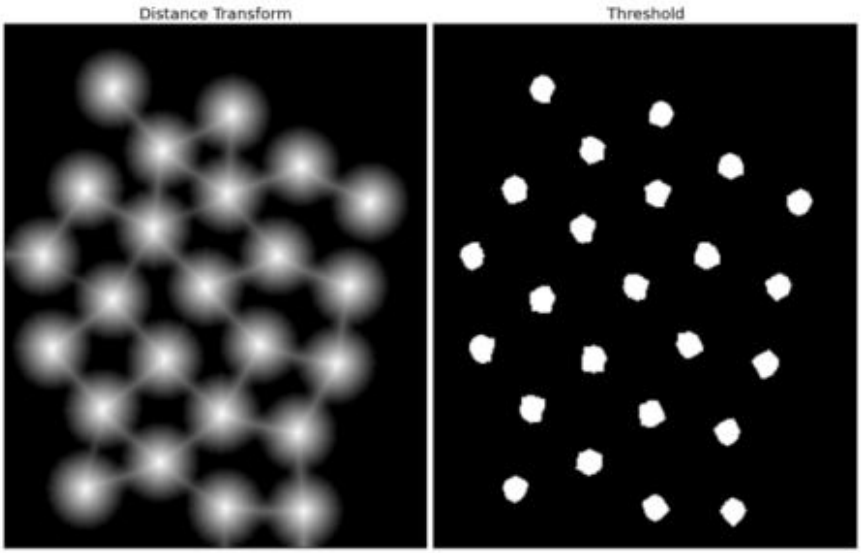
所以我们需要提取我们确定它们是硬币的区域。侵蚀消除了边界像素。所以无论如何，我们可以肯定它是硬币。如果物体没有相互接触，这将起作用。但由于它们相互接触，另一个好的选择是找到距离变换并应用适当的阈值。接下来我们需要找到我们确定它们不是硬币的区域。为此，我们扩大了结果。膨胀将物体边界增加到背景。这样，我们可以确保结果中背景中的任何区域确实是背景，因为边界区域被移除。见下图。



剩下的区域是我们不知道的区域，无论是硬币还是背景。分水岭算法应该找到它。这些区域通常围绕前景和背景相遇的硬币边界（甚至两个不同的硬币相遇）。我们称之为边界。它可以从 sure_bg 区域中减去 sure_fg 区域获得。

```
# noise removal
kernel = np.ones((3,3),np.uint8)
opening = cv.morphologyEx(thresh,cv.MORPH_OPEN,kernel, iterations = 2)
# sure background area
sure_bg = cv.dilate(opening,kernel,iterations=3)
# Finding sure foreground area
dist_transform = cv.distanceTransform(opening,cv.DIST_L2,5)
ret, sure_fg = cv.threshold(dist_transform,0.7*dist_transform.max(),255,0)
# Finding unknown region
sure_fg = np.uint8(sure_fg)
unknown = cv.subtract(sure_bg,sure_fg)
```

看到结果。在阈值图像中，我们得到了一些我们确定硬币的硬币区域，现在它们已经分离。（在某些情况下，你可能只对前景分割感兴趣，而不是分离相互接触的物体。在这种情况下，你不需要使用距离变换，只需要侵蚀就足够了。侵蚀只是提取确定前景区域的另一种方法，那就是所有。）

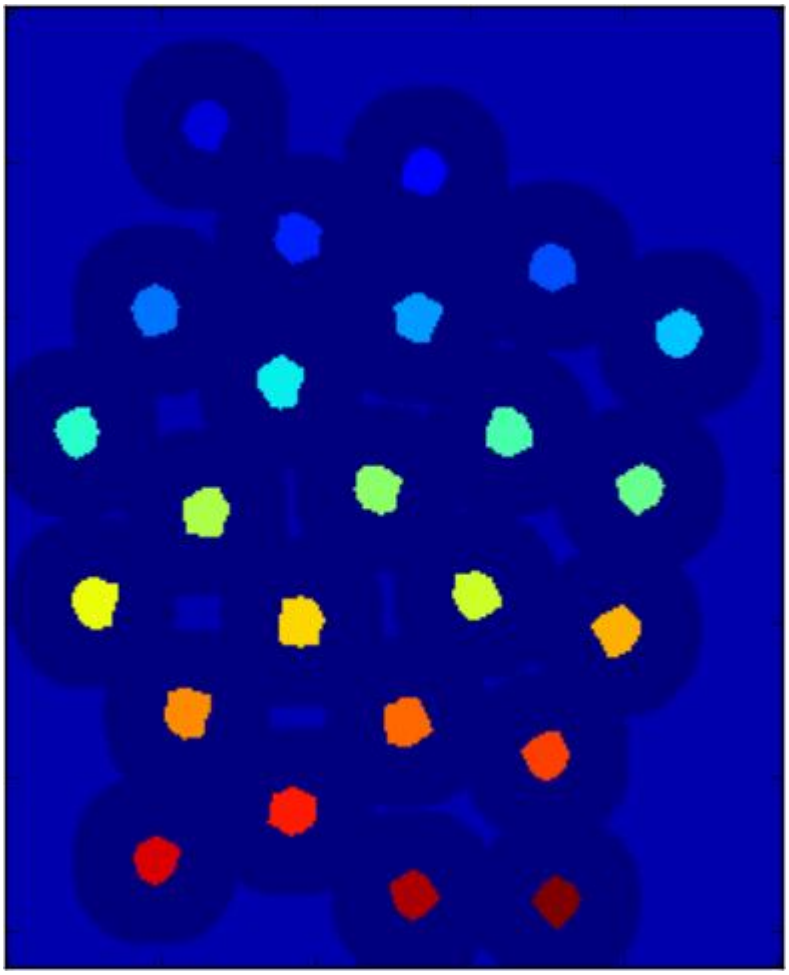


现在我们确定哪个是硬币区域，哪个是背景和所有。所以我们创建标记（它是一个与原始图像大小相同的数组，但是使用 int32 数据类型）并标记其中的区域。我们确切知道的区域（无论是前景还是背景）都标有任何正整数，但不同的整数，我们不确定的区域只是保留为零。为此，我们使用**cv.connectedComponents（）**。它用 0 标记图像的背景，然后其他对象用从 1 开始的整数标记。

但我们知道，如果背景标记为 0，分水岭会将其视为未知区域。所以我们想用不同的整数来标记它。相反，我们将标记由未知定义的未知区域，为 0。

```
# Marker labelling
ret, markers = cv.connectedComponents(sure_fg)

# Add one to all labels so that sure background is not 0, but 1
markers = markers+1
# Now, mark the region of unknown with zero
markers[unknown==255] = 0
```

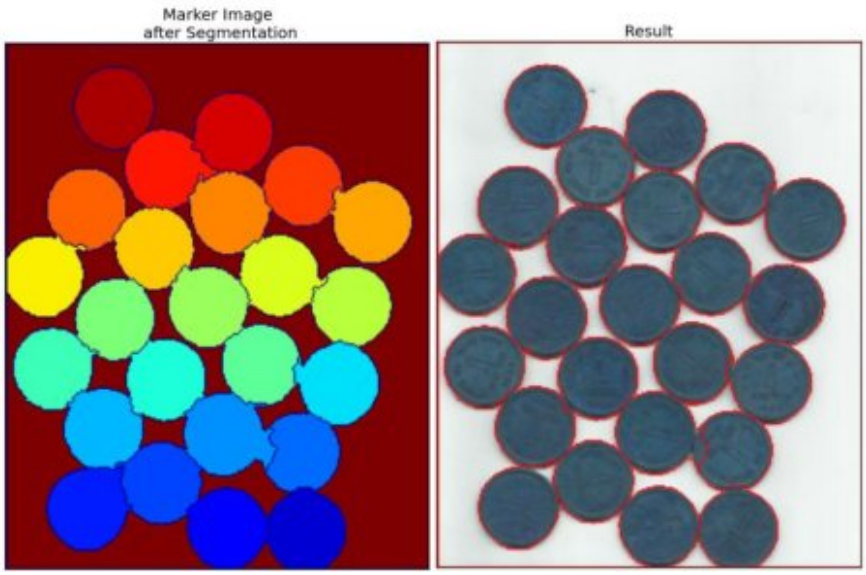
查看 JET 色彩映射中显示的结果。深蓝色区域显示未知区域。确定的硬币用不同的值着色。与未知区域相比，确定背景的剩余区域以浅蓝色显示。



现在我们的标记准备好了。现在是最后一步的时候，应用分水岭。然后将修改标记图像。边界区域将标记为-1。

```
markers = cv.watershed(img,markers)
img[markers == -1] = [255,0,0]
```

请参阅下面的结果。对于某些硬币，它们触摸的区域被正确分割，而对于某些硬币则不然。



其他资源

1. 分水岭转型的 CMM 页面

演习

1. OpenCV 样本有一个关于分水岭分割的交互式样本，watershed.py。运行它，享受它，然后学习它。

基于 GrabCut 算法的交互式前景提取

目标

在这一章当中

- 我们将看到 GrabCut 算法来提取图像中的前景
- 我们将为此创建一个交互式应用程序。

理论

GrabCut 算法由 Carsten Rother，Vladimir Kolmogorov＆amp;来自英国剑桥微软研究院的 Andrew Blake。在他们的论文中，“GrabCut”：使用迭代图切割中提出来的。该算法需要最小的人工交互来做前景提取，这个算法被称为 GrabCut。

从用户的角度来看，该算法是如何工作的呢？最初用户在前景区域周围绘制一个矩形(该矩形需要完全框住所有的前景区域) 。然后算法对其进行迭代分割，得到最佳结果。但在某些情况下，分割的不是那么理想，比如说，它可能把一些前景区域标成了背景，或者反过来。如果发生了这样的情况，用户需要进行仔细的修正。只要在有错误结果的地方“划一下”就行了。“划一下”基本上的意思就是说，*"嘿，这个区域应该是前景，你标记它为背景，在下一次迭代中更正它。"* 或者对背景来说，反过来。然后再下一次迭代中，你就会得到更好的结果

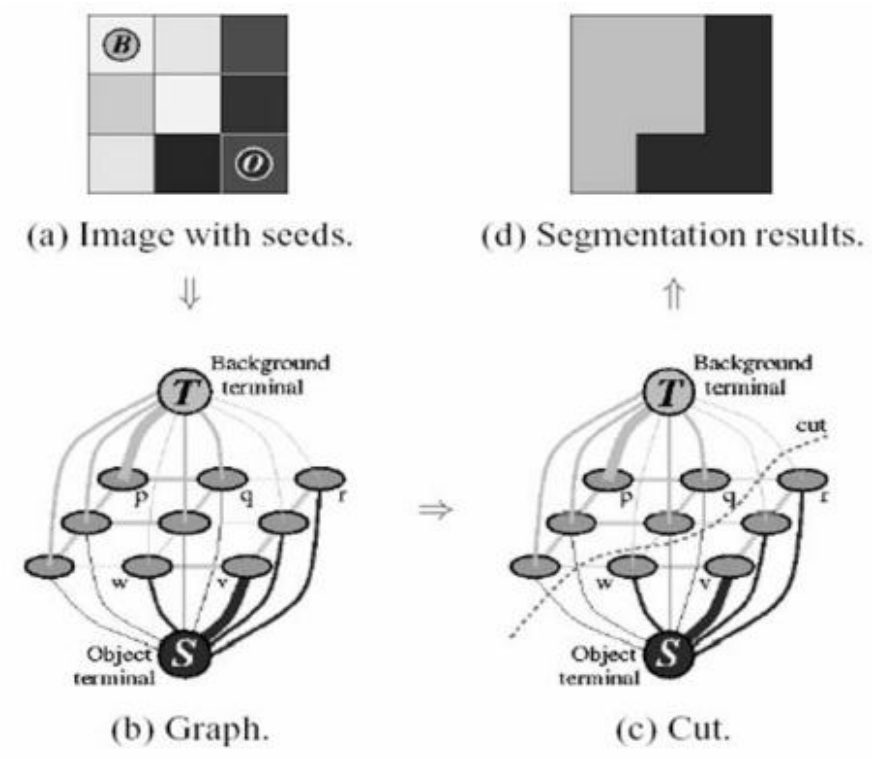
见下图。第一名球员和足球被包围在一个蓝色矩形中。然后进行一些具有白色笔划（表示前景）和黑色笔划（表示背景）的最终修饰。我们得到了一个很好的结果。

那背景会发生什么？

- 用户输入矩形。这个矩形之外的所有东西都将被视为确定的背景（这就是之前提到的矩形应该包括所有对象的原因）。矩形内的一切都是未知的。类似地，任何指定前景和背景的用户输入都被视为硬标签，这意味着它们不会在过程中发生变化。
- 计算机根据我们提供的数据进行初始标记。它标记前景和背景像素（或硬标记）
- 现在使用高斯混合模型（GMM）来模拟前景和背景。

- 根据我们提供的数据，GMM 学习并创建新的像素分布。也就是说，未知像素被标记为可能的前景或可能的背景，这取决于其在颜色统计方面与其他硬标记像素的关系（它就像聚类一样）。
- 从该像素分布构建图形。图中的节点是像素。添加了另外两个节点，**源节点**和 **Sink 节点**。每个前景像素都连接到源节点，每个背景像素都连接到 Sink 节点。
- 连接像素到源节点/端节点的边的权重由像素是前景/背景的概率来定义。像素之间的权重由边缘信息或像素相似性定义。如果像素颜色存在较大差异，则它们之间的边缘将获得较低的权重。
- 然后使用 mincut 算法来分割图形。它将图形切割成两个分离源节点和汇聚节点，具有最小的成本函数。成本函数是被切割边缘的所有权重的总和。切割后，连接到 Source 节点的所有像素变为前景，连接到 Sink 节点的像素变为背景。
- 该过程一直持续到分类收敛为止。

如下图所示（图片提供： http://www.cs.ru.ac.za/research/g02m1682/ ）



演示

现在我们使用 OpenCV 进行抓取算法。 OpenCV 具有此功能， **cv.grabCut（）** 。我们将首先看到它的参数：

- *img* - 输入图像
- *mask* - 这是一个掩码图像，我们指定哪些区域是背景，前景或可能的背景/前景等。它由以下标志完成， **cv.GC_BGD** ， **cv.GC_FGD** ， **cv.GC_PR_BGD** ， **cv.GC_PR_FGD**，或简单地将 0,1,2,3 传递给图像。
- *rect* - 它是一个矩形的坐标，包括格式为（x，y，w，h）的前景对象

- *bdgModel* ， *fgdModel* - 这些是内部算法使用的数组。您只需创建两个大小为（n = 1.65）的 np.float64 类型零数组。
- *iterCount* - 算法运行的迭代次数。
- *模式* - 它应该是 **cv.GC_INIT_WITH_RECT**或 **cv.GC_INIT_WITH_MASK**或合并后决定我们是否正在绘图矩形或最终修饰笔画。

首先让我们看看矩形模式。我们加载图像，创建一个类似的蒙版图像。我们创建 *fgdModel* 和 *bgdModel* 。我们给出矩形参数。这一切都是直截了当的。让算法运行 5 次迭代。模式应该是 *cv.GC_INIT_WITH_RECT*，因为我们使用的是矩形。然后运行抓取。它修改了蒙版图像。在新的掩模图像中，像素将标记有四个标记，表示背景/前景，如上所述。因此，我们修改掩模，使得所有 0 像素和 2 像素都被置为 0（即背景），并且所有 1 像素和 3 像素被置为 1（即前景像素）。现在我们的最后面具准备好了。只需将其与输入图像相乘即可得到分割后的图像。

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.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)
cv.grabCut(img,mask,rect,bgdModel,fgdModel,5,cv.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，梅西的头发都没了。 *谁没有他的头发喜欢梅西？* 我们需要把它带回来。所以我们将用 1 像素（确定的前景）给出一个精细的修饰。与此同时，有些地方已经出现了我们不想要的图片，还有一些标识。我们需要删除它们。在那里我们提供一些 0 像素的修饰（确定背景）。因此，正如我们现在所说的那样，我们在前一种情

我实际上做的是，我在绘画应用程序中打开了输入图像，并在图像中添加了另一层。在画中使用画笔工具，我在这个新图层上标记了带有黑色的白色和不需要的背景（如徽标，地面等）的前景（头发，鞋子，球等）。然后用灰色填充剩余的背

景。然后在 OpenCV 中加载该掩模图像，编辑我们在新添加的掩模图像中使用相应值的原始掩模图像。检查以下代码：

```
# newmask is the mask image I manually labelled
newmask = cv.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 = cv.grabCut(img,mask,None,bgdModel,fgdModel,5,cv.GC_INIT_WIT
mask = np.where((mask==2)|(mask==0),0,1).astype('uint8')
img = img*mask[:,:,np.newaxis]
plt.imshow(img),plt.colorbar(),plt.show()
```

看下面的结果：

就是这样了。这里不是在 rect 模式下初始化，而是直接进入掩码模式。只需用 2 像素或 3 像素（可能的背景/前景）标记蒙版图像中的矩形区域。然后像我们在第二个例子中那样用 1 像素标记我们的 sure_foreground。然后直接应用具有掩码模式的 grabCut 函数。

其他资源

练习

1. OpenCV 示例包含一个示例 grabcut.py，它是一个使用 grabcut 的交互式工具。核实。另请观看 youtube 视频如何使用它。
2. 在这里，您可以将其制作成交互式样本，使用鼠标绘制矩形和笔划，创建轨迹栏以调整笔划宽度等。

理解特征

目标

在本章节中， 我们将理解什么是特征以及为什么他们这么重要，还有边缘（角落）如此重要等等。

解释说明

我们大多数人都玩过拼图游戏。我们获得由大量图片碎片，并且需要将他们组合起来形成一张完整的图片。问题来了，我们如何实现呢？ 我们如何将相同的理论投射到计算机程序中，以便计算机可以玩拼图游戏？如果计算机可以玩拼图游戏了，为什么我们不能给计算机提供很多真实的自然风景图像，并告诉它将所有这些图像拼接成一个大的单一图像？如果计算机可以将多个自然图像拼接成一个，那么如何提供大量建筑物或任何结构的图片并告诉计算机从中创建 3D 模型呢？

那么，问题和想象力还在继续。但这一切都取决于最基本的问题：你如何玩拼图游戏？你如何将大量的混乱图像片段排列成一个大的单个图像？如何将大量自然图像拼接到单个图像中？

答案是，我们正在寻找独特的特定模式或特定特征，可以轻松跟踪和比较。如果我们找到这样一个特征的定义，我们可能会发现很难用文字表达，但我们知道它们是什么。如果有人要求您指出可以在多个图像之间进行比较的一个好的功能，您可以指出一个。这就是为什么即使是小孩子也可以简单地玩这些游戏。我们在图像中搜索这些特征，找到它们，在其他图像中查找相同的特征并对齐它们。而已。 （在拼图游戏中，我们更多地关注不同图像的连续性）。所有这些能力都存在于我们身上。

因此，我们的一个基本问题扩展到更多，但变得更具体。这些功能是什么？ （答案对于计算机也应该是可以理解的。）

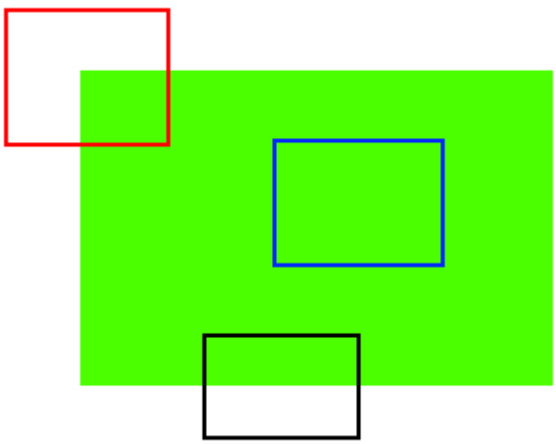
很难说人类是如何找到这些特征。此能力在人类大脑中早已根深蒂固。但是如果我们深入研究一些图片并搜索不同的图案，我们会发现一些有趣的东西。例如，如下图所示：

图像非常简单。在图像的顶部，给出了六个小图像补丁。你的问题是如何在原始图像中找到这些补丁的确切位置。你能找到多少正确的结果？

A 和 B 是平坦的表面，它们分布在很多区域。找到这些补丁的确切位置是很难的。

C 和 D 要简单得多。它们是建筑物的边缘。您可以找到一个大概的位置，但确切的位置仍然很困难。这是因为沿着边缘的模式是相同的。然而，在边缘，它是不同的。因此，与平坦区域相比，边缘是更好的特征，但是不够好（用于比较边缘的连续性在拼图中是好的）。

最后，E 和 F 是建筑物的一些角落。它们很容易找到。因为在角落，无论你移动这个补丁，它都会有所不同。所以他们可以被认为是很好的特征。所以现在我们来看看更简单（和广泛使用的图像）以便更好地理解。



就像上面一样，蓝色斑块是平坦的区域，很难找到和跟踪。无论你移动蓝色补丁，它看起来都一样。黑色贴片有边缘。如果沿垂直方向（即沿着渐变方向）移动它会改变。沿边缘移动（平行于边缘），看起来一样。对于红色补丁，它是一个角落。

无论您移动补丁，它看起来都不同，意味着它是独一无二的。所以基本上，角被认为是图像中的好特征。（不仅仅是角落，在某些情况下，blob 被认为是很好的特征）。

所以现在我们回答了我们的问题，“这些特征是什么？”。但接下来的问题出现了。我们如何找到它们？或者我们如何找到角落？我们以直观的方式回答了这一点，即在图像中寻找在其周围的所有区域中移动（少量）时具有最大变化的区域。在接下来的章节中，这将被投射到计算机语言中。因此，查找这些图像特征称为特征检测。

我们在图像中找到了这些特征。一旦找到它，您应该能够在其他图像中找到相同的内容。这是怎么做到的？我们在这个特征周围区域，我们用自己的话来解释，比如“上部是蓝天，下部是建筑物的区域，那个建筑物有玻璃等”，你在另一个地方寻找相同的区域图片。基本上，您正在描述该功能。类似地，计算机也应该描述特征周围的区域，以便它可以在其他图像中找到它。所谓的描述称为特征描述。获得这些功能及其描述后，您可以在所有图像中找到相同的功能并对齐它们，将它们拼接在一起或做任何您想做的事情。

因此，在本单元中，我们正在寻找 OpenCV 中的不同算法来查找功能，描述功能，匹配它们等。

额外资源

练习

Harris 角点检测

目标

在这一章当中，

- 我们将了解 Harris 角点检测背后的概念。
- 我们将看到函数： **cv.cornerHarris()** ， **cv.cornerSubPix()**

理论

在上一章中，我们看到角点是图像中的区域，在所有方向上的强度变化都很大。找到这些角点的一个早期尝试出现在**Chris Harris 和 Mike Stephens**1988 年的论文**A Combined Corner and Edge Detector**中，因此现在它被称为 Harris 角点探测器。他把这个简单的想法变成了数学形式。基本原理是将窗口向各个方向上位移$$(u,v)$$，然后计算滑动前后强度差异的总和。表达式如下：

$$E(u,v) = \sum *{x,y}* *underbrace{w(x,y)}* {window function} [\underbrace{I(x+u,y+v)} *{shifted intensity}*-*underbrace{I(x,y)}* {intensity}] ^ 2$$

窗口函数可以是矩形窗口，也可以是对每个像素赋予不同权重的高斯窗口。

我们必须使$$E(u,v)$$的值最大化以进行角点检测。这意味着，必须最大化方程右侧的第二项。将泰勒展开应用于上述方程并进行一些数学转换（请参考您喜欢的任何标准教科书进行完全推导），我们得到最终的等式：

$$E(u,v) \approx \begin {bmatrix} u&v \end {bmatrix} M \begin {bmatrix} u \ v \end {bmatrix}$$

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$$I_y $$分别是 x 和 y 方向的图像导数。（可以使用 **cv.Sobel()** 计算得到）。

然后是主要部分。在此之后，他们建立了一个等式来评分，从而判断窗口中是否含有角点：

$$R = det(M)-k(trace(M))^ 2$$

其中

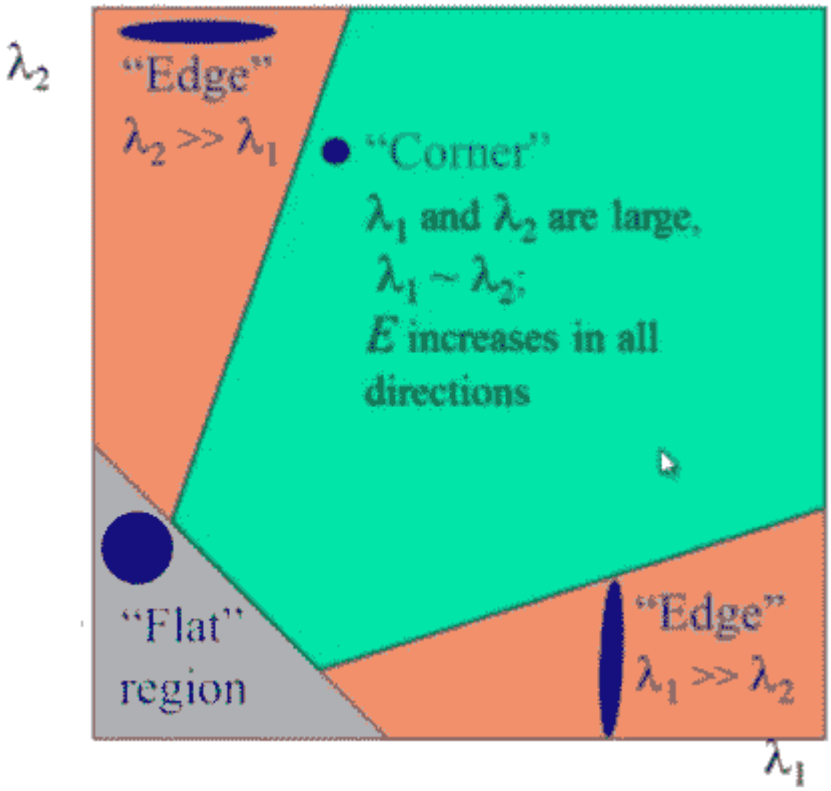
- $$det(M)=\lambda_1 \lambda_2$$
- $$trace(M)= \lambda_1+\lambda_2$$
- $$\lambda_1$$和$$\lambda_2$$是 M 的特征值

根据这些特征值可以判断一个区域是角点，边缘还是平面。

- 当$$\lambda_1$$和$$\lambda_2$$都很小时，$$| R |$$也很小，该区域是平面。

- 当$$\lambda_1\gt\gt\lambda_2$$或者$$\lambda_2\gt\gt\lambda_1$$时，$$R\lt0$$，该区域是边缘。
- 当$$\lambda_1$$和$$\lambda_2$$大且$$\lambda_1 \sim \lambda_2$$时，$$R$$很大，该区域是一个角点。

上述结论可以用下图表示：



因此，Harris 角点检测的结果是具有这些分数的灰度图像。选取适当的阈值即可筛选出图像中的角点。我们将用一个简单的图像来进行演示。

OpenCV 中的 Harris 角点检测器

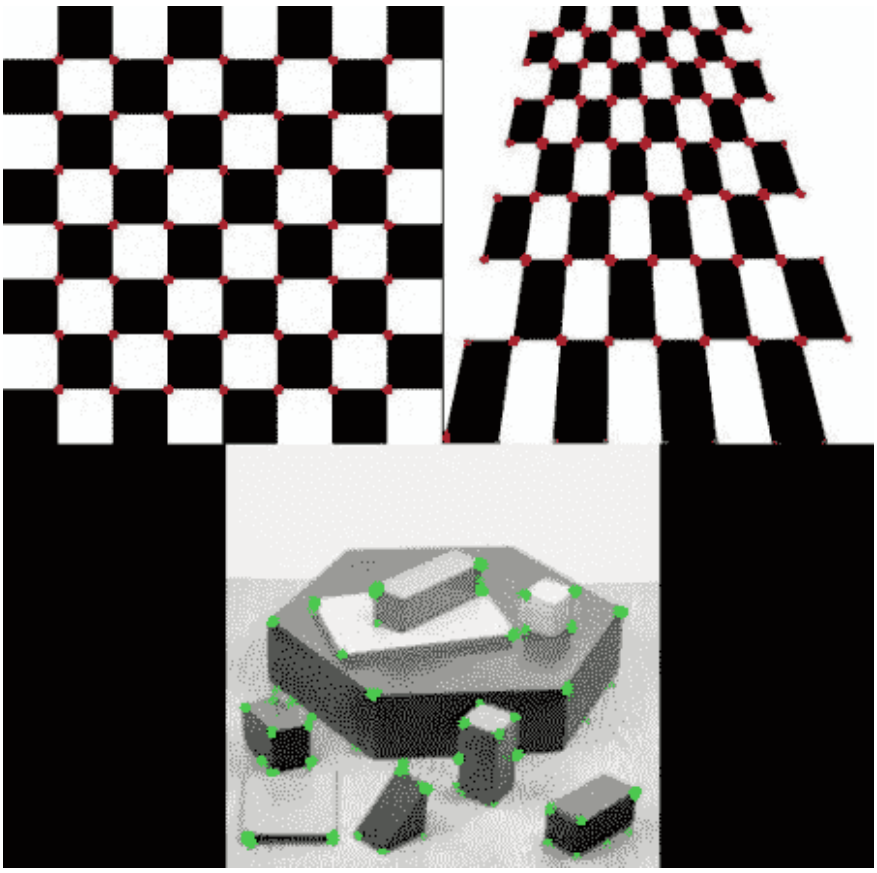
OpenCV 中的 **cv.cornerHarris()** 函数用来实现 Harris 角点检测。它的参数是：

- **img** - 输入图像，应为 float32 类型的灰度图。
- **blockSize** - 角点检测所考虑的邻域大小。
- **ksize** - Sobel 导数的内核大小。
- **k** - Harris 检测器方程中的自由参数。

请参阅以下示例：

```
import numpy as np
import cv2 as cv
filename = 'chessboard.png'
img = cv.imread(filename)
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
gray = np.float32(gray)
dst = cv.cornerHarris(gray,2,3,0.04)
#result is dilated for marking the corners, not important
dst = cv.dilate(dst,None)
# Threshold for an optimal value, it may vary depending on the image.
img[dst>0.01*dst.max()]=[0,0,255]
cv.imshow('dst',img)
if cv.waitKey(0) & 0xff == 27:
    cv.destroyAllWindows()
```

以下是三个结果：

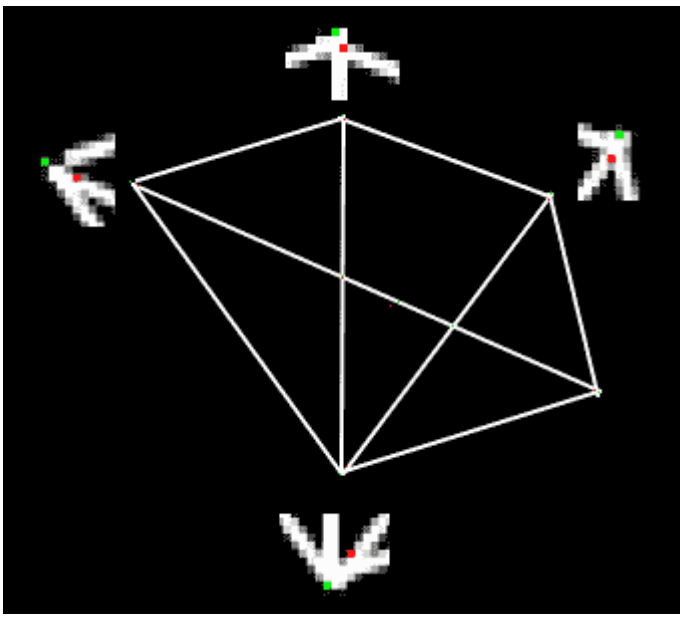


具有亚像素级精度的角点

有时，您可能需要以最高精度找到角点。OpenCV 带有一个函数 **cv.cornerSubPix()**，它进一步细化了角点检测，以达到亚像素级精度。以下是一个例子。像往常一样，我们需要先找到 Harris 角点。然后将这些角的质心（角点处可能有一堆像素，我们采用它们的质心）传递给该函数来细化它们。Harris 角点以红色像素标记，细化后的角点以绿色像素标记。对于此函数，我们必须定义迭代停止的条件。我们在经过指定次数的迭代或达到一定精度后停止它，以先发生者为准。我们还需要定义进行角点搜索的邻域大小。

```
import numpy as np
import cv2 as cv
filename = 'chessboard2.jpg'
img = cv.imread(filename)
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
# find Harris corners
gray = np.float32(gray)
dst = cv.cornerHarris(gray,2,3,0.04)
dst = cv.dilate(dst,None)
ret, dst = cv.threshold(dst,0.01*dst.max(),255,0)
dst = np.uint8(dst)
# find centroids
ret, labels, stats, centroids = cv.connectedComponentsWithStats(dst)
# define the criteria to stop and refine the corners
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 100, 0.001)
corners = cv.cornerSubPix(gray,np.float32(centroids),(5,5),(-1,-1),criteria)
# Now draw them
res = np.hstack((centroids,corners))
res = np.int0(res)
img[res[:,1],res[:,0]]=[0,0,255]
img[res[:,3],res[:,2]] = [0,255,0]
cv.imwrite('subpixel5.png',img)
```

下面是结果，其中的一些重要位置进行了缩放：



其他资源

练习

Shi-Tomasi 角点检测和追踪的良好特征

目标

在这一章当中，

- 我们将了解另一个角落探测器：Shi-Tomasi 角落探测器
- 我们将看到函数： **cv.goodFeaturesToTrack()**

理论

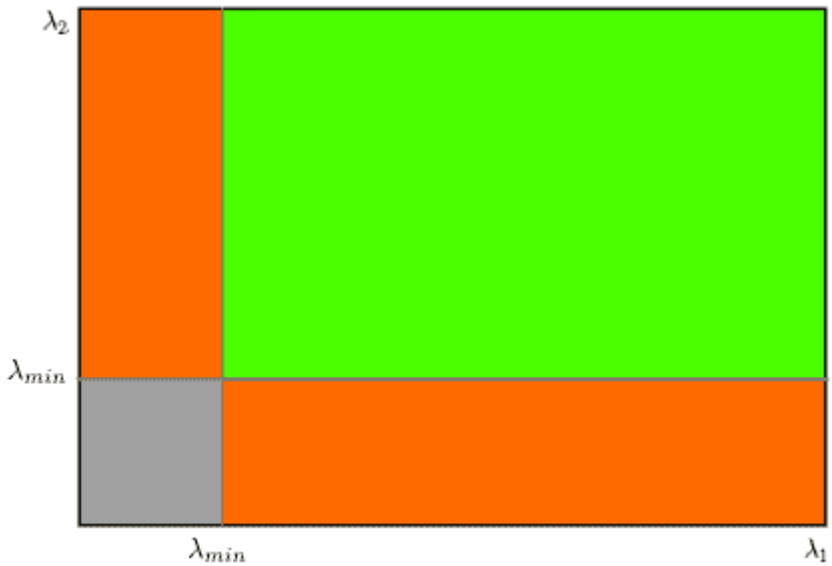
在上一章中，我们看到了 Harris 角点检测。 之后在 1994 年，J. Shi 和 C. Tomasi 在他们的论文 **Good Features to Track** 中做了一个小修改，与 Harris 角点检测相比显示出更好的结果。Harri 角点检测的评分由下式给出：

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^ 2$$

不同于此，Shi-Tomasi 提出：

$$R = min(\lambda_1,\lambda_2)$$

如果它大于阈值，则将其视为角点。如果我们像在 Harris 角点检测中那样将它绘制在$$\lambda_1- \lambda_2$$空间中，将得到如下图像：



从图中可以看出，只有当$$\lambda*1$$和$$\lambda_2$$都大于最小值$$\lambda*{min}$$时，它才被视为一个角点（绿色区域）。

代码

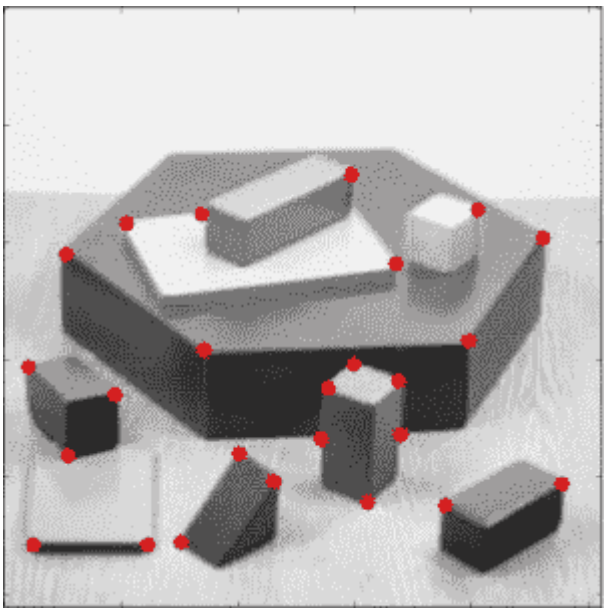
OpenCV 有一个函数， **cv.goodFeaturesToTrack()** 。它通过 Shi-Tomasi 方法（或 Harris 角点检测，如果你指定它）在图像中找到 N 个最佳的角点。像往常一样，图像应该是灰度图像。然后指定要查找的角点数量。然后指定质量等级，该等级是 0-1 之间的值，所有低于这个质量等级的角点都将被忽略。最后设置检测到的两个角点之间的最小欧氏距离。

通过所有这些信息，该函数可以在图像中找到角点。所有低于质量等级的角点都将被忽略。然后它根据质量按降序对剩余的角点进行排序。该函数选定质量等级最高的角点（即排序后的第一个角点），忽略该角点最小距离范围内的其余角点，以此类推最后返回 N 个最佳的角点。

在下面的例子中，我们将尝试找到 25 个最佳角点：

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('blox.jpg')
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
corners = cv.goodFeaturesToTrack(gray,25,0.01,10)
corners = np.int0(corners)
for i in corners:
    x,y = i.ravel()
    cv.circle(img,(x,y),3,255,-1)
plt.imshow(img),plt.show()
```

结果如下：



以后我们会发现这个函数更适合在目标追踪中使用。

其他资源

练习

SIFT 简介（尺度不变特征变换）

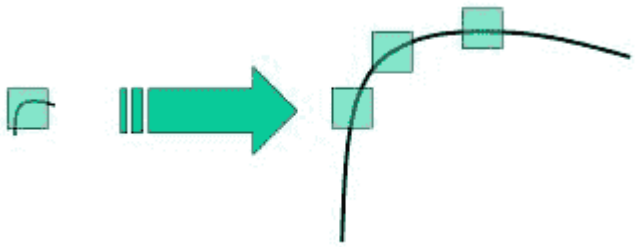
目标

在这一章当中，

- 我们将了解 SIFT 算法的概念
- 我们将学习如何找到 SIFT 特征点和描述子。

理论

在前几章中，我们看到了一些角点检测器，如 Harris 等。它们具有旋转不变性，这意味着，即使图像旋转，我们也可以找到相同的角点。很明显，因为角点在旋转图像中也是角点。但是缩放呢？如果图像缩放，角点可能就不再是角点。以下图为例，在小图像中使用一个小窗口能够检测出角点，然而将图像放大后，在同一窗口中的图像变得平坦，无法检测出角点。所以 Harris 角点不是尺度不变的。



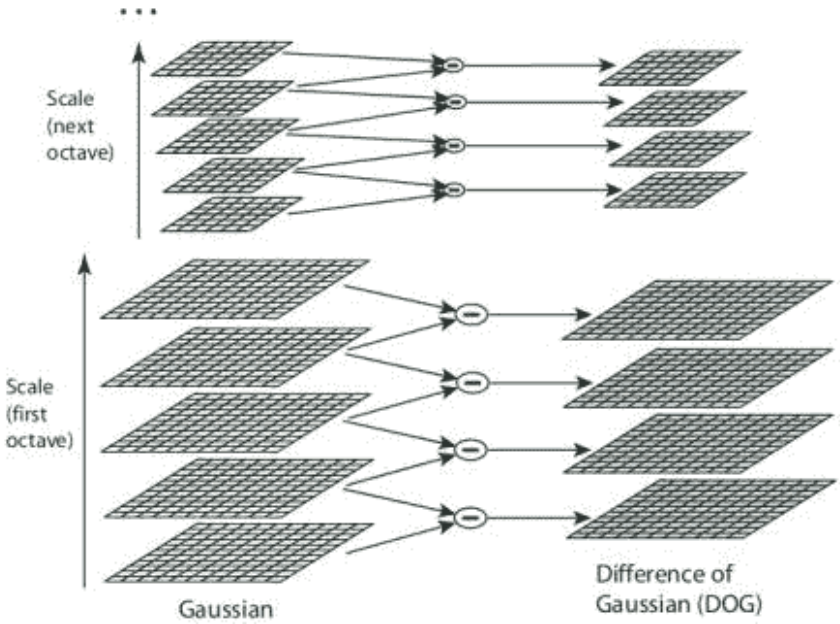
因此，在 2004 年，不列颠哥伦比亚大学的 **D.Lowe** 在他的论文**Distinctive Image Features from Scale-Invariant Keypoints**中提出了一种新算法，尺度不变特征变换（SIFT），用以提取特征点并计算其描述子。 *（这篇论文易于理解，被认为关于 SIFT 的最佳材料。以下解释只是对该论文的简短摘要）*。

SIFT 算法主要涉及四个步骤。我们将逐一看到它们。

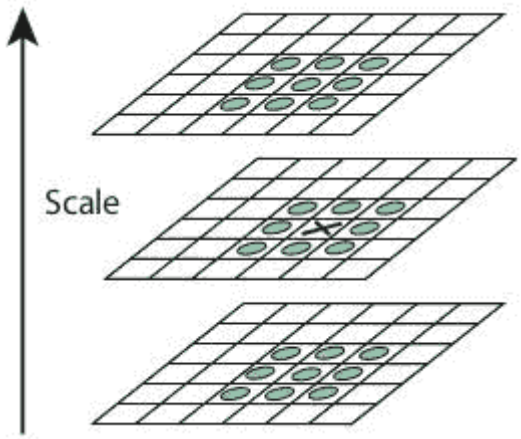
1.尺度空间极值检测

从上图可以看出，我们不能使用同一个窗口来检测不同尺度空间中的角点。检测小的角点可以用小的窗口，但检测更大的角点则需要更大的窗口。为此需要进行尺度空间滤波。使用不同$$\sigma$$值的高斯拉普拉斯算子（LoG）对图像进行卷积。具有不同$$\sigma$$值的 LoG 可以检测不同大小的斑点。简而言之，$$\sigma$$相当于一个尺度变换因子。例如，在上图中，使用具有低$$\sigma$$的高斯核可以检测出小的角点，而具有高$$\sigma$$的高斯核则适合于检测大的角点。因此，我们可以在尺度空间和二维平面中找到局部最大值$$(x, y, \sigma)$$，这意味着在$$\sigma$$尺度中的(x,y)处有一个潜在的特征点。

但是 LoG 的计算量非常大，因此 SIFT 算法使用高斯差分，这是 LoG 的近似值。分别使用方差值为$$\sigma$$和$$k\sigma$$的高斯核对图像进行模糊，对所得的结果再求二者的差值就是 DoG。如下图所示：



找到 DoG 后，搜索不同尺度空间和二维平面中的局部最大值。例如，将图像中的一个像素与其 8 领域、尺度空间上一层中相邻的 9 个像素以及尺度空间下一层中相邻的 9 个像素做比较。如果它是一个局部最大值，那么它就是一个潜在的特征点。基本上可以说特征点是相应尺度空间的最佳代表。如下图所示：



对于参数的取值论文中给出了一些经验数据，可以概括为：octaves= 4，尺度等级数为 5，初始$$\sigma$$ = 1.6 ，$$k = \sqrt {2} $$作为最佳值。

2.特征点定位

一旦找到潜在的特征点，就必须对其进行细化以获得更准确的结果。论文作者使用尺度空间的泰勒展开来获得更准确的极值位置，并且如果此极值处的强度小于阈值（0.03），则将其忽略。该阈值在 OpenCV 中被称为**contrastThreshold**。

DoG 对边缘更加敏感，因此需要去除边缘。为此，使用了类似于 Harris 角点检测的思路。论文作者使用 2x2 Hessian 矩阵（H）来计算主曲率。我们从 Harris 角点检测得知，当一个特征值大于另一个特征值时检测到的是边界。所以这里论文作者使用了一个简单的函数，如果比率大于阈值则丢弃该特征点，在 OpenCV 中这个比率被称为 **edgeThreshold** 。论文中给出的边界阈值是 10。

所以，任何低对比度特征点点和边缘特征点被去除后，剩下的就是我们所感兴趣的特征点。

3.指定方向

现在，为每个特征点指定方向，以实现图像旋转的不变性。获取特征点所在的尺度空间的领域，并且在计算该区域的梯度大小和方向。创建了一个包含 36 个 bins（每 10°一个 bin，覆盖了 360°）的方向直方图（由梯度幅度和圆形高斯窗口加权，其中高斯窗口的$$\sigma$$等于当前特征点尺度空间$$\sigma$$的 1.5 倍）。采用直方图中的最高峰为主方向，同时也考虑其余任何高于最高峰 80%的峰来计算方向。这就会创建具有相同位置和尺度空间但不同方向的特征点。这样做有助于匹配的稳定性。

4.特征点描述子

现在创建特征点描述子。在关键点周围选取 16x16 的邻域。将它为 16 个 4x4 大小的子块。对于每个子块，创建包含 8 个 bin 的方向直方图。因此总共有 128 个 bin 值可用。由这 128 个形成的向量构成了特征点描述子。除此之外，还采取了一些措施来实现对光照变化，旋转等的鲁棒性。

5.特征点匹配

通过识别两幅图像中距离最近的特征点来进行特征点匹配。但在某些情况下，第二个最接近的匹配可能非常接近第一个。它可能由噪音或其他原因而引起。在这种情况下，计算最近距离与第二最近距离的比率。如果比率大于 0.8，则忽略它们。根据论文，这样做消除了大约 90%的错误匹配，而同时只去除了 5%的正确匹配。

以上是 SIFT 算法的总结。关于更多详细信息和理解，强烈建议阅读原始论文。记住一件事，这个算法受到专利保护，所以这个算法被包含在 opencv contrib repo 中。

OpenCV 中的 SIFT

现在让我们看看 OpenCV 中提供的 SIFT 函数。让我们从特征点检测及绘制开始。首先，我们必须构造一个 SIFT 对象。我们可以使用不同的参数，这不是必须的，关于参数的解释可参考文档。

```
import numpy as np
import cv2 as cv
img = cv.imread('home.jpg')
gray= cv.cvtColor(img,cv.COLOR_BGR2GRAY)
sift = cv.xfeatures2d.SIFT_create()
kp = sift.detect(gray,None)
img=cv.drawKeypoints(gray,kp,img)
cv.imwrite('sift_keypoints.jpg',img)
```

sift.detect()函数查找图像中的特征点。如果只想搜索图像的一部分，则可以传递一个掩模。返回的每个特征点都是一个特殊的结构，它有许多属性，如(x,y)坐标，有意义邻域的大小，指定方向的角度，指定特征点强度的响应等。

OpenCV 还提供 **cv.drawKeyPoints()**函数，该函数在特征点的位置上绘制小圆圈。如果您向其传递一个标志 **cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS**，它将绘制一个特征点大小的圆圈，它甚至会显示其方向。见下面的例子。

```
img=cv.drawKeypoints(gray,kp,img,flags=cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv.imwrite('sift_keypoints.jpg',img)
```

请参阅以下两个结果：

现在要计算描述子，OpenCV 提供了两种方法。

1. 既然你已经找到了特征点，你可以调用 **sift.compute()**来计算我们找到的特征点的描述子。例如：kp,des = sift.compute(gray,kp)。
2. 如果您没有找到特征点，请使用函数 **sift.detectAndCompute()**在一个步骤中直接查找特征点和描述子。

我们将看到第二种方法：

```
sift = cv.xfeatures2d.SIFT_create()
kp, des = sift.detectAndCompute(gray,None)
```

这里 kp 是一个特征点列表，des 是一个 numpy 数组，该数组大小是特征点数目乘 128。

所以我们得到了特征点，描述子等。现在我们想看看如何匹配不同图像中的特征点。我们将在接下来的章节中学习。

其他资源

练习

SURF 简介（加速鲁棒特性）

目标

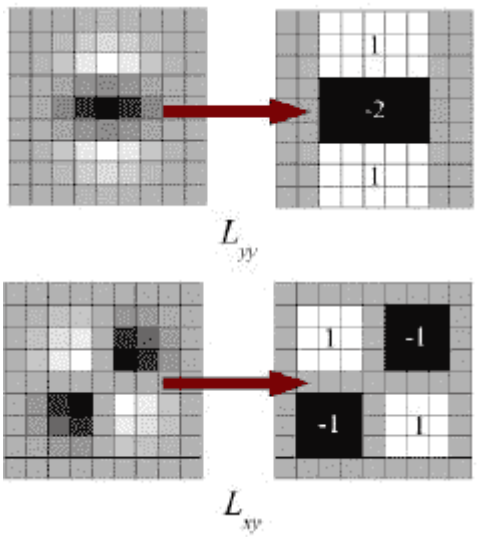
在这一章当中，

- 我们将看到 SURF 的基础知识
- 我们将看到 OpenCV 中的 SURF

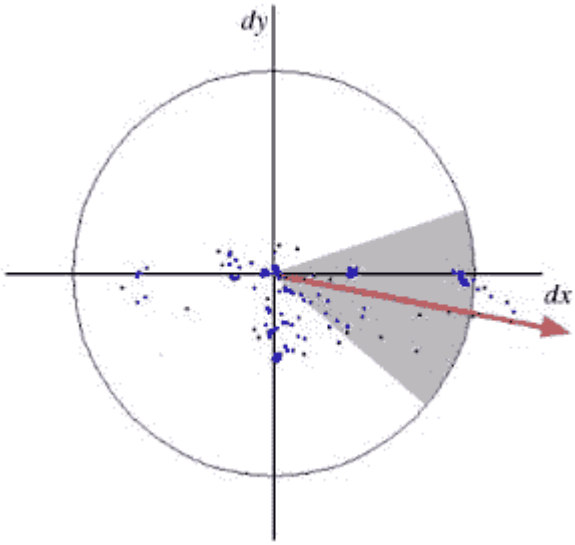
理论

在上一章中，我们看到了使用 SIFT 算法进行特征点的检测和描述。但它相对较慢，人们需要更加快速的算法。 2006 年，Bay，H.，Tuytelaars，T.和 Van Gool，L 三人发表了另一篇论文“SURF：Speeded Up Robust Features”，引入了一种名为 SURF 的新算法。顾名思义，它是 SIFT 的加速版本。

在 SIFT 中，Lowe 用高斯差分（DoG）去近似高斯拉普拉斯算子（LoG），从而构造尺度空间。 SURF 则更进一步，用盒子滤波器去近似 LoG。下图显示了这种近似。这种近似的一个很大的优点是，借助积分图像可以很容易地计算出盒子滤波器的卷积，而且可以在不同的尺度空间同时进行计算。同样，SURF 依赖于 Hessian 矩阵的行列式去计算特征点的尺度和位置。



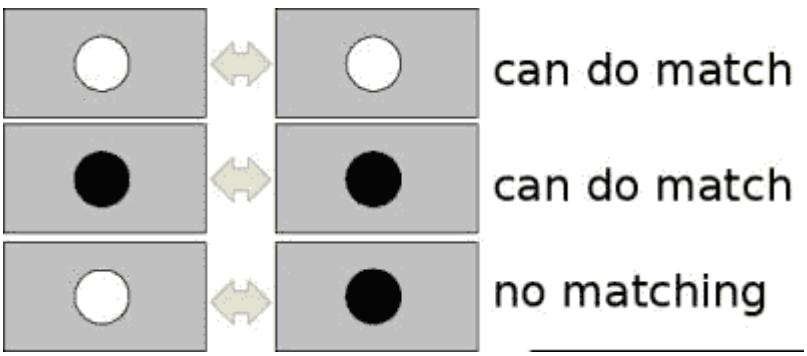
对于主方向，在 SURF 中是对以特征点为中心的 6s（s 为特征点的尺度）为半径的圆形区域内的图像进行 Haar 小波响应运算得到的。使用高斯函数对小波响应进行加权。然后将它们绘制在下图中给出的图像中。通过计算角度为 60 度的定向滑动窗口内的所有响应的总和来估算主方向。有趣的是，任意尺度空间上的小波响应都可以很容易地使用积分图像找到。对于许多应用，不需要旋转不变性，因此无需找到此方向，从而加快了过程。 SURF 提供被称为 Upright-SURF 或 U-SURF 的功能。它可以提高速度，并且保持了对$$\pm 15 ^ {\circ} $$旋转的鲁棒性。OpenCV 支持这两种模式，取决于**upright**标志位。如果标志位为 0，则计算方向。如果为 1，则不计算方向同时速度更快。



对于特征描述，SURF 在水平和垂直方向上使用小波响应（再次，使用积分图像使事情变得更容易）。在特征点周围获取大小为 20sX20s 的邻域，其中 s 是特征点尺度大小。将它分为 4x4 子区域。对于每个子区域，采用水平和垂直小波响应，并形成一个像这样的矢量，$$v =（\sum {d_x}，\sum {d_y}，\sum {| d_x |}，\sum {| d_y | }）$$。当表示为矢量时，这就是 64 维的 SURF 特征点描述子。（与 SIFT 相比）描述子的维度降低了，计算和匹配的速度提高了，但又提供了更具有独特性的特征。

为了增加特征点的独特性，SURF 特征点描述子具有扩展的 128 维版本。 在$$d_y\lt 0$$和$$d_y \geq 0$$时分别计算$$d_x$$和$$| d_x | $$的和。类似地，$$d_y $$和$$| d_y | $$的和也根据$$d_x $$的符号进行区分，从而使特征数量加倍，但不会增加太多的计算复杂性。 OpenCV 支持将 **extended**标志位的值分别设置为 0 和 1，0 为 64 维，1 为 128 维（默认为 128 维）。

另一个重要的改进是使用拉普拉斯算子（Hessian 矩阵的迹）作为潜在特征点。它不增加计算成本，因为它已经在检测期间计算出来。拉普拉斯算子的符号将黑暗背景上的明亮斑点与相反情况区分开来。在匹配阶段，我们只比较具有相同类型对比度的特征（如下图所示）。这种方法可以更快地匹配，而不会降低描述子的性能。



简而言之，SURF 采用了许多方法来提高每一步的速度。分析显示在性能相当的情况下它比 SIFT 快 3 倍。 SURF 擅长处理模糊和旋转的图像，但不善于处理视角变化和光照变化。

OpenCV 中的 SURF

OpenCV 就像 SIFT 一样提供 SURF 功能。您使用一些可选参数（如 64/128-dim 描述符，Upright / Normal SURF 等）构造 SURF 对象。所有详细信息都在文档中进行了详细说明。然后就像我们在 SIFT 中所做的那样，我们可以使用 SURF.detect()，SURF.compute()等来查找特征点和描述子。

首先，我们将看到一个关于如何查找 SURF 特征点和描述子并绘制它的简单演示。由于与 SIFT 相同因此所有示例都显示在 Python 终端中。

```
>>> img = cv.imread('fly.png',0)
# Create SURF object. You can specify params here or later.
# Here I set Hessian Threshold to 400
>>> surf = cv.xfeatures2d.SURF_create(400)
# Find keypoints and descriptors directly
>>> kp, des = surf.detectAndCompute(img,None)
>>> len(kp)
 699
```

1199 个关键点太多，无法在图片中显示。我们将它减少到大约 50 以将其绘制在图像上。在匹配时，我们可能需要所有这些功能，但现在不需要。所以我们增加了 Hessian 阈值。

```
# Check present Hessian threshold
>>> print( surf.getHessianThreshold() )
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.setHessianThreshold(50000)
# Again compute keypoints and check its number.
>>> kp, des = surf.detectAndCompute(img,None)
>>> print( len(kp) )
47
```

现在小于 50 了，让我们在图像上绘制。

```
>>> img2 = cv.drawKeypoints(img,kp,None,(255,0,0),4)
>>> plt.imshow(img2),plt.show()
```

请参阅下面的结果。你可以看到 SURF 更像是斑点检测器。它检测到蝴蝶翅膀上的白色斑点。您可以使用其他图像进行测试。

现在我想试一下 U-SURF，不检测特征点的方向。

```
# Check upright flag, if it False, set it to True
>>> print( surf.getUpright() )
False
>>> surf.setUpright(True)
# Recompute the feature points and draw it
>>> kp = surf.detect(img,None)
>>> img2 = cv.drawKeypoints(img,kp,None,(255,0,0),4)
>>> plt.imshow(img2),plt.show()
```

请参阅下面的结果。所有方向都以相同的方向显示。它比以前更快。如果您的工作中不需要检测方向（如全景拼接）等，这种方法更好。

最后，我们检查描述符大小，如果它只有 64 维，则将其更改为 128。

```
# Find size of descriptor
>>> print( surf.descriptorSize() )
64
# That means flag, "extended" is False.
>>> surf.getExtended()
 False
# So we make it to True to get 128-dim descriptors.
>>> surf.setExtended(True)
>>> kp, des = surf.detectAndCompute(img,None)
>>> print( surf.descriptorSize() )
128
>>> print( des.shape )
(47, 128)
```

剩下的部分是匹配，我们将在另一章中做。

其他资源

练习

角点检测的 FAST 算法

目标

在这一章当中，

- 我们将了解 FAST 算法的基础知识
- 我们将使用 OpenCV 中的 FAST 算法找出角点。

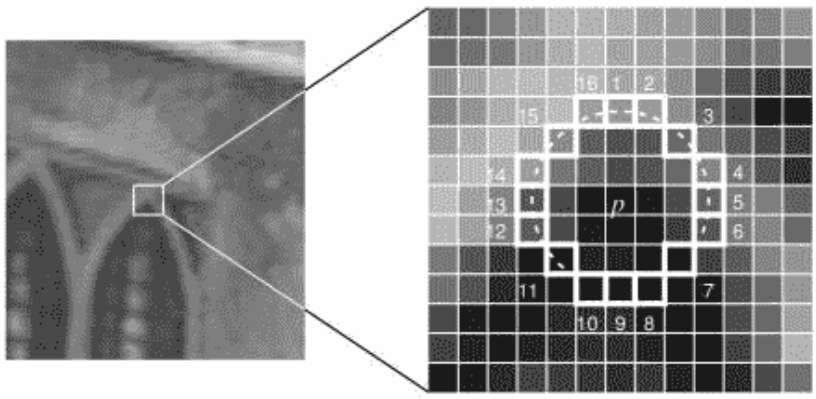
理论

我们看到了几个特征点检测器，其中很多都非常好。但从实时处理的角度来看，它们还不够快。一个最好的例子是 SLAM（Simultaneous Localization and Mapping，同时定位和地图构建）移动机器人，它们的计算能力十分有限。

为解决这一问题，Edward Rosten 和 Tom Drummond 在他们 2006 年（后来在 2010 年修订）的论文“Machine learning for high-speed corner detection”中提出了 FAST（Features from Accelerated Segment Test）算法。该算法的基本概要如下。有关详细信息，请参阅原始论文（所有图像均来自原始论文）。

使用 FAST 进行特征检测

1. 在图像中选择要待标识的像素$$p $$。让它的强度为$$I_p$$。
2. 选择适当的阈值$$t$$。
3. 考虑被测像素周围的 16 个像素的圆圈。（见下图）



4. 如果圆（16 个像素）中存在一组$$n$$个连续像素，它们都比$$I_p + t $$更亮，或者全部比$$I_p - t $$更暗，那么像素$$p$$就被认为是一个角点（如上图中白色虚线所示）。 $$n$$选取的值为 12。
5. 为了排除大量的非角点提出了一种高速测试方法。此方法仅检查 1,9,5 和 13 处的四个像素（先测试前 1 和 9。如果它们都比$$I_p + t $$更亮，或者全部比$$I_p - t $$更暗，则检查 5 和 13）。如果$$p $$是一个角点，那么其中至少

有三个必须比$$I_p + t$$更亮或比$$I_p - t $$更暗。如果这两种情况都不是，那么$$p $$不能成为一个角点。如果$$p $$可能是一个角点，再检查圆圈中的所有像素。这种检测方法本身具有很高的性能，但存在一些缺点：

- 当 n<12 时它不会丢弃很多候选点。
- 像素的选择不是最佳的，因为它的效果取决于要解决的问题和角点分布。
- 高速测试的结果被丢弃了。
- 检测到的多个特征点彼此相邻。

前 3 点可以用机器学习方法解决。最后一点使用非最大值抑制来解决。

机器学习角点检测器

1. 选择一组训练图像（最好从待应用的领域中选择）
2. 在每个图像中使用 FAST 算法找出特征点。
3. 对于每个特征点，将其周围的 16 个像素存储为一个向量。对所有的图像都这样做来构建一个特征向量$$P $$。
4. 这 16 个像素中的每个像素（例如$$x$$）可以具有以下三种状态之一：

$$S_{p\to x} = \begin{cases} d, & I_{p\to x} \leq I_p - t & \text{(darker)} \\ s, & I_p - t < I_{p\to x} < I_p + t & \text{(similar)} \\ b, & I_p + t \leq I_{p\to x} & \text{(brighter)} \end{cases}$$

5. 根据这些状态，特征向量$$P $$被细分为 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$$p $$是一个角，则为 true，否则为 false。
7. 使用 ID3 算法（决策树分类器）查询每个子集。
8. 递归计算所有子集，直到$$K_p$$的熵为零。
9. 如此创建的决策树用于在其他图像中的 FAST 检测。

非最大值抑制

在相邻位置中检测到多个特征点是另一个问题。它通过使用非最大值抑制来解决。

1. 计算所有检测到的特征点的分数$$V $$。$$V$$是$$p$$和 16 个周围像素值之间的绝对差值之和。
2. 考虑两个相邻的特征点并比较它们的$$V$$值。
3. 丢弃具有较低$$V $$值的那个。

总结

它比其他现有的角点检测器快几倍。

但在噪音很高时不够鲁棒，取决于阈值。

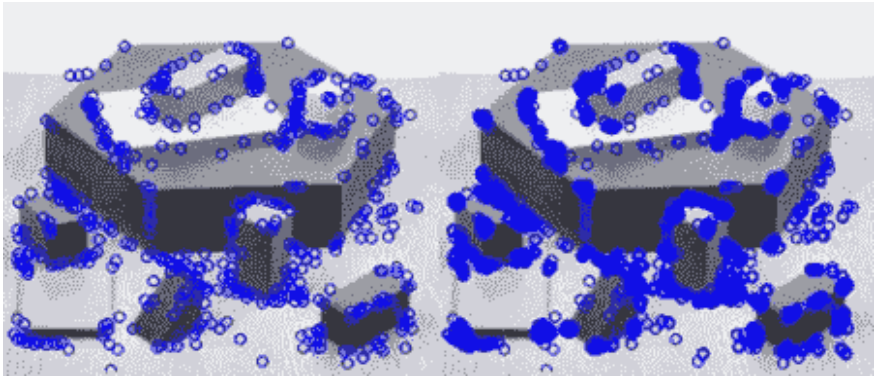
OpenCV 中的 FAST 特征点检测器

它可以像 OpenCV 中的任何其他特征点检测器一样调用。如果需要，您可以指定阈值，是否应用非最大值抑制，要使用的邻域等。

对于邻域，定义了三个标志，cv.FAST_FEATURE_DETECTOR_TYPE_5_8，cv.FAST_FEATURE_DETECTOR_TYPE_7_12 和 cv.FAST_FEATURE_DETECTOR_TYPE_9_16。下面是一个关于如何检测和绘制 FAST 特征点的简单代码。

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('simple.jpg',0)
# Initiate FAST object with default values
fast = cv.FastFeatureDetector_create()
# find and draw the keypoints
kp = fast.detect(img,None)
img2 = cv.drawKeypoints(img, kp, None, color=(255,0,0))
# Print all default params
print( "Threshold: {}".format(fast.getThreshold()) )
print( "nonmaxSuppression:{}".format(fast.getNonmaxSuppression()) )
print( "neighborhood: {}".format(fast.getType()) )
print( "Total Keypoints with nonmaxSuppression: {}".format(len(kp)) )
cv.imwrite('fast_true.png',img2)
# Disable nonmaxSuppression
fast.setNonmaxSuppression(0)
kp = fast.detect(img,None)
print( "Total Keypoints without nonmaxSuppression: {}".format(len(kp)) )
img3 = cv.drawKeypoints(img, kp, None, color=(255,0,0))
cv.imwrite('fast_false.png',img3)
```

结果如下。第一张图片使用了非最大值抑制的 FAST，而第二张没有使用：



其他资源

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

BRIEF（Binary Robust Independent Elementary Features）

目标

在这一章当中

- 我们将看到 BRIEF 算法的基础知识

理论

我们知道 SIFT 使用 128 维向量作为描述子。由于它使用浮点数，因此需要 512 个字节。类似地，SURF 也至少需要 256 个字节（对于 64 维描述子）。为数千个特征点创建这样的向量需要大量的内存，这在资源有限的情况下是不可行的，特别是嵌入式系统。内存越大，匹配所需的时间越长。

但实际匹配时可能不需要所有的维度。我们可以使用 PCA，LDA 等几种方法对其进行压缩。甚至使用 LSH（Locality Sensitive Hashing，局部敏感哈希）等其他方法也可以将浮点数格式的这些 SIFT 描述子转换为二进制字符串。对这些二进制字符串使用汉明距离进行匹配。这样速度更快，因为计算汉明距离只需要进行异或和位计数，这在具有 SSE 指令的现代 CPU 中非常快。但是我们仍然需要先找到描述符，然后才能应用哈希方法，这并不能解决我们在内存上的初始问题。

这就需要 BRIEF 算法。它提供了直接查找二进制字符串而无需找到描述子的快捷方式。它采用平滑后的图像，并以特定的方式（在文中解释）选择一组$$n_d$$(x,y)位置对。然后在这些位置对上进行一些像素强度比较。例如，让第一个位置对为$$p$$和$$q$$。如果$$I(p)\lt I(q)$$，则其结果为 1，否则为 0。对所有$$n_d $$位置对进行对比以获得$$n_d$$维二进制字符串。

$$n_d$$可以是 128，256 或 512\。OpenCV 支持所有这些值，但默认情况下，它是 256（OpenCV 以字节表示它，所以这些值将对应 16，32 和 64）。获得这些二进制字符串后就可以使用汉明距离进行匹配。

重要的一点是，BRIEF 是一个特征点描述子，它没有提供任何方法来查找这些特征点。所以你必须使用任何其他特征检测器，如 SIFT，SURF 等。本文建议使用 CenSurE，这是一个快速检测器，而且 BREIF 算法对于 CenSurE 特征点的效果比 SURF 特征点稍微好。

简而言之，BRIEF 是一种更快计算和匹配特征点描述子的方法。它能提供较高的识别率，除非存在大的平面内旋转。

OpenCV 中的 BRIEF

下面的代码展示了在 CenSurE 检测器的帮助下计算 BRIEF 描述子的方法。（CenSurE 探测器在 OpenCV 中被称为 STAR 探测器）

请注意，您需要 opencv contrib 来使用它。

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('simple.jpg',0)
# Initiate FAST detector
star = cv.xfeatures2d.StarDetector_create()
# Initiate BRIEF extractor
brief = cv.xfeatures2d.BriefDescriptorExtractor_create()
# find the keypoints with STAR
kp = star.detect(img,None)
# compute the descriptors with BRIEF
kp, des = brief.compute(img, kp)
print( brief.descriptorSize() )
print( des.shape )
```

函数 brief.getDescriptorSize()给出以字节为单位的$$n_d$$大小。默认情况下为 32。下一步是匹配，这将在另一章中完成。

其他资源

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. 维基百科的LSH（Locality Sensitive Hashing）。

ORB（Oriented FAST and Rotated BRIEF）

目标

在这一章当中，

- 我们将看到 ORB 的基础知识

理论

作为 OpenCV 爱好者，关于 ORB 最重要的是它来自“OpenCV Labs”。这个算法在 2011 年由 Ethan Rublee，Vincent Rabaud，Kurt Konolige 和 Gary R. Bradski 在他们的论文 **ORB: An efficient alternative to SIFT or SURF** 中提出的。如标题所述，它是一个很好的 SIFT 和 SURF 的替代，在计算成本，匹配性能，尤其是专利方面。是的，SIFT 和 SURF 已获得专利，您应该支付它们的使用费用。但是 ORB 不是!!!

ORB 基本上是 FAST 特征点检测器和 Brief 描述子的融合，并进行了许多修改以增强性能。首先，它使用 FAST 查找特征点，然后应用 Harris 角点的测量方法来查找其中的前 N 个点。它还使用金字塔来生成多尺度特征。但有一个问题是，FAST 不计算方向。那么旋转不变性呢？作者提出了以下修改。

它计算以角点为中心的图像块的强度加权质心。从该角点到质心的矢量方向给出了方向。为了改善旋转不变性，使用 x 和 y 计算矩，该 x 和 y 应该在半径为$$r $$的圆形区域中，其中$$r $$是图像块的大小。

现在来看描述子，ORB 使用 BRIEF 描述子。但我们已经看到，Brief 在图像旋转时表现不佳。因此，ORB 所做的是根据特征点的方向“引导”BRIEF。对于位于 $$(x*i,y_i)$$的$$n $$二进制测试的任意特征集，定义一个$$2 \times n $$矩阵，$$S $$包含这些像素的坐标。然后使用图像块的方向，$$\theta $$，找到其旋转矩阵并旋转$$S $$以获得转向（旋转）版本$$S* \theta $$。

ORB 将角度划分为$$2 \pi / 30 $$（12 度）的增量，并构建一个预先计算的 BRIEF 的查找表。只要特征点方向$$\theta $$在视图之间保持一致，就会使用正确的点集$$S_ \theta $$来计算其描述符。

BRIEF 具有一个重要特性，即每个位特征具有较大的方差，平均值接近 0.5。但是一旦它沿着特征点方向定向，它就会失去这个特性并变得更加分散。高的方差使得特征更易于分辨，因为它对输入有不同的响应。另一个理想的特性是使测试不相关，因为每次测试都会对结果产生影响。为了解决所有这些问题，ORB 在所有可能的二进制测试中运行贪婪搜索，以找到具有高方差和意味着接近 0.5 的那些，以及不相关的。结果称为 **rBRIEF** 。

对于描述子匹配，使用对传统 LSH 进行改善后的多探针 LSH。该论文称 ORB 比 SURF 和 SIFT 快得多，并且 ORB 描述子比 SURF 效果更好。 ORB 是用于全景拼接等的低功率设备的不错选择。

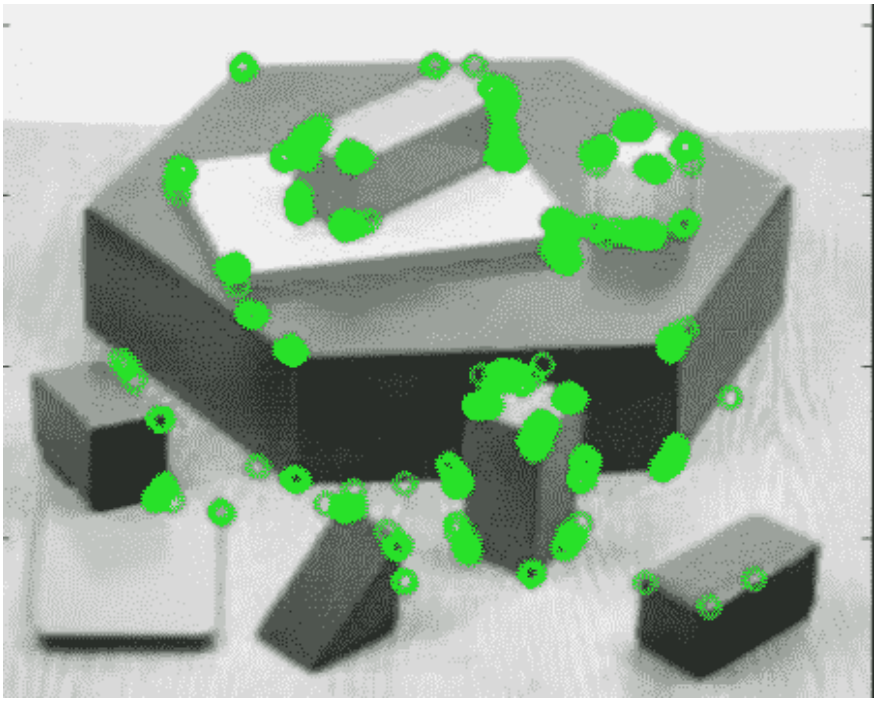
OpenCV 中的 ORB

像往常一样，我们必须使用函数 **cv.ORB()** 或使用 feature2d 通用接口创建 ORB 对象。它有许多可选参数。最有用的是 nFeatures，表示要保留的最大要素数量（默认为 500），scoreType 表示对特征点进行排序使用 Harris 得分或 FAST 得分（默认情况下为 Harris 得分）等。另一个参数 WTA_K 决定生成一个 oriented BRIEF 描述子的所用的像素点数目。默认情况下它是 2，即一次选择两个点。在这种情况下进行匹配，使用 NORM_HAMMING 距离。如果 WTA_K 为 3 或 4，则需要 3 或 4 个点来产生 BRIEF 描述子，匹配距离由 NORM_HAMMING2 定义。

下面是一个简单的代码，展示了 ORB 的用法。

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('simple.jpg',0)
# Initiate ORB detector
orb = cv.ORB_create()
# 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 = cv.drawKeypoints(img, kp, None, color=(0,255,0), flags=0)
plt.imshow(img2), plt.show()
```

看下面的结果：



ORB 特征点的匹配，我们将在另一章中做。

其他资源

1. Ethan Rublee, Vincent Rabaud, Kurt Konolige, Gary R. Bradski: ORB: An efficient alternative to SIFT or SURF. ICCV 2011: 2564-2571.

练习

特征匹配

目标

在这一章当中

- 我们将看到如何将一个图像中的特征点与其他图像进行匹配。
- 我们将在 OpenCV 中使用蛮力（Brute-Force）匹配和 FLANN 匹配

蛮力匹配的基础知识

蛮力匹配器很简单。它采用第一组中的一个特征点描述子，并使用一些距离计算与第二组中的所有其他特征点匹配。并返回距离最近的一个特征点。

对于 BF 匹配器，首先我们必须使用 **cv.BFMatcher()** 创建 BFMatcher 对象。它需要两个可选的参数。第一个是 normType。它指定要使用的距离测量方法。默认情况下，它是 cv.NORM_L2 。它很适合于 SIFT 和 SURF 等（ cv.NORM_L1 也可以）。对于基于二进制字符串的描述子，如 ORB，BRIEF，BRISK 等，应使用 cv.NORM_HAMMING ，它使用汉明距离作为度量。如果 ORB 使用 WTA_K == 3 或 4，则应使用 cv.NORM_HAMMING2 。

第二个参数是布尔变量 crossCheck，默认为 false。如果为 True，则 Matcher 仅返回具有值(i,j)的匹配,使得集合 A 中的第 i 个描述子具有集合 B 中的第 j 个描述子作为最佳匹配，反之亦然。也就是说，两组中的两个特征点应该相互匹配。它提供了一致的结果，是 D.Lowe 在 SIFT 论文中提出的比率测试的一个很好的替代方案。

一旦创建，两个重要的方法是 *BFMatcher.match()* 和 *BFMatcher.knnMatch()*。第一个返回最佳匹配。第二种方法返回 k 个最佳匹配，其中 k 由用户指定。当我们需要做更多的工作时，它可能会有用。

就像我们使用 cv.drawKeypoints()绘制特征点一样， **cv.drawMatches()** 帮助我们绘制匹配。它水平堆叠两个图像，并从第一个图像到第二个图像绘制线条，显示最佳匹配。还有 **cv.drawMatchesKnn** ，它绘制了所有 k 个最佳匹配。如果 k = 2，它将为每个关键点绘制两条匹配线。因此，如果我们想要有选择地绘制它，我们必须传递一个掩模。

让我们看一下 SURF 和 ORB 的两个例子（两者都使用不同的距离测量）。

对 ORB 描述子使用蛮力匹配

在这里，我们将看到一个关于如何匹配两个图像之间的特征的简单示例。在这种情况下，我有一个 queryImage 和一个 trainImage。我们将尝试使用特征匹配在 trainImage 中查找 queryImage。 （图片为/samples/c/box.png 和/samples/c/box_in_scene.png）

我们使用 ORB 描述符来匹配功能。所以让我们从加载图像，查找描述子等开始。

```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
img1 = cv.imread('box.png',0)          # queryImage
img2 = cv.imread('box_in_scene.png',0) # trainImage
# Initiate ORB detector
orb = cv.ORB_create()
# find the keypoints and descriptors with ORB
kp1, des1 = orb.detectAndCompute(img1,None)
kp2, des2 = orb.detectAndCompute(img2,None)
```

接下来，我们使用 cv.NORM_HAMMING （因为我们使用的是 ORB）创建一个 BFMatcher 对象并且启用了 crossCheck 以获得更好的结果。然后我们使用 Matcher.match()方法在两个图像中获得最佳匹配。我们按照距离的升序对它们进行排序，以便最佳匹配（距离最小）出现在前面。然后我们只画出前 10 个匹配（仅为了能见度，你可以随意增加匹配的个数）。

```
# create BFMatcher object
bf = cv.BFMatcher(cv.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)
# Draw first 10 matches.
img3 = cv.drawMatches(img1,kp1,img2,kp2,matches[:10], None, flags=2)
plt.imshow(img3),plt.show()
```

以下是我得到的结果：

这个匹配器对象是什么？

matches = bf.match(des1,des2)的结果是 DMatch 对象的列表。此 DMatch 对象具有以下属性：

- DMatch.distance - 描述子之间的距离。越低越好。
- DMatch.trainIdx - 目标图像中描述子的索引
- DMatch.queryIdx - 查询图像中描述子的索引

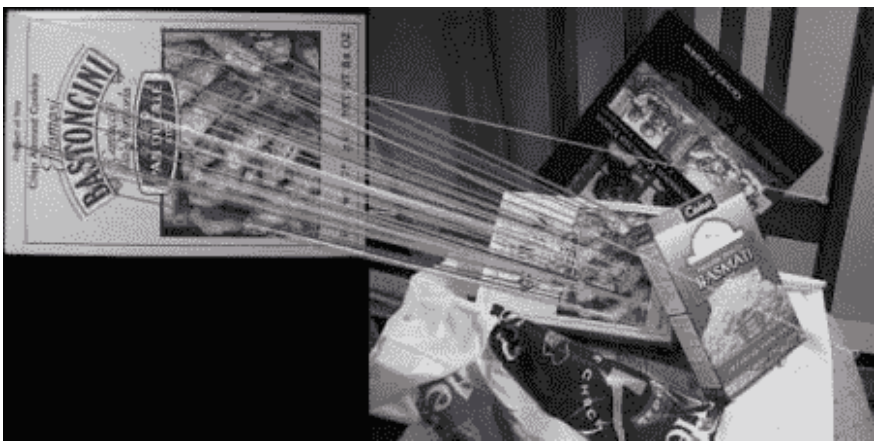
- DMatch.imgIdx - 目标图像的索引。

对 SIFT 描述符进行蛮力匹配和比率测试

这一次，我们将使用 BFMatcher.knnMatch()来获得最佳匹配。在这个例子中，我们将采用 k = 2，以便我们可以在使用 D.Lowe 论文中的比率测试。

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img1 = cv.imread('box.png',0)          # queryImage
img2 = cv.imread('box_in_scene.png',0) # trainImage
# Initiate SIFT detector
sift = cv.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 = cv.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])
# cv.drawMatchesKnn expects list of lists as matches.
img3 = cv.drawMatchesKnn(img1,kp1,img2,kp2,good,flags=2)
plt.imshow(img3),plt.show()
```

结果如下：



基于 FLANN 的匹配器

FLANN 代表快速最近邻搜索包（Fast Library for Approximate Nearest Neighbors）。它包含一系列算法，这些算法针对大型数据集中的快速最近邻搜索和高维特征进行了优化。对于大型数据集，它比 BFMatcher 工作得更快。我们将看到基于 FLANN 的匹配器的第二个示例。

对于基于 FLANN 的匹配器，我们需要传递两个字典作为参数，用来指定要使用的算法及其相关参数等。首先是 IndexParams。对于各种算法的信息在 FLANN 文档中进行了解释。总而言之，对于 SIFT 和 SURF 等算法，您可以传递以下参数：

```
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
```

使用 ORB 时，您可以传递以下参数。注释掉的参数是文献中推荐使用的，但在某些情况下不会提供所需的结果。其他值可能更好：

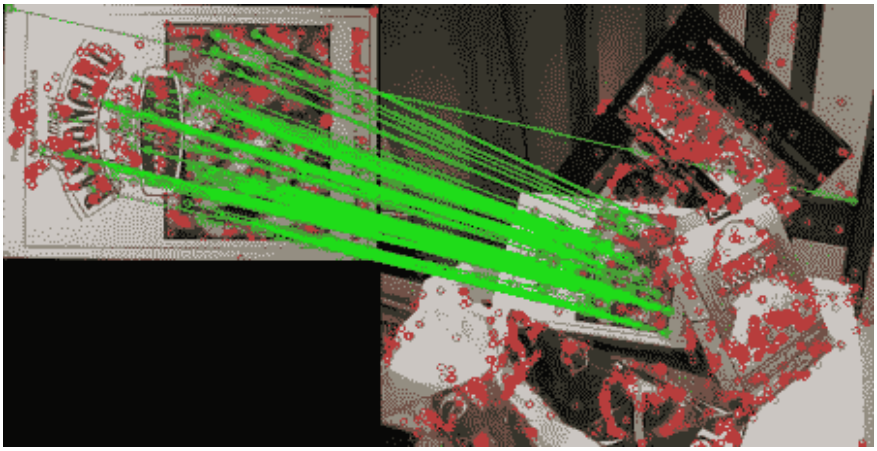
```
index_params= dict(algorithm = FLANN_INDEX_LSH,
                   table_number = 6, # 12
                   key_size = 12,     # 20
                   multi_probe_level = 1) #2
```

第二个字典是 SearchParams。它指定应递归遍历索引中的树的次数。值越高，精度越高，但也需要更多时间。如果要更改该值，请传入参数：search_params = dict（checks = 100）。

有了这些信息，我们就可以开始了。

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img1 = cv.imread('box.png',0)          # queryImage
img2 = cv.imread('box_in_scene.png',0) # trainImage
# Initiate SIFT detector
sift = cv.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 = 1
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks=50)   # or pass empty dictionary
flann = cv.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 = cv.drawMatchesKnn(img1,kp1,img2,kp2,matches,None,**draw_params)
plt.imshow(img3,),plt.show()
```

结果如下：



其他资源

练习

特征匹配+单应性查找对象

目标

在这一章当中，

- 我们将联合使用来自 calib3d 模块的特征匹配和 findHomography 来查找复杂图像中的已知对象。

基础

那我们上一节做了什么？我们使用了一个 queryImage，在其中找到了一些特征点，我们采用了另一个 trainImage，找到了该图像中的特征，最后找到它们之间特征点的最佳匹配。简而言之，我们在另一个杂乱的图像中找到了一个对象的某些部分的位置。这些信息足以在 trainImage 上准确找到对象。

为此，我们可以使用来自 calib3d 模块的函数，即 **cv.findHomography()** 。如果将两个图像中的特征点集传递给这个函数，它将找到该对象的透视变换。然后我们可以使用 **cv.perspectiveTransform()** 来查找对象。它需要至少四个正确的点来找到这种变换。

我们已经看到匹配时可能存在一些可能的错误，这可能会影响结果。为了解决这个问题，算法使用 RANSAC 或 LEAST_MEDIAN（可以由标志位决定）。因此，提供正确估计的良好匹配称为内点，剩余称为外点。 **cv.findHomography()** 返回一个指定了内点和外点的掩模。

那就让我们做吧！

代码

首先，像往常一样，让我们在图像中找到 SIFT 特征并应用比率测试来找到最佳匹配。

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
MIN_MATCH_COUNT = 10
img1 = cv.imread('box.png',0)          # queryImage
img2 = cv.imread('box_in_scene.png',0) # trainImage
# Initiate SIFT detector
sift = cv.xfeatures2d.SIFT_create()
# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks = 50)
flann = cv.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)
```

现在我们设置一个条件，即至少 10 个匹配（由 MIN_MATCH_COUNT 定义）时才查找目标对象。否则只显示一条消息，说明没有足够的匹配。

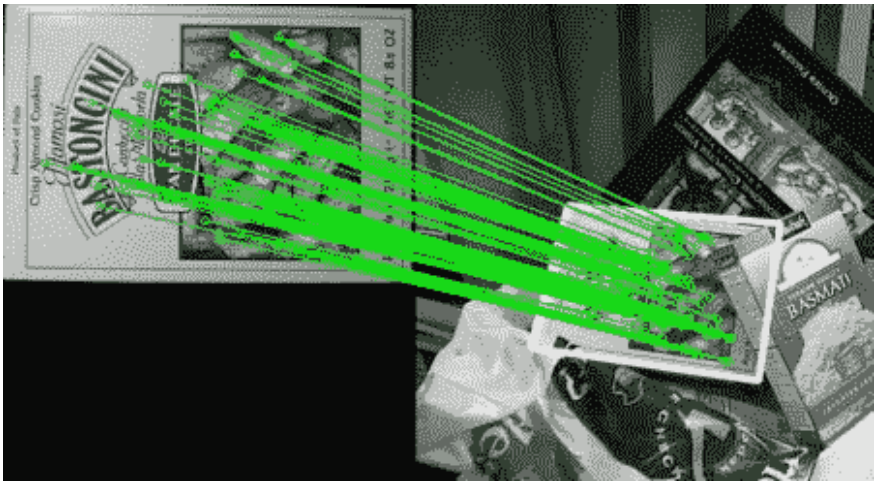
如果找到足够的匹配，我们将提取两个图像中匹配的特征点的位置。他们被传入函数中以找到透视变换。一旦我们得到这个 3x3 变换矩阵，我们就用它将 queryImage 中的角点转换为 trainImage 中的对应点。然后绘制出来。

```
if len(good)>MIN_MATCH_COUNT:
    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)
    M, mask = cv.findHomography(src_pts, dst_pts, cv.RANSAC,5.0)
    matchesMask = mask.ravel().tolist()
    h,w= img1.shape
    pts = np.float32([ [0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2)
    dst = cv.perspectiveTransform(pts,M)
    img2 = cv.polylines(img2,[np.int32(dst)],True,255,3, cv.LINE_AA)
else:
    print( "Not enough matches are found - {}/{}".format(len(good), MIN_MATCH_COUNT) )
    matchesMask = None
```

最后，我们绘制内点（如果成功找到对象）或匹配特征点（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 = cv.drawMatches(img1,kp1,img2,kp2,good,None,**draw_params)
plt.imshow(img3, 'gray'),plt.show()
```

请参阅下面的结果。对象在图像中以白色标记：



其他资源

练习

Meanshift 和 Camshift

目标

在本章中，

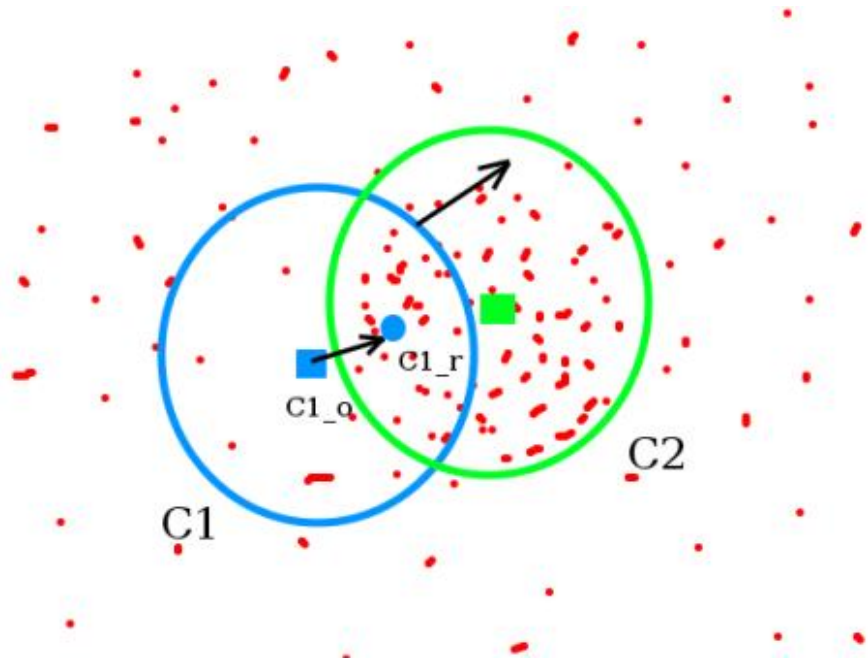
- 我们将学习如何让利用 Meanshift 和 Camshift 算法用以在视频中寻找并跟踪对象

译者注：

Meanshift 中译均值漂移聚类， Camshift 则是连续自适应 Meanshift 算法，
以下均用英文表示这两个名词。

Meanshift

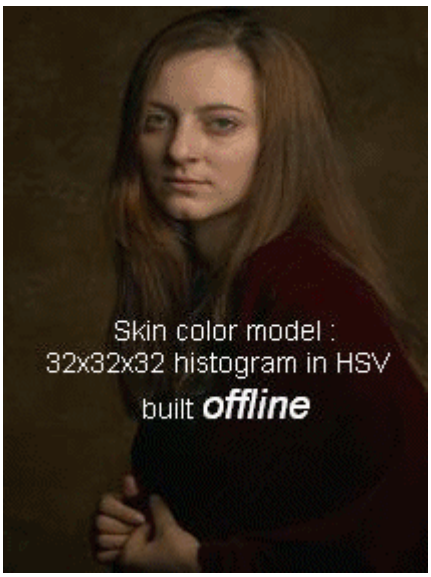
Meanshift 背后的直观感受是很简单的。想象一下，你有一堆的随机散列点(这些点可以是像直方图反向投影那样的像素分布)。同时你拥有一个小窗口(或许是一个圆圈)，现在你需要移动这个窗口直到窗口内的像素密度最大(或者说所含点的数量最多)。这在下面这幅图像中是很容易说明的：



meanshift image

起始窗口“C1”如蓝色圆圈所示，其中心区域被蓝色矩形所标记，且命名为“C1_o”。但是，如果你寻找那个窗口的质心，你将会得到点“C1_r”(用蓝色小圆圈标记)，它是窗口真正的质心。显而易见它们并不匹配。因此移动窗口似的新窗口的中心与之前的质心相匹配，此时将再次找到新的质心。最有可能的是，质心与中心两个依旧不匹配。因此，再次移动窗口，并持续迭代下去，使得窗口中心与质心落在同一位

置(或者期望误差较小)。所以最终你获得的便是一个包含有最大像素分布的窗口。将其标注为绿色圆圈，命名为“C2”。正如你在图片中看到的那样，这个窗口所包含的点的数量也是最多的。整个过程在下面的动图中有演示：



meanshift face image

所以我们通常会传递直方图反向投影图像和起始目标位置。当物体移动的时候，显然运动将会反应在直方图的反向投影图像中。最终，meanshift 算法将窗口移动到具有最大密度的新位置。

在 opencv 中使用 Meanshift

为了在 OpenCV 中使用 meanshift，首先我们需要设置目标，寻找其直方图以便我们可以对于反向投影每一帧的直方图用以利用 Meanshift 算法进行计算。我们还需要提供窗口的起始位置。对于直方图，在此时仅需考虑色相。此外，为了避免由于低亮度引起的错误值，使用cv2.inRange()函数丢弃低亮度值。

```
import numpy as np
import cv2 as cv

cap = cv.VideoCapture('slow.flv')

# 获取视频的第一帧
ret,frame = cap.read()

# 设置窗口的初始位置
r,h,c,w = 250,90,400,125  # 简单地硬编码值
track_window = (c,r,w,h)

# 设置 ROI(图像范围)以进行跟踪
roi = frame[r:r+h, c:c+w]
hsv_roi =  cv.cvtColor(roi, cv.COLOR_BGR2HSV)
mask = cv.inRange(hsv_roi, np.array((0., 60.,32.)), np.array((180.,255.,255.)))
roi_hist = cv.calcHist([hsv_roi],[0],mask,[180],[0,180])
cv.normalize(roi_hist,roi_hist,0,255,cv.NORM_MINMAX)

# 设置结束标志，10 次迭代或至少 1 次移动
term_crit = ( cv.TERM_CRITERIA_EPS | cv.TERM_CRITERIA_COUNT, 10, 1 )

while(1):
    ret ,frame = cap.read()

    if ret == True:
        hsv = cv.cvtColor(frame, cv.COLOR_BGR2HSV)
        dst = cv.calcBackProject([hsv],[0],roi_hist,[0,180],1)

        # 运行 meanshift 算法用以获取新的位置
        ret, track_window = cv.meanShift(dst, track_window, term_crit)

        # 绘制到新图像中
        x,y,w,h = track_window
        img2 = cv.rectangle(frame, (x,y), (x+w,y+h), 255,2)
        cv.imshow('img2',img2)

        k = cv.waitKey(60) & 0xff
        if k == 27:
            break
        else:
            cv.imwrite(chr(k)+".jpg",img2)
    else:
        break
cv.destroyAllWindows()
cap.release()
```

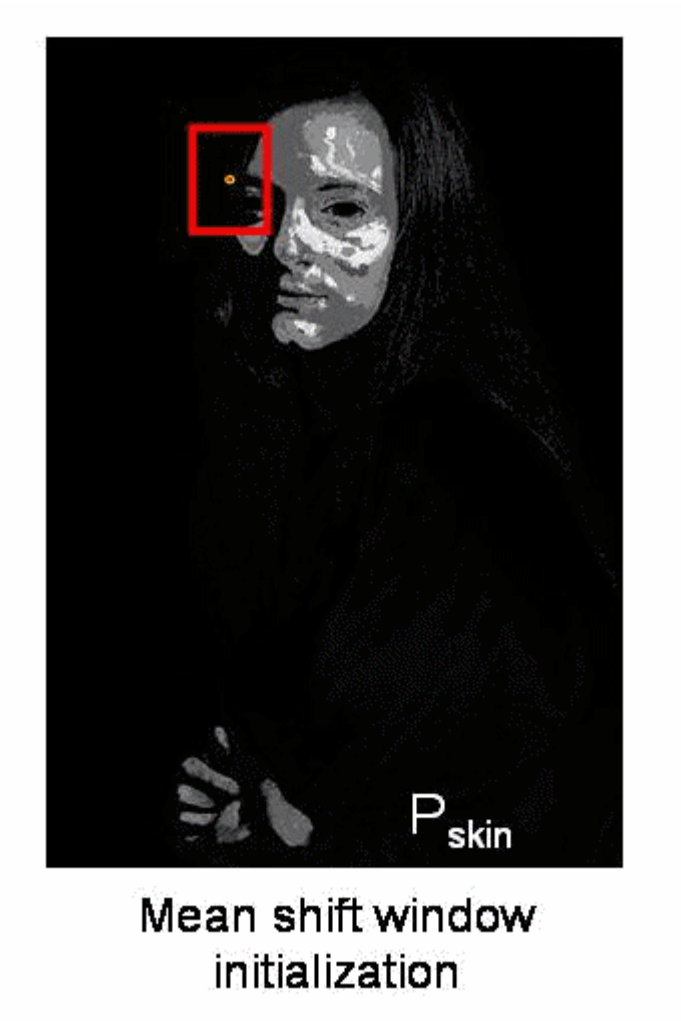
我使用的视频中的三个帧如下：

meanshift result image

Camshift

你细致的观察最后的结果了吗？这里其实有一个问题。就是无论汽车离得摄像机很远还是很近我们的窗口大小总是一样的。这并不好。我们需要根据目标的大小和旋转来适应调整窗口大小。该解决方案又一次来自“OpenCV 实验室”，这个算法被称为 CAMshift（连续自适应 Meanshift 算法），并由 Gary Bradsky 于 1998 年在他的论文“用于感知用户界面的计算机视觉面部跟踪”中发布。

这个算法首先利用了 meanshift。一旦 meanshift 收敛，他将自动将窗口的大小更新为$$s = 2 \times \sqrt{\frac{M_{00}}{256}}$$。并且寻找出最佳拟合椭圆的方向。同样的，在缩放后的窗口和先前窗口都会应用 meanshift 算法。该过程将持续到精确度满足要求。



camshift face image

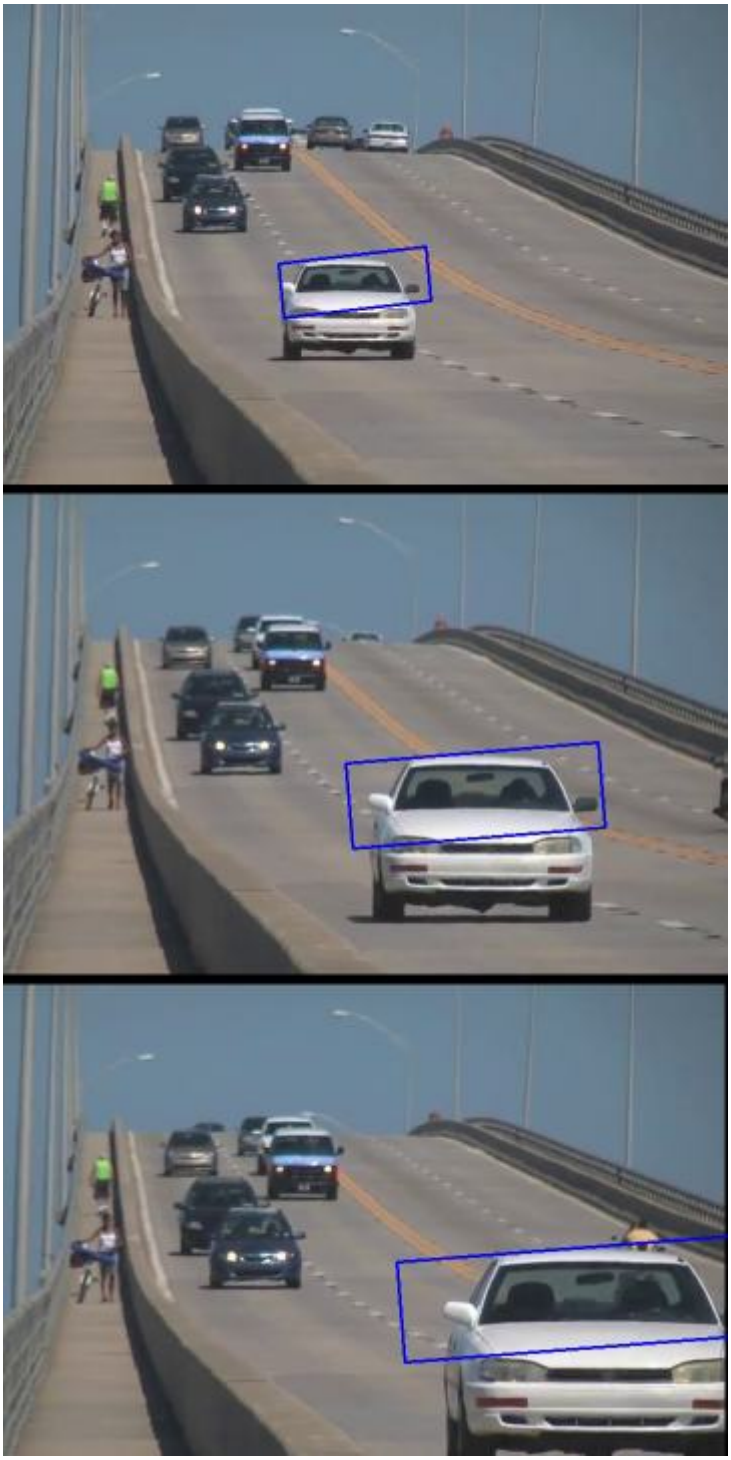
在 OpenCV 里使用 Camshift

它大部分与 meanshift 相同，但是其返回值是一个旋转的矩阵(这是我们得出的结果)和 box 参数(用于在下一次迭代中搜索窗口传递)

代码如下：

```
import numpy as np
import cv2 as cv

cap = cv.VideoCapture('slow.flv')

# 获取视频的第一帧
ret,frame = cap.read()

# 设置窗口的初始位置
r,h,c,w = 250,90,400,125  # 简单地硬编码值
track_window = (c,r,w,h)

# 设置 ROI(图像范围)以进行跟踪
roi = frame[r:r+h, c:c+w]
hsv_roi =  cv.cvtColor(roi, cv.COLOR_BGR2HSV)
mask = cv.inRange(hsv_roi, np.array((0., 60.,32.)), np.array((180.,255.,255.)))
roi_hist = cv.calcHist([hsv_roi],[0],mask,[180],[0,180])
cv.normalize(roi_hist,roi_hist,0,255,cv.NORM_MINMAX)

# 设置结束标志，10 次迭代或至少 1 次移动
term_crit = ( cv.TERM_CRITERIA_EPS | cv.TERM_CRITERIA_COUNT, 10, 1 )

while(1):
    ret ,frame = cap.read()

    if ret == True:
        hsv = cv.cvtColor(frame, cv.COLOR_BGR2HSV)
        dst = cv.calcBackProject([hsv],[0],roi_hist,[0,180],1)

        # 运行 Camshift 用以获取新的位置
        ret, track_window = cv.CamShift(dst, track_window, term_crit)

        # 绘制到新图像中
        pts = cv.boxPoints(ret)
        pts = np.int0(pts)
        img2 = cv.polylines(frame,[pts],True, 255,2)
        cv.imshow('img2',img2)

        k = cv.waitKey(60) & 0xff
        if k == 27:
            break
        else:
            cv.imwrite(chr(k)+".jpg",img2)
    else:
        break
cv.destroyAllWindows()
cap.release()
```

我使用的视频中的三个帧如下：



camshift result image

其他资源

1. 关于Camshift的法语维基百科页面。(上面的两幅动图出自于此)
2. Bradski, G.R.的 “Real time face and object tracking as a component of a perceptual user interface “论文

练习

1. OpenCV 附带了一个关于 camshift 交互式演示的 Python 示例。使用它，拆解它，理解它。

光流

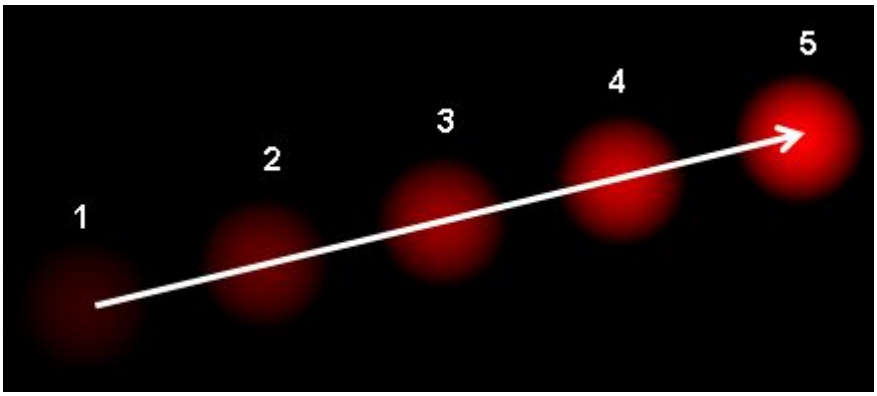
目标

在本章中

- 我们将理解光流的概念并且使用使用 Lucas-Kanade 方法估计它。
- 我们将使用像是 **cv.calcOpticalFlowPyrLK()** 的函数来跟踪视频中的特征点。

光流

光流是由于对象或者相机的移动引起的两个连续帧之间的时变图像的运动模式。这是一个二维的矢量场，其中，每一个矢量都是一个位移矢量用以显示从第一帧到第二帧的点的移动(位移)。思考下面的这张图片(图片提供：维基百科的光流词条)



optical flow basic image

这张图片展示了一个小球连续 5 帧的运动轨迹。箭头所展示的便是位移矢量。

光流在很多领域都有着很多的应用，如：

- 3D 重建
- 视频压缩
- 视频防抖
- ...

光流的概念基于以下假设：

- 对象的像素强度在连续帧之间不变化
- 相邻像素具有相似的运动

思考第一帧的一个像素$$I(x,y,t)$$(注意我们在这里添加了维度与时间概念。在之前我们只处理图像，所以不需要考虑时间)。这个像素将在$$dt$$时间后的下一帧移动$$(dx,dy)$$的距离。因此在那些像素点不会变化且亮度也不发生改变之后，我们可以说：

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

$$ 然后采用泰勒级数右近似，删除常数项并同时除以$$dt$$便最终得到了下面这个方程：

$$ {\notag} f_x u + f_y v + f_t = 0 \;

$$ 其中：

$$ {\notag} f_x = \frac{\partial f}{\partial x} \; ; \; f_y = \frac{\partial f}{\partial y}\ u = \frac{dx}{dt} \; ; \; v = \frac{dy}{dt}

$$ 上面的方程便是光流方程了。其中，我们可以找到$$f_x$$和$$f_y$$，它们是图像的梯度。同样，$$f_t$$是时间的梯度。但是$$(u,v)$$我们并不知道。我们不能带着两个未知变量来求解单个方程定解。所以人们寻找到了几种方案来解决这个问题，其中一种便是 Lucas-Kanade 方法。

Lucas-Kanade 方法

我们在之前提到过光流基于“ 相邻像素具有相似的运动 ”这个假设。Lucas-Kanade 方法将在像素点周围建立一个 3x3 邻域像素系统。因为假设 2，所以这九个点有着相同的运动。我们便可以在这九个点中寻找到$$(f_x, f_y, f_t)$$。所以我们的问题现在就变成了如何求解这九个方程组成的方程组，其中所求的两个变量是超定的。所以更好的解决方案则是利用最小二乘法拟合。下面这两个方程便是用以解决两个未知数问题的最终的解决方案。

$$ {\notag} \begin{bmatrix} u \ v \end{bmatrix} = \begin{bmatrix} \sum*{i}{f{xi}}^2 & \sum{i}{f{x_i} f{yi} } \ \sum{i}{f{x_i} f{yi}} & \sum{i}{f{y_i}}^2 \end{bmatrix}^{-1} \begin{bmatrix} - \sum{i}{f{x_i} f{ti}} \ - \sum{i}{f{y_i} f{t_i}} \end{bmatrix}

$$ (利用 Harris 角检测器来检查逆矩阵的相似性。这表明了角落是更好的跟踪点)

所以从使用者的角度来看，这个想法是很简单的，我们给出一些用以跟踪的点，我们接收那些点的光流向量。但是这又有一些问题。到目前为止，我们只不过是在处理一些小动作，所以当出现大动作时便会失败。我们将使用图像金字塔来解决这个问题。当我们沿金字塔向上时，小的动作被移除，大的动作则变成小动作。因此通过在金字塔最高层应用 Lucas-Kanade 方法，我们得到了小范围的光流。

在 OpenCV 里使用 Lucas-Kanade 光流算法

OpenCV 将这些功能都集成在了一个函数中， **cv.calcOpticalFlowPyrLK()**。这里，我们创建了一个用以在视频中跟踪某些点的简单程序。为了决定特征点，我们使用**cv.goodFeaturesToTrack()**函数。获取第一帧，并在其中检测 Shi-Tomasi 角点，然后我们使用 Lucas-Kanade 光流算法对于这些点进行迭代跟踪。对于函数 **cv.calcOpticalFlowPyrLK()**，我们将前一帧，之前的特征点和下一帧传入函数。它将返回下一组特征点以及状态向量，如果相应的特征点被发现，状态向量的每个元素被设置为 1，否则，被置为 0。我们将返回的这些点作为下一次迭代中所传递的参数。参照下面的代码：

```
import numpy as np
import cv2 as cv

cap = cv.VideoCapture('slow.flv')

# ShiTomasi 角点检测的参数
feature_params = dict( maxCorners = 100,
                       qualityLevel = 0.3,
                       minDistance = 7,
                       blockSize = 7 )

# Lucas-Kanade 光流算法的参数
lk_params = dict( winSize  = (15,15),
                  maxLevel = 2,
                  criteria = (cv.TERM_CRITERIA_EPS | cv.TERM_CRITERIA_COUNT, 10, 0.03))

# 创建一组随机颜色数
color = np.random.randint(0,255,(100,3))

# 取第一帧并寻找角点
ret, old_frame = cap.read()
old_gray = cv.cvtColor(old_frame, cv.COLOR_BGR2GRAY)
p0 = cv.goodFeaturesToTrack(old_gray, mask = None, **feature_params)

# 创建绘制轨迹用的遮罩图层
mask = np.zeros_like(old_frame)

while(1):
    ret,frame = cap.read()
    frame_gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)

    # 计算光流
    p1, st, err = cv.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None, **lk_params)

    # 选取最佳始末点
    good_new = p1[st==1]
    good_old = p0[st==1]

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

    cv.imshow('frame',img)
    k = cv.waitKey(30) & 0xff
    if k == 27:
        break

    # 更新选取帧与特征点
    old_gray = frame_gray.copy()
    p0 = good_new.reshape(-1,1,2)

cv.destroyAllWindows()
cap.release()
```

opticalflow lk image

(这个代码并不会检查下一组选取点是否正确，因此即使图像中任意特征点消失，光流也有可能寻找到可能看起来接近的点作为特征点。所以实际上对于稳定跟踪，需要在特定间隔后重新检查角点。OpenCV 里提供了这样的一个样例，它可以每 5 帧重新寻找特征点，而且还会对光流特征点进行反复检查，以便选择最优特征点。查看 samples/python/lk_track.py)

在 OpenCV 里计算稠密光流

Lucas-Kanade 方法是求稀疏光流的一种重要方法(在我们的例子中，使用 Shi-Tomasi 算法检测到角点)。而 OpenCV 提供了另一种算法用以计算稠密光流。这个方法将计算一帧中所有点的光流。这个方法基于 Gunner Farneback 算法，该算法在 Gunner Farneback 于 2003 年的所著的“基于多项式展开的双帧运动估计“论文中做了解释。

下面的例子将展示如何利用上面的算法寻找稠密光流。我们得到一个带有光流向量的双通道矩阵，$$(u,v)$$。我们将寻找其大小与方向。各种颜色代码用以获得更好的视觉效果。方向对应于图像的色相值。而大小则对应明度位面。代码如下：

```
import cv2 as cv
import numpy as np

cap = cv.VideoCapture("vtest.avi")

ret, frame1 = cap.read()
prvs = cv.cvtColor(frame1,cv.COLOR_BGR2GRAY)
hsv = np.zeros_like(frame1)
hsv[...,1] = 255

while(1):
    ret, frame2 = cap.read()
    next = cv.cvtColor(frame2,cv.COLOR_BGR2GRAY)

    flow = cv.calcOpticalFlowFarneback(prvs,next, None, 0.5, 3, 15, 3, 5, 1.2, 0)

    mag, ang = cv.cartToPolar(flow[...,0], flow[...,1])
    hsv[...,0] = ang*180/np.pi/2
    hsv[...,2] = cv.normalize(mag,None,0,255,cv.NORM_MINMAX)
    bgr = cv.cvtColor(hsv,cv.COLOR_HSV2BGR)

    cv.imshow('frame2',bgr)
    k = cv.waitKey(30) & 0xff
    if k == 27:
        break
    elif k == ord('s'):
        cv.imwrite('opticalfb.png',frame2)
        cv.imwrite('opticalhsv.png',bgr)
    prvs = next

cap.release()
cv.destroyAllWindows()
```

结果如下图：

optical fb image

OpenCV 附带有一个关于稠密光流的更加高级的样例。请看文件 samples/python/opt_flow.py。

其他资源

练习

1. 查看 samples/python/lk_track.py 文件的代码并尝试着理解它。
2. 查看 samples/python/opt_flow.py 的代码并尝试着理解它。

背景减法

目标

在本章中，

- 我们将熟悉在 OpenCV 里几种可行的背景减除方法

基础

背景减法是许多基于计算机视觉应用的主要预处理操作。举个例子，想象一下一个固定在柜台的摄像机需要计算进来与出去房间的人的数量的情况抑或是提取有关车辆等信息的交通摄像头等等。在所有这些情况下，首先你需要做的便是单独提取出人亦或是交通工具。从技术上来讲，你需要从静态的背景中提取出移动的前景。

如果你只有背景图像，像是没有访客的房间图像，没有车辆的道路图像等等，进行背景减法将是一件容易的事。只需要从背景图像中减去新图像。你便能得到前景对象了。但是，在大多数情况下，你可能并没有这样的图片，所以你需要从我们拥有的图像中提取背景。当车辆有阴影时情况会变得更加复杂。由于阴影也会移动，因此简单的减法也会将其标记为前景。这使得背景减除变得复杂。

为此我们引入了几种算法。OpenCV 已经实现了三种很容易使用的算法。我们将逐一看到它们。

BackgroundSubtractorMOG

这是一个基于高斯混合背景/前景分割的算法。这个算法在 P. KadewTraKuPong 和 R. Bowden 于 2001 年的论文“一种改进的自适应背景混合模型，用于带阴影检测的实时跟踪”中被引出。这个算法使用 K 个高斯模型进行混合用以表征背景中各个像素点(K = 3 到 5)。每个高斯模型的权值代表着当前场景下各种颜色所占时间的比例。而可能是背景的颜色则停留时间更长且更具静态性。

当我们编写程序的时候，我们需要使用**cv.createBackgroundSubtractorMOG()**函数创建背景对象。这个函数有一些可选参数，像是用于构建背景模型的帧数，混合的高斯模型个数，阈值等等。将它们全部设定为默认值。然后在视频循环中，使用backgroundsubtractor.apply()函数获取前景蒙版。

下面是一个简单的例子：

```
import numpy as np
import cv2 as cv

cap = cv.VideoCapture('vtest.avi')

fgbg = cv.bgsegm.createBackgroundSubtractorMOG()

while(1):
    ret, frame = cap.read()

    fgmask = fgbg.apply(frame)

    cv.imshow('frame',fgmask)
    k = cv.waitKey(30) & 0xff
    if k == 27:
        break

cap.release()
cv.destroyAllWindows()
```

(所有结果都在最后给出以供比较)

BackgroundSubtractorMOG2

这也是一个基于高斯混合背景/前景分割的算法。这个算法基于 Z.Zivkovic 于 2004 年发布的“改进的自适应高斯混合模型用于背景减法”和“2006 年发布的“用于背景减法任务的每个图像像素的有效自适应密度估计”两篇论文。该算法的一个重要的特征便是它为每一个像素点均选择适当数量的高斯分布。(记住，在最后一种情况，我们在整个算法中利用了 K 高斯模型)。它应对不同场景中光照变化等因素将提供更好的适应性。

在上一种情况中，我们创建了一个背景减法器的对象。在这里，你可以选择是否检测阴影。如果 detectShadows = True (默认是 True)，它将检测并遮罩阴影部分，但这样会降低计算速度。阴影将会被标记为灰色。

```
import numpy as np
import cv2 as cv

cap = cv.VideoCapture('vtest.avi')

fgbg = cv.createBackgroundSubtractorMOG2()

while(1):
    ret, frame = cap.read()

    fgmask = fgbg.apply(frame)

    cv.imshow('frame',fgmask)
    k = cv.waitKey(30) & 0xff
    if k == 27:
        break

cap.release()
cv.destroyAllWindows()
```

(结果在最下方给出)

BackgroundSubtractorGMG

这个算法结合了静态背景图像估计和每个像素的贝叶斯分割。此算法于 2012 年由 Andrew B. Godbehere，Akihiro Matsukawa 和 Ken Goldberg 发布的“可变照明条件下的人类访问者视觉跟踪以响应音频艺术装置“论文中被提出并介绍。根据该论文，该系统于 2011 年 3 月 31 日至 7 月 31 日在加利福尼亚州旧金山的当代犹太博物馆举办了一场名为“我们在那里吗？”的成功互动音频艺术装置。

它使用起始的一些帧(默认为 120 个)进行背景建模。它采用概率前景分割算法，使用贝叶斯推断识别可能的前景对象。它采用概率前景分割算法也就是使用贝叶斯推理识别可能的前景对象。这种估计是自适应的。在适应变化的光照中，新的观察者比旧观察者拥有更高的权重。进行几个形态过滤上操作如闭和开操作用以移除不必要的噪点。在前几帧你将得到一个完全黑色的窗口。

如果能够利用开运算移除噪声结果会更好。

```
import numpy as np
import cv2 as cv

cap = cv.VideoCapture('vtest.avi')

kernel = cv.getStructuringElement(cv.MORPH_ELLIPSE,(3,3))
fgbg = cv.bgsegm.createBackgroundSubtractorGMG()

while(1):
    ret, frame = cap.read()

    fgmask = fgbg.apply(frame)
    fgmask = cv.morphologyEx(fgmask, cv.MORPH_OPEN, kernel)

    cv.imshow('frame',fgmask)
    k = cv.waitKey(30) & 0xff
    if k == 27:
        break

cap.release()
cv.destroyAllWindows()
```

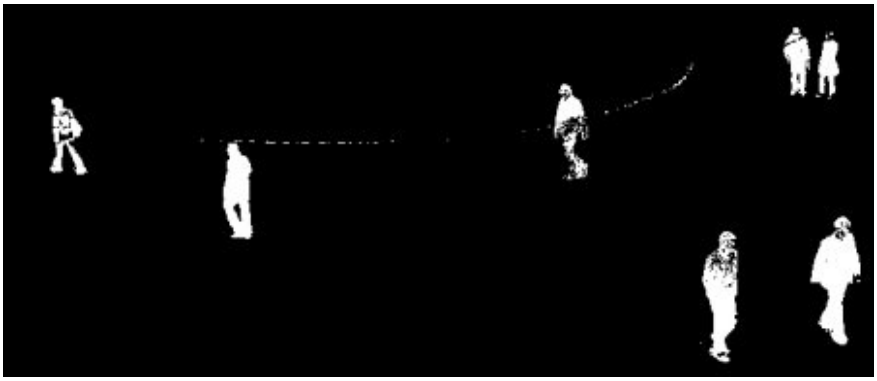
结果

原始帧

下面这张图片展示了一个视频的第 200 帧

resframe image

使用 BackgroundSubtractorMOG 后



resmog image

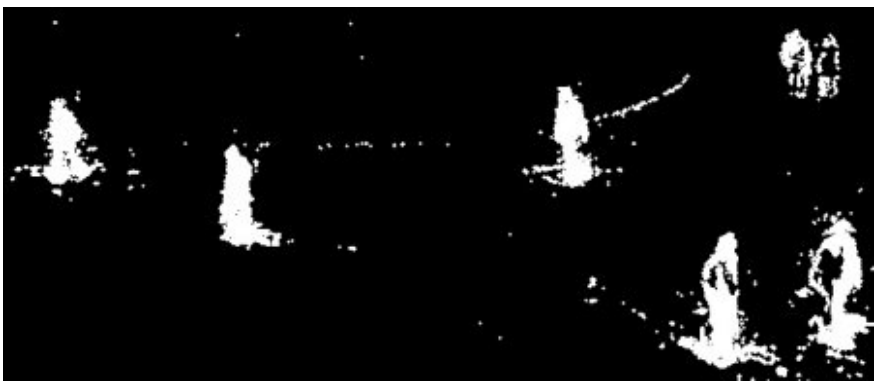
使用 BackgroundSubtractorMOG2 后

灰色区域显示阴影区域

resmog2 image

使用 BackgroundSubtractorGMG 后

通过数学形态学中的开运算移除噪声



resgmg image

其他资源

练习

相机校准

目标

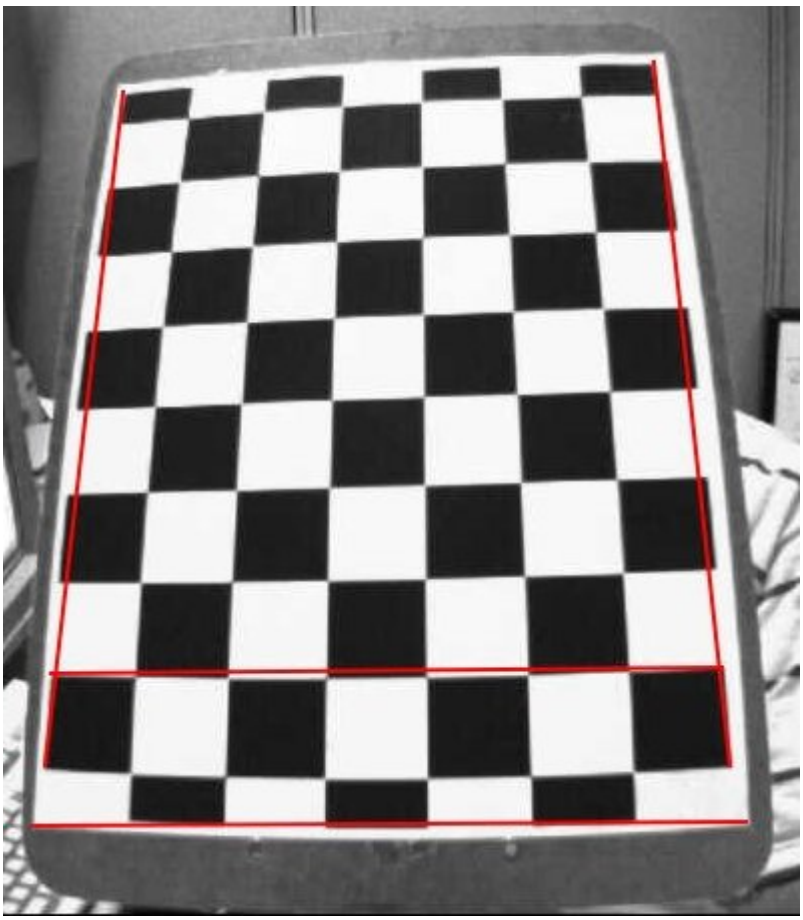
在这一部分中，我们将了解关于：

- 相机畸变的类型
- 如何找出相机的固有属性和可变属性
- 如何利用这些属性校准相机图像

基础

一些针孔摄像机会有严重的图像畸变的问题。其中径向畸变和切向畸变是两种主要的畸变现象。

径向畸变使得直线变得弯曲。切向畸变使得离图像中心点越远的点看上去更远。举个例子，如下图像展示了一个两条边界被红线标记的棋盘。但是，你可以看到棋盘的边缘不仅不是直线，而且与红线偏差很大。所有预期的直线都弯曲了。访问畸变(光学)来获取更多信息。



calib radial image

径向畸变可表示为如下公式：

$$ {\notag} x*{distorted} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \ y*{distorted} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)

$$ 相似的，发生切向畸变是因为摄像镜头未完全平行于图像平面。所以，图像中的某些区域可能看起来比预期的更近。切向畸变可表示如下公式：

$$ {\notag} x*{distorted} = x + [2p_1xy + p_2(r^2+2x^2)] \ y*{distorted} = y + [p_1(r^2+ 2y^2)+ 2p_2xy]

$$ 简而言之，我们需要找到上面的五个参数，其被称为畸变系数，由下式给出：

$$ {\notag} Distortion \; coefficients=(k_1 \hspace{10pt} k_2 \hspace{10pt} p_1 \hspace{10pt} p_2 \hspace{10pt} k_3)

$$ 除此之外，我们还需要一些其他信息，像是相机的固有属性和可变属性。固有属性是每个相机的特有属性。其中包括像是焦距$$(f_x,f_y)$$和光心$$(c_x, c_y)$$。焦距和光心可以被用于创建相机矩阵，用于消除相机镜头特有属性造成的畸变。每个相机的相机矩阵都是独一无二的，所以一旦我们计算出来，便可以在同一相机所拍摄的其他图像上重复使用。其表示为 3x3 的矩阵：

$$ {\notag} camera \; matrix = \left [\begin{matrix} f_x & 0 & c_x \ 0 & f_y & c_y \ 0 & 0 & 1 \end{matrix} \right]

$$ 外部参数对应于旋转和平移矢量，其将 3D 点的坐标平移到坐标系。

对于立体应用方面，这些畸变现象首先需要解决。为了寻找到这些参数，我们必须提供一些被明确定义的图像(比如棋盘)。我们可以寻找到一些我们早就知道相对位置的点(比如棋盘的方格的角点)。我们也知道这些点在真实世界中的坐标以及图像中的坐标，由此我们便可以解出畸变系数。如果想要获取更好的结果，我们需要至少 10 个测试图像。

代码

正如上面所言，我们至少需要 10 个图像用以相机校准。OpenCV 提供了一些棋盘的图片(参见 samples/data/left01.jpg – left14.jpg)，所以我们将利用这些图像。思考一个棋盘的图像。相机校准所需要的重要输入数据便是 3D 真实世界点的集合以及在图像中这些点所对应的 2D 坐标。我们可以轻易从这些图像中寻找到 2D 图像点。(这些图像点是棋盘中两个黑色块相交的位置)。

那真实世界中的 3D 点又如何呢？这些图像于同一相机静止拍摄，其中的棋盘放置于不同的位置与方向。所以我们需要知道$$(X,Y,Z)$$的值。但是为了简单起见，我们可以说棋盘在 XY 平面保持静止，(所以 Z 恒等于 0)而相机是移动物件。这个考量帮助我们可以仅找出 X，Y 的值。现在对于 X，Y 的值，我们可以简单地传递像是(0,0), (1,0), (2,0), ... 之类的点用于表示点的位置。在此之下，我们得到的结果将是棋盘方块相对的大小。但是如果我们知道棋盘方块的大小，(大约 30mm)，我们便可以传递像(0,0), (30,0), (60,0), ...这样的值。因此，我们的到的结果也是 mm 为单位的。(在这种情况下，我们不知道方块尺寸，因为我们没有拍摄这些图像，所以我们将方块尺寸作为参数传入)。

3D 点被称作**对象点**，2D 点被称作**图像点**

标定

所以为了寻找到棋盘上的图案，我们可以使用一个函数，**cv.findChessboardCorners()**。我们同样需要传递我们寻找的图案类型，像是 8x8 网格，5x5 网格之类的。在这个例子中，我们使用 7x6 的网格(一般的棋盘规格是 8x8 网格和 7x7 的内角)。它返回角点和阈值，如果成功找到所有角点，则返回 True。这些角落将按顺序放置（从左到右，从上到下）

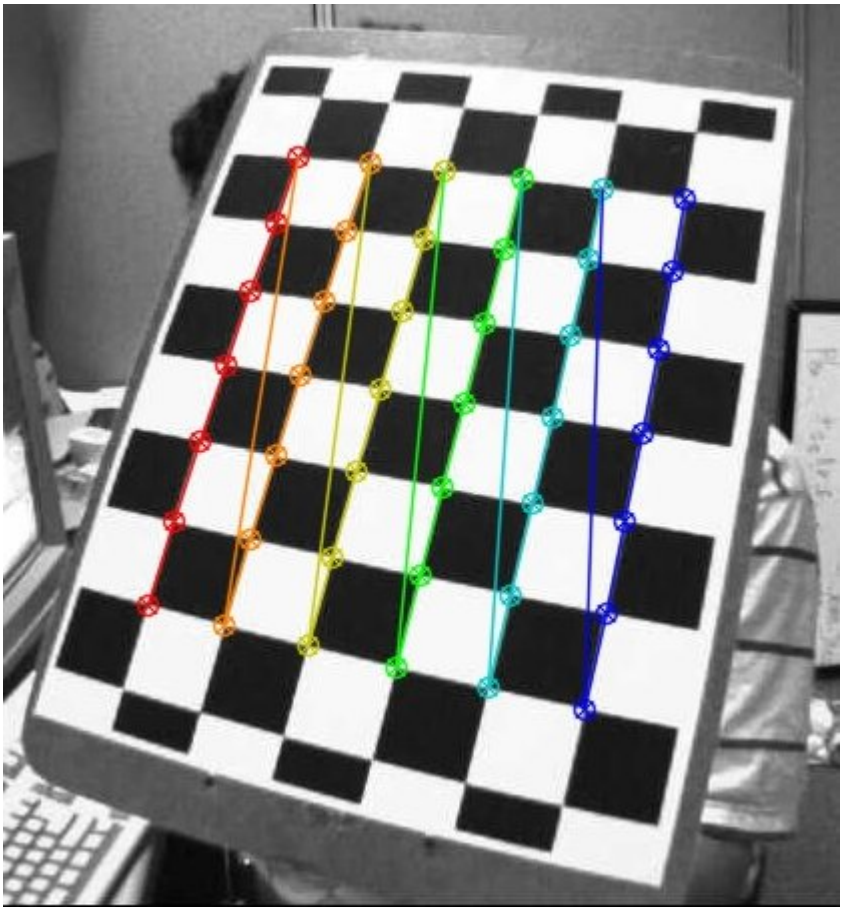
参考

- 这个函数可能不能够找到所有图像中需求的图案。所以，有一个很好的选项便是编写代码，以便它启动相机并检查每一帧是否需要获取图案。一旦图案确定了，便寻找角点并存入一个列表当中。同样的，在阅读下一帧之前提供一些间隔，以便我们可以在不同的方向上调整我们的棋盘。持续此过程，直至获得所需要的优良图案。甚至在我们在这里给出的示例当中，我们也无法确定在我们给出的 14 张图像中有多少是优良的。由此，我们必须读入所有的图像并仅选取好的图像。
- 除了棋盘，我们也可使用圆形网格。在此之下，我们必须使用**cv.cornerSubPix()**函数用以寻找图案。使用圆形网格使得相机需更少的图像便可足以校准。

一旦我们找到了角点，我们可以使用**cv.cornerSubPix()**函数来提高它们的准确性。我们同样可以使用 **cv.drawChessboardCorners()**函数绘制图案。所有步骤均包含在下面代码当中：

```
import numpy as np
import cv2 as cv
import glob

# 终止标准
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# 准备对象点，如 (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)

# 用于存储所有图像对象点与图像点的矩阵
objpoints = [] # 在真实世界中的 3d 点
imgpoints = [] # 在图像平面中的 2d 点

images = glob.glob('*.jpg')

for fname in images:
    img = cv.imread(fname)
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    # 找到棋盘上所有的角点
    ret, corners = cv.findChessboardCorners(gray, (7,6), None)

    # 如果找到了，便添加对象点和图像点(在细化后)
    if ret == True:
        objpoints.append(objp)

        corners2 = cv.cornerSubPix(gray,corners, (11,11), (-1,-1), criteria)
        imgpoints.append(corners)

        # 绘制角点
        cv.drawChessboardCorners(img, (7,6), corners2, ret)
        cv.imshow('img', img)
        cv.waitKey(500)

cv.destroyAllWindows()
```

绘制完成的图像如下所示：



calib pattern image

校准

现在我们拥有了对象点与图像点，我们便可以准备开始校准了。我们使用函数返回相机矩阵，畸变系数，旋转和平移向量等等。

```
ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints, gray.shape[::-
```

矫正

现在，我们可以选取图像并矫正它们了。OpenCV 包含有两种方案来实现这件事。然而首先，我们需要使用 **cv.getOptimalNewCameraMatrix()**函数根据自由缩放系数细化相机矩阵。如果缩放参数 alpha = 0，这个函数将返回最小不必要像素的校正图像。所以它甚至可能会移除图像角落的一些像素。如果 alpha = 1，则所有像素都会保留一些额外的黑色图像。此函数还返回图像 ROI，可用于裁剪结果。

所以我们需要一张新的图像。(在此选用 left12.jpg，这是本章的第一张图片)

```
img = cv.imread('left12.jpg')
h,  w = img.shape[:2]
newcameramtx, roi = cv.getOptimalNewCameraMatrix(mtx, dist, (w,h), 1, (w,h))
```

1. 使用cv.undistort()函数

这是最简单的方法。只需要调用这个函数并使用使用上面获得的 ROI 来裁剪结果。

```
# 矫正
dst = cv.undistort(img, mtx, dist, None, newcameramtx)

# 裁切图像
x, y, w, h = roi
dst = dst[y:y+h, x:x+w]
cv.imwrite('calibresult.png', dst)
```

2. 使用重映射

这个方法稍微困难一点，首先，找到一个从畸变的图像到矫正过的图像的映射函数。然后使用重映射函数。

```
# 矫正
mapx, mapy = cv.initUndistortRectifyMap(mtx, dist, None, newcameramtx, (w,h), 5)
dst = cv.remap(img, mapx, mapy, cv.INTER_LINEAR)

# 裁切图像
x, y, w, h = roi
dst = dst[y:y+h, x:x+w]
cv.imwrite('calibresult.png', dst)
```

尽管如此，这两种方法都给出了相同的结果。如下：

calib result image

可以看到现在所有边都是直的。

现在你可以利用 NumPy 中的写入函数(np.savez, np.savetxt 等)用来保存你的相机矩阵和畸变系数用以备用。

重投影误差

重投影误差可以很好的估计我们计算出的参数的精确程度。重投影误差越接近于零，我们计算出的参数便越准确。给出固有，畸变，旋转和平移矩阵，我们首先必须使用**cv.projectPoints()**函数将对象点转换为图像点。然后，我们便可以计算出我们变换结果和发现角点的算法之间的绝对范数。为了找到平均误差，我们将计算所有图像计算误差的算术平均值。

```
mean_error = 0
for i in xrange(len(objpoints)):
    imgpoints2, _ = cv.projectPoints(objpoints[i], rvecs[i], tvecs[i], mtx, dist)
    error = cv.norm(imgpoints[i], imgpoints2, cv.NORM_L2)/len(imgpoints2)
    mean_error += error

print( "total error: {}".format(mean_error/len(objpoints)) )
```

其他资源

练习

1. 尝试用圆形网格进行相机校准

姿势估计

目标

在这一部分，

- 我们会了解到如何利用 calib3d 模块在图像中实现 3D 效果。

基础

这将仅是很小一部分。在上一章节(相机校准)，你已经找到了相机矩阵，畸变系数等等参数。给出一个图案图像，我们便可以利用上面的信息用于计算其姿势，或者物体在空间中位于何处，比如如何旋转，如何移动等等问题。对于一个平面物体，我们可以假定 Z = 0，这样，问题现在便转化为了如何放置摄像机才能查看到我们的图案图像。所以如果我们知道物体在空间中的位置，我们便可以绘制一些 2D 图像用以模拟 3D 效果。让我们看一下如何做到这件事情。

我们的问题是，我们想在我们棋盘的第一个角上绘制 3D 坐标系(x, y, z 坐标系)，其中 X 轴是蓝色，Y 轴是绿色，Z 轴是红色。所以从效果上讲，Z 轴应该感觉像是与棋盘垂直的。

首先，让我们读取从上一章节(相机校准)存储的结果中读取出相机矩阵与畸变参数。

```
import numpy as np
import cv2 as cv
import glob

# 读取事先存好的数据
with np.load('B.npz') as X:
    mtx, dist, _, _ = [X[i] for i in ('mtx','dist','rvecs','tvecs')]
```

现在让我们创建一个函数 `draw`，用以利用棋盘角点(利用 **cv.findChessboardCorners()**函数)和**坐标轴点**绘制 3D 坐标系。

```
def draw(img, corners, imgpts):
    corner = tuple(corners[0].ravel())
    img = cv.line(img, corner, tuple(imgpts[0].ravel()), (255,0,0), 5)
    img = cv.line(img, corner, tuple(imgpts[1].ravel()), (0,255,0), 5)
    img = cv.line(img, corner, tuple(imgpts[2].ravel()), (0,0,255), 5)
    return img
```

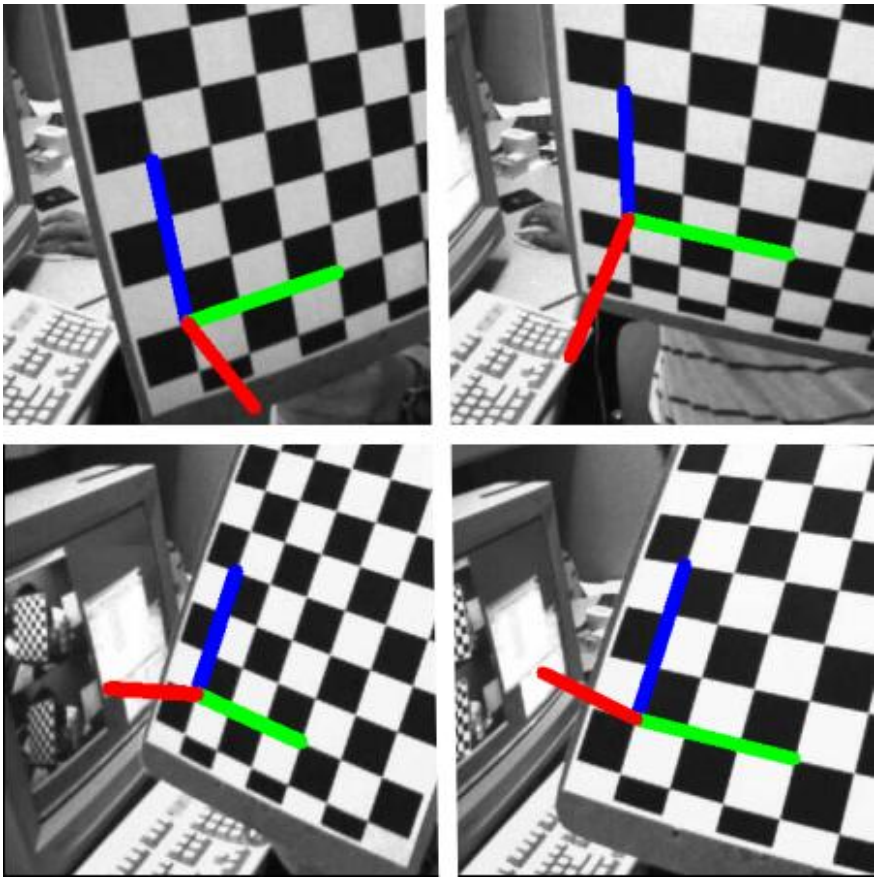
然后正如前面情况那样，我们创建了终止条件，物体点(棋盘上的 3D 角点)和坐标轴点。坐标轴点是 3D 空间中用于绘制轴的点。我们绘制长度为 3 的轴(单位是国际象棋的方形尺寸，我们会根据这个尺寸校准)。所以我们的 X 轴是从(0,0,0)到(3,0,0)绘制的，所以对于 Y 轴。对于 Z 轴，它是从(0,0,0)到(0,0，-3)绘制的。负数表示其接近摄像机。

```
criteria = (cv.TERM_CRITERIA_EPS + cv.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)
```

现在，像往常一样，我们读取一张图片。搜索 7x6 网格。如果我们找到了，我们用子角像素优化一下它，然后计算旋转和平移，我们使用**cv.solvePnPRansac()**函数。一旦我们计算完那些旋转矩阵后，我们便用它们来将我们的**坐标轴点**投影到平面图像上。简而言之，我们寻找到平面图象上的点对应 3D 空间里的(3,0,0), (0,3,0), (0,0,3)。一旦我们找到后，我们便可以从第一个角到每个我们找到的坐标轴点之间利用 `draw()` 函数连线。搞定！！！

```
for fname in glob.glob('left*.jpg'):
    img = cv.imread(fname)
    gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
    ret, corners = cv.findChessboardCorners(gray, (7,6),None)

    if ret == True:
        corners2 = cv.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)

        # 找到旋转和平移向量
        ret,rvecs, tvecs = cv.solvePnP(objp, corners2, mtx, dist)

        # 投射 3D 点到平面图像上
        imgpts, jac = cv.projectPoints(axis, rvecs, tvecs, mtx, dist)

        img = draw(img,corners2,imgpts)
        cv.imshow('img',img)
        k = cv.waitKey(0) & 0xFF
        if k == ord('s'):
            cv.imwrite(fname[:6]+'.png', img)

cv.destroyAllWindows()
```

请看下面的一些结果。注意，每个轴长 3 个方格：



pose 1 image

渲染立方体

如果你想要绘制一个立方体，你需要根据如下步骤改进 `draw()` 函数和坐标轴点。

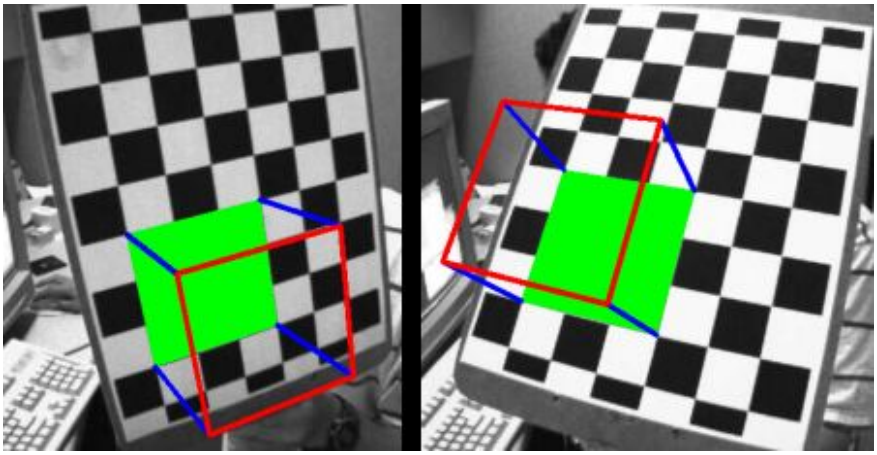
改进 `draw()` 函数：

```
def draw(img, corners, imgpts):
    imgpts = np.int32(imgpts).reshape(-1,2)

    # 将底面绘制为绿色
    img = cv.drawContours(img, [imgpts[:4]],-1,(0,255,0),-3)

    # 将支柱绘制为蓝色
    for i,j in zip(range(4),range(4,8)):
        img = cv.line(img, tuple(imgpts[i]), tuple(imgpts[j]),(255),3)

    # 将顶面绘制为红色
    img = cv.drawContours(img, [imgpts[4:]],-1,(0,0,255),3)

    return img
```

改进坐标轴点。它们是 3D 空间中立方体的 8 个角：

```
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] ])
```

然后看上去就像下面这样：



pose 2 image

如果你对图形渲染，增强现实等感兴趣，你可以使用 OpenGL 来渲染更加复杂的图像。

其他资源

练习

线性几何

目标

在本章中，

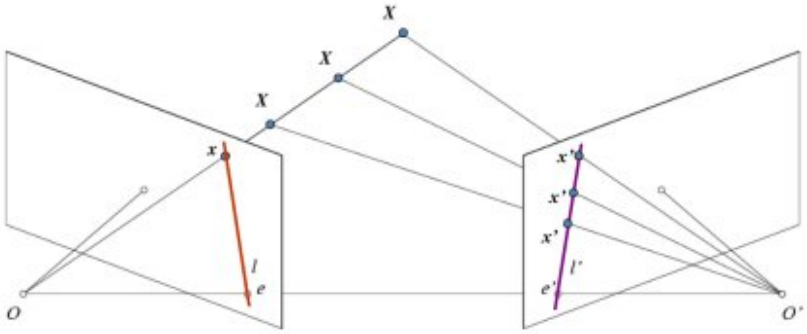
- 我们将了解基础的多视图几何
- 我们将看到什么是极点，极线，线性约束等等

基本概念

当我们使用针孔摄像机拍照的时候，我们将失去一个重要的信息，即图像的深度。或者是每一个点由于 3D 到 2D 转换导致的到摄像机远近距离问题。所以这有着一个严重的问题就是我们能否使用这些摄像机找到深度信息。这个问题的答案便是使用不止一个摄像机。我们的眼睛也以相似的方式工作，我们使用两个摄像机(两只眼睛)，这种方案分支被称作立体视觉。所以让我们看看 OpenCV 在这个领域提供了什么信息。

(《学习 OpenCV》(Gary Bradsky 著)有着关于这个领域很多的知识)

在我们投入到图像深度问题前，我们首先先要了解一些关于多视图几何的基本概念。在这一部分，我们将讨论线性几何。请看下面这张图片，它展示了一个两个摄像头拍摄相同场景的基本配置。



epipolar image

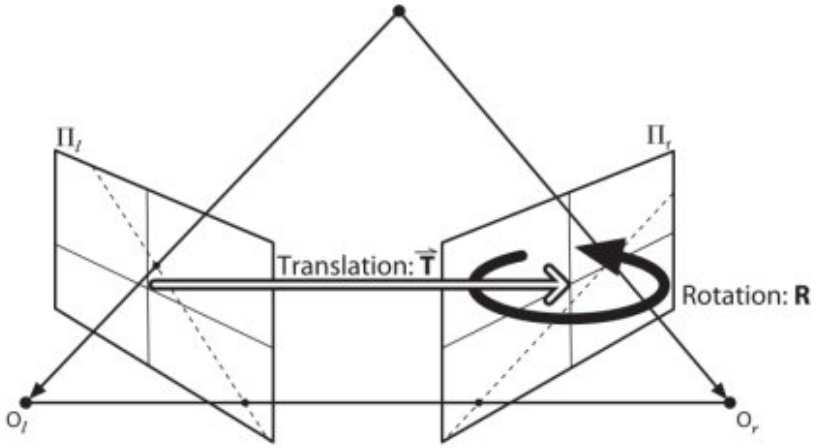
如果我们只使用左边的摄像机，我们将不能找到 3D 点在图像中对应的$$x$$坐标因为在$$OX$$直线上的每一个点都会投影到图像平面中的同一个点。但是我们也会考虑右侧的图像。现在，$$OX$$直线上的不同的点会投影到右平面上不同的点($$x'$$)了。所以通过这两个图像，我们便可以三角测量出正确的 3D 点。这便是整个的思路。

所有在$$OX$$上不同的点在右平面上形成了一条直线(直线$$l'$$)。我们将其称为点$$x$$对应的**极线**。这意味着在右侧图像寻找点$$x$$仅需要搜寻极线上的点。这个点可能在这条线的任意位置(试想一下，为了在图像中寻找到匹配点，你不需要在整张图像中寻找，而是仅在一条极线上寻找。这会大大提升运算效率和准确性)。这种方法被称作**线性约束**。类似地，所有点将在另一图像中具有其对应的极线。平面$$XOO'$$被称作**极平面**。

$$O$$和$$O'$$是相机投影中心。通过上面的方法，我们可以看到右侧摄像机的$$O'$$在左侧图像上的投影点，$$e$$。这个点被称作**极点**。这个极点是投影中心连线与图像平面的交点。类似的，$$e'$$是左摄像机的极点。在某些情况下，你将无法在图像中找到极点，它们可能置于图像外(这意味着，一台摄像机无法看到另一台)。

所有极线都会过极点。所以找到极点，我们便可以找到许多极线并找到他们的交叉点。

所以在这一部分，我们将着眼于寻找极线和极点。但是为了找到它们，我们还需要两个参数，**基础矩阵(F)**和**本征矩阵(E)**。本征矩阵包括平移和旋转的信息，用以描述第二个摄像机在全局坐标中相对于第一个摄像机的位置。观察下面的图像(图片来源：Gary Bradsky 所著《学习 OpenCV》)



essential matrix image

但我们更喜欢以像素坐标进行测量，不是吗？基础矩阵包含有同本质矩阵相同的信息，以及关于两个摄像头的本征信息，以便我们可以在像素坐标中关联两个摄像头(如果我们使用校正后的图像，并用焦距除以该点标准化，则$$F=E$$)。简而言之，基础矩阵 F，会将图像上的一个点映射为另一个图像的一条线(极线)。这是通过两个图像之间的匹配点计算出来的。最少需要 8 个这样的点用以寻找基础矩阵(同时使用 8 点算法)。越多点是越好的，这使得我们可以使用 RANSAC 获得更加可信的结果。

代码

首先，我们需要在两个图像之间找到尽可能多的匹配，以找到基础矩阵。为此，我们将 SIFT 描述符与基于 FLANN 的匹配器和比率测试结合使用。

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

img1 = cv.imread('myleft.jpg',0)  #查询图像 # 左图像
img2 = cv.imread('myright.jpg',0) #训练图像 # 右图像

sift = cv.SIFT()

# 使用 SIFT 获得特征点和描述符
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

# FLANN 参数
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks=50)

flann = cv.FlannBasedMatcher(index_params,search_params)
matches = flann.knnMatch(des1,des2,k=2)

good = []
pts1 = []
pts2 = []

# 根据 Lowe's 的论文进行比率测试
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)
```

现在我们有两张图片的最佳匹配列表。让我们找到基础矩阵。

```
pts1 = np.int32(pts1)
pts2 = np.int32(pts2)
F, mask = cv.findFundamentalMat(pts1,pts2,cv.FM_LMEDS)

# 我们只会选取内部点
pts1 = pts1[mask.ravel()==1]
pts2 = pts2[mask.ravel()==1]
```

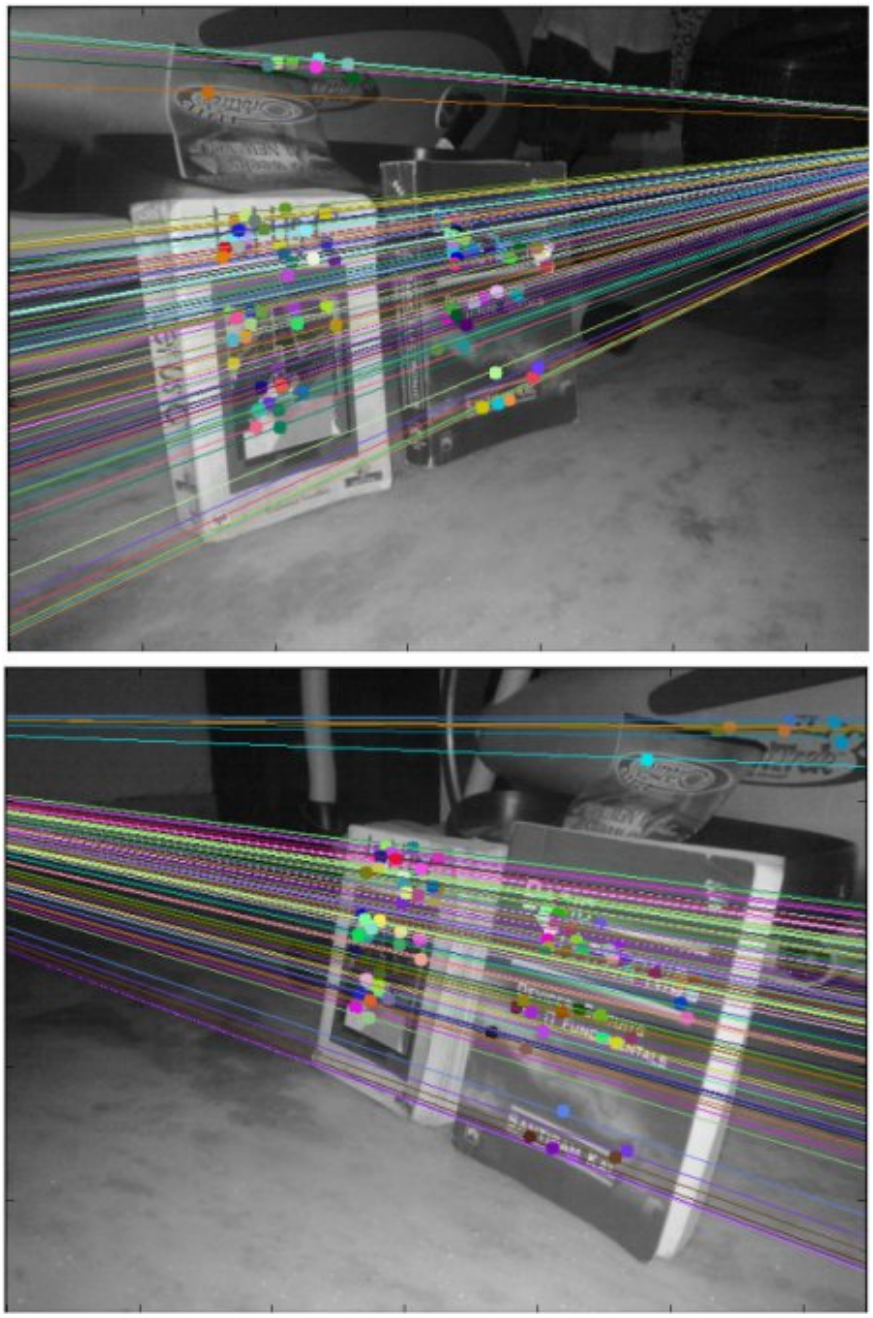
接下来我们寻找极线。极线对应的于第一图像上的点将被绘制到第二图像上。所以在这里提到正确图像很重要。我们得到了一系列线组成的数组。所以我们定义一个新的函数来在图象上绘制这些线条。

```
def drawlines(img1,img2,lines,pts1,pts2):
    ''' img1 - 为了在 img2 为点绘制极线的图像
        lines - 对应的极线 '''
    r,c = img1.shape
    img1 = cv.cvtColor(img1,cv.COLOR_GRAY2BGR)
    img2 = cv.cvtColor(img2,cv.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 = cv.line(img1, (x0,y0), (x1,y1), color,1)
        img1 = cv.circle(img1,tuple(pt1),5,color,-1)
        img2 = cv.circle(img2,tuple(pt2),5,color,-1)
    return img1,img2
```

现在我们在两个图像中找到了极线并绘制它们。

```
# 寻找到极线在右图像(第二图像)对应的点并
# 在左图像绘制连线
lines1 = cv.computeCorrespondEpilines(pts2.reshape(-1,1,2), 2,F)
lines1 = lines1.reshape(-1,3)
img5,img6 = drawlines(img1,img2,lines1,pts1,pts2)

# 寻找到极线在左图像(第一图像)对应的点并
# 在右图像绘制连线
lines2 = cv.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()
```

下面就是我们得到的结果：



epiresult image

你可以看到左边的图像所有的极线都汇聚到右侧图像外的一点。那个汇聚点便是极点。

为了获得更好的效果，应使用具有良好分辨率且包含许多非平面点的图像。

其他资源

练习

1. 一个重要的话题是相机的向前移动。然后，在固定点出现的 epilines 中，将在相同的位置看到 epipoles。查看这个讨论。

2. 基础矩阵估计对于匹配质量，异常值等等很敏感。当所有选定的匹配点位于同一平面上时，情况会变得更糟。查看这个讨论。

立体图像的深度图

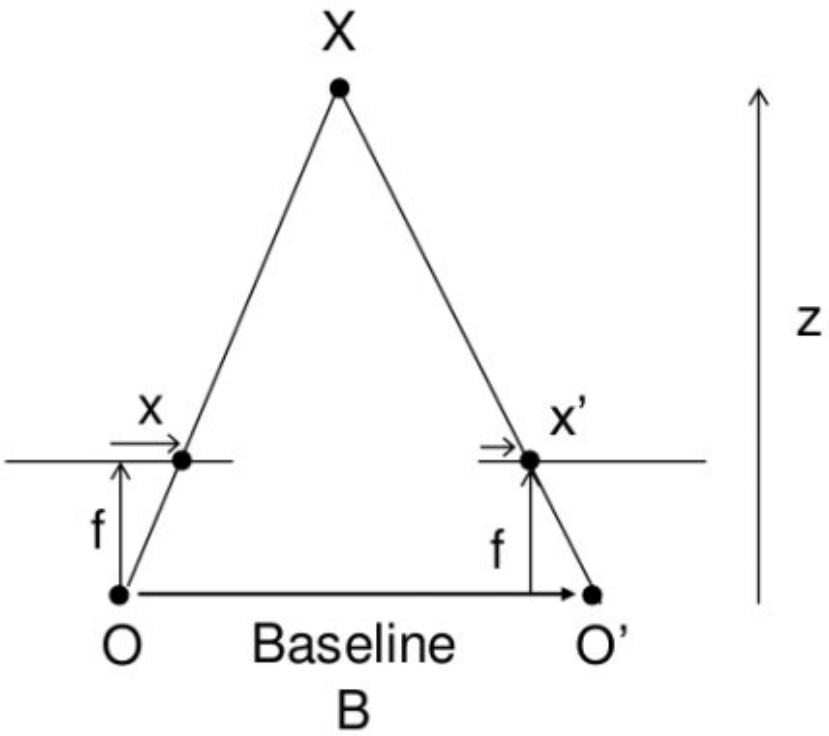
目标

在本章节中，

- 我们将学习到如何通过立体图像来创建深度图。

基础

在上一章节中，我们看到了一些基本概念像是线性约束和其他相关术语。我们同时也看到，如果我们有两个相同场景的图像，我们便可以直观的获取到其深度信息。下面便是一张图片和一些用以证明这种直观现象的数学公式(图片提供：(译者注：这里原文中就没写出处)



stereo depth image

上图包含等边三角形。写出其等效方程则如下：

$$ {\notag} disparity = x - x' = \frac{Bf}{Z}

$$

$$x$$和$$x'$$是图像平面上的点对应的 3D 场景点与相机投影中心之间的距离。$$B$$是两个摄像机之间的距离(这是我们知道的)，$$F$$是相机的焦距(早已知晓)。所以长话短说，上面的等式便是说一个场景点的深度是与相应图像点及其相机投影中心的距离差成反比的。所以通过这个信息，我们边可以推导出图像中所有像素的深度。

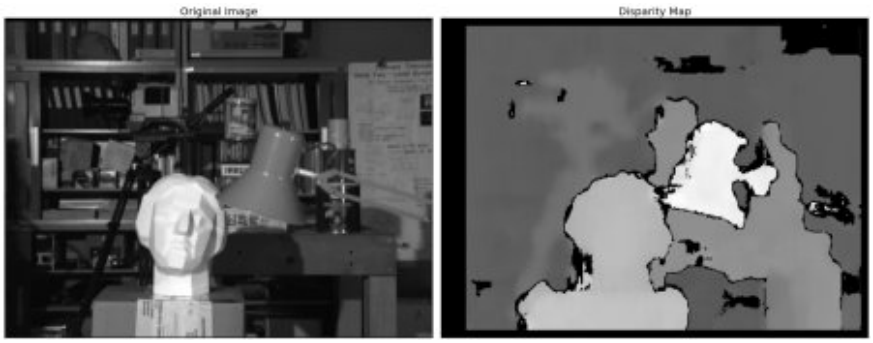
所以它将在两个图象之间进行匹配。我们已经看到过如何使用线性约束使这个操作更快更准确。一旦寻找到匹配点，我们便可以找到其视差。让我们看看我们如何使用 OpenCV 来实现它。

代码

下面的代码片段展示了创建视差图的简单过程。

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

imgL = cv.imread('tsukuba_l.png',0)
imgR = cv.imread('tsukuba_r.png',0)

stereo = cv.StereoBM_create(numDisparities=16, blockSize=15)
disparity = stereo.compute(imgL,imgR)
plt.imshow(disparity,'gray')
plt.show()
```

下面的图像包含有原图像(左)和视差图(右)。正如你看到的，结果被高度的噪声所污染。通过调整 numDisparities 和 blockSize 的值，你可以获得更好的结果。



disparity map image

这里有一些参数是你学习 StereoBM 时了解过的，您可能需要对参数进行微调以获得更好，更平滑的结果。

参数：

- texture_threshold: 过滤掉没有足够纹理以进行可靠匹配的区域。
- Speckle range and size: 基础块匹配器通常会在物体边界产生“斑点”，其中匹配窗口在一侧捕获前景而在另一侧捕获背景。在此场景中，似乎匹配器还在桌子上的投影纹理中发现了小的伪匹配。为了解决这些问题，我们使用由 speckle_size 和 speckle_range 参数控制的散斑滤波器对视差图像进行后处理。speckle_size 是像素数，低于该数量的视差斑点被视为“散斑”。speckle_range 控制值差异的接近程度，必须被视为同一个斑点的一部分。
- Number of disparities: 窗口滑过的像素数。这个参数越大，其可见深度就越大，但也需要更多的计算量。
- min_disparity: 从左侧 x 点向右进行搜索的偏移量。
- uniqueness_ratio: 另一个后处理滤波。如果最佳匹配视差不足够好于搜索范围中的每个其他视差，这些像素就会被过滤掉。如果 texture_threshold 和噪点滤波仍然使错误匹配通过。

- prefilter_size and prefilter_cap: 预处理滤波参数，用于标准化图像亮度与提升纹理用以准备块匹配。通常你不需要调整它们。

其他资源

- 旋转立体图像处理的维基百科页面

练习

1. OpenCV 实例中包含有一个关于生成视差图及其三维重建的例子。查看在 OpenCV-Python 例子中的 stereo_match.py 文件。

K-最近邻算法

理解 K-最近邻算法

对“kNN 是什么”有个基本了解

使用 kNN 进行手写识别

现在，让我们用 OpenCV 中的 kNN 算法进行数字识别（光学字符识别，简称 OCR ）。

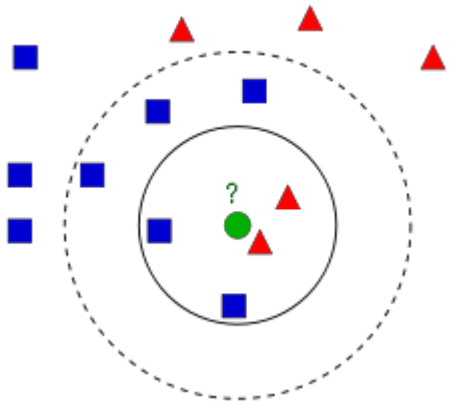
理解 K-最近邻算法

目标

在本章中，我们将理解 K-最近邻（kNN）算法的基本概念。

原理

kNN 是可用于监督学习的最简单的分类算法之一。其主要思路是在特征空间中搜索最接近测试数据的匹配项。我们可以用下图来表示它



图像

在上图中，有两个家族，蓝色方块和红色三角形。我们将每个家族称之为**类** 。他们的房子在我们称之为特征空间的城镇地图上显示出来。*（你可以把特征空间理解为一个投影了所有数据的空间。例如，想象一个二维坐标空间。每个元素有两个特征，x 和 y 坐标。你可以在你的二维坐标空间里表示这些元素，对不对？现在想象一下，如果有 3 个特征，你需要 3 维空间。如果有 N 个特征，你需要 N 维空间，对不对？这个 N 维空间就是它的特征空间。在我们的图像中，你可以把它当作一个有着两个特征的 2 维案例）*。

现在，一个新成员进入城镇并创建了一个新家，显示为绿色圆圈。他将被加入到 Blue/Red 家族中的一个。我们将该过程称为**分类**。那么，我们该怎么做？由于我们正在研究 kNN，那么让我们应用这个算法。

一种方法是检查谁是他最近的邻居，从图像来看，很明显是红色三角形家族。所以他也被加入到红色三角形家族。此方法简称为**最近邻**，因为分类仅取决于最近邻居。

但是，这有个问题。红色三角形可能是最近的，但如果有很多蓝色方块也和他很近，那会怎么样呢？那么，蓝色方块在该地区有着比红色三角形更强的影响力，因此，仅仅检查最近的一个邻居是不够的。取而代之，我们检查 k 个最近的邻居，然后，谁占大多数，新人就属于那个家族。在我们的图像中，让我们取 k=3，即 3 个最近的邻居。他有 2 个红色邻居和 1 个蓝色邻居（有两个蓝色邻居的距离是一样的，但因为 k=3，所有我们只取他们中的一个），因此，他再一次应该被加到红色家族。但是，如果我们取 k=7 又会如何？会发现，他有 5 个蓝色邻居和 2 个红色邻居。很好！！现在，他应该被加入到蓝色家族。我们可以看到，这些变化都取决

于 k 的值。更有趣的是，如果 k=4 会怎么样？他将有 2 个红色邻居和 2 个蓝色邻居。一个平局！！！因此，k 最好取一个奇数。因为分类取决于 k 个最近的邻居，所有这种方法被称为 **K-最近邻算法**

再想想，在 kNN 中，我们考虑了 k 个邻居，这没问题，但是，每个邻居我们都给了相同的权重，这样对吗？这样公平吗？举例来说，就举 k=4 的例子吧，我们说这是一个平局，但是，仔细想想，2 个红色邻居比 2 个蓝色邻居离他更近，所以他更有资格加入到红色家族。那么我们怎么在数学上表示这一点呢？我们根据他们与新来者的距离给每个家庭一些权重。对于那些靠近他的家庭来说，获得更高的权重，而那些距离较远的家庭则获得较低的权重。然后，我们分别计算每个家族的总权重。谁的总权重高，新来者就加入那个家族。这就是**修改后的 kNN**

那么，讲到这里，什么是重要的呢？

- 你需要小镇上所有房子的信息，对不对？因为，我们必须检查新人到所有现有房屋的距离，以找到最近的邻居。如果有很多房屋和家族，那么将占用很多内存，计算上也将更费时间。
- 在训练和预处理上几乎不花时间

现在，让我们看看 OpenCV 中的 kNN。

OpenCV 中的 kNN

我们将在这里举一个简单的例子，有两个家族（类），就像上面一样。然后在下一章中，我们将举一个更好的例子。

所以在这里，我们将红色家族标记为**Class-0**（用 0 表示），将蓝色家族标记为 **Class-1**（用 1 表示）。我们创建了 25 个家庭（25 个训练数据），并将他们标记为 Class-0 和 Class-1。我们在 Numpy 的随机数生成器的帮助下完成所有这些工作。

然后我们在 Matplotlib 的帮助下绘制它。 红色家族显示为红色三角形，蓝色家族显示为蓝色方块。

```
import cv2 as cv
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()
```

你将获得与我们的第一张图片类似的图片。 由于你使用的是随机数生成器，因此每次运行代码时都会获得不同的图片。

接下来初始化 kNN 算法，并把训练数据和响应传递给它，以训练 kNN（构建一个搜索树）。

然后我们将引入一个新成员，并在 OpenCV 的 kNN 的帮助下将他归类到某个家族。在讲 kNN 之前，我们需要了解一些关于测试数据（新成员的数据）的知识。我们的数据应该是一个浮点数数组，其大小为 **测试数据的个数 × 特征数**。然后，我们找到了新成员的最近的邻居们。我们可以指定最近的邻居的数量。返回如下内容：

1.新成员的标签取决于我们之前所说的 kNN 理论。如果你想用最近邻算法，只要指定 k=1 就可以了，这里的 k 是邻居的个数。

2.k 个邻居的标签

3.从新成员到每个邻居的距离

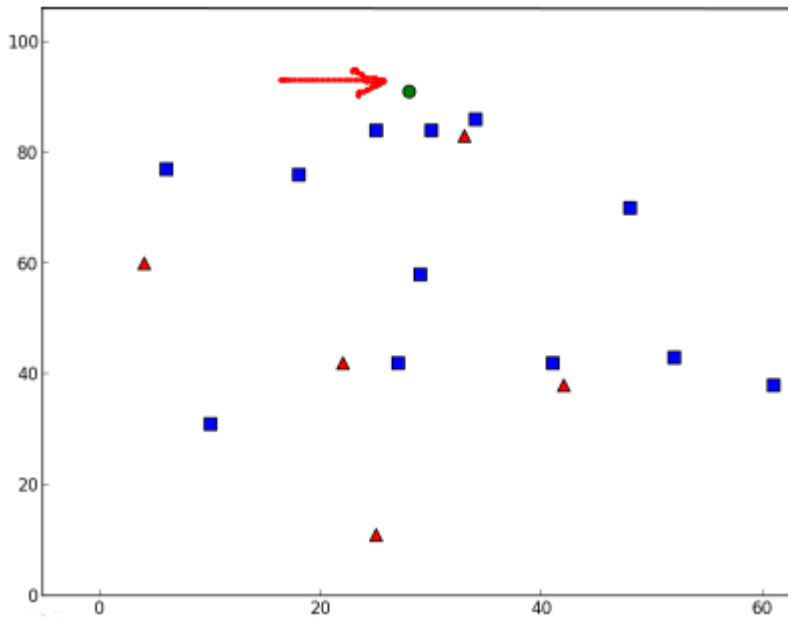
接下来，让我们看看它是如何工作的。新成员被标记为绿色。

```
newcomer = np.random.randint(0,100,(1,2)).astype(np.float32)
plt.scatter(newcomer[:,0],newcomer[:,1],80,'g','o')

knn = cv.ml.KNearest_create()
knn.train(trainData, cv.ml.ROW_SAMPLE, responses)
ret, results, neighbours ,dist = knn.findNearest(newcomer, 3)

print( "result:  {}\n".format(results) )
print( "neighbours:  {}\n".format(neighbours) )
print( "distance:  {}\n".format(dist) )

plt.show()
```

我得到了如下结果：

```
result:  [[ 1.]]
neighbours:  [[ 1.  1.  1.]]
distance:  [[ 53.  58.  61.]]
```

这说明我们的新成员有 3 个邻居，全部来自蓝色家族。因此，他被贴上了蓝色家族的标签。下图更为明显：



图像

如果你的测试数据很多，你可以使用数组。对应的结果也是个数组。

```
# 10 new comers
newcomers = np.random.randint(0,100,(10,2)).astype(np.float32)
ret, results,neighbours,dist = knn.findNearest(newcomer, 3)
# The results also will contain 10 labels.
```

其他资源

1.NPTEL notes on Pattern Recognition, Chapter 11

练习

使用 kNN 进行手写识别

目标

在本章

- 我们将使用我们在 kNN 上的知识来构建基本的 OCR 应用程序。
- 我们将尝试使用 OpenCV 自带的数字和字母数据。

手写数字的 OCR

我们的目标是构建一个可以读取手写数字的应用程序。为此，我们需要一些训练数据和测试数据。OpenCV 自带了一张图片 digits.png（在文件夹 opencv/samples/data/ 中），它有 5000 个手写数字（每个数字 500 个）。每个数字是 20x20 图像。所以我们的第一步是将这个图片分成 5000 个不同的数字。对于每个数字，我们将其展平为一个 400 像素的行。这是我们的特征集，即所有像素的强度值。这是我们可以创建的最简单的特征集。我们使用每个数字的前 250 个样本作为训练数据，接下来 250 个样本作为测试数据。所以让我们先做好准备吧。

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

img = cv.imread('digits.png')
gray = cv.cvtColor(img,cv.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 = cv.ml.KNearest_create()
knn.train(train, cv.ml.ROW_SAMPLE, train_labels)
ret,result,neighbours,dist = knn.findNearest(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 )
```

到这里，我们的基础 OCR 应用已经准备好了。这个特定的例子给了我 91%的准确率。提高准确性的一个选择是为训练添加更多数据，尤其是错误的数据。因此，为了避免每次都要启动程序找这些训练数据，我最好保存他们，以便下次我直接从文件中读取这些数据并开始分类。你可以借助 np.savetxt、np.savez、np.load 等 Numpy 函数来完成。请查看它们的文档以获取更多详细信息。

```
# save the data
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，这需要大约 4.4 MB 的内存。由于我们使用强度值（uint8 数据）作为特征，最好先将数据转换为 np.uint8 然后保存。在这种情况下，它只需要 1.1 MB。然后在加载时，你可以转换回 float32。

英文字母的 OCR

接下来我们将对英文字母执行相同操作，但数据和特征集略有变化。在这里，我们用 OpenCV 自带的数据文件，opencv/samples/cpp/（译者注：应为 opencv/samples/data） 目录里的 letter-recognition.data，来代替图像数据。 如果你打开它，你会看到 20000 行，乍一看，有点像垃圾数据。实际上，在每一行中，第一列是一个字母，它也是我们的标签，接下来的 16 个数字是它的不同特征。这些特征可从UCI 机器学习库获得，你可以在此页面中找到这些特征的详细信息。

有 20000 个样本可用，因此我们首先将 10000 个数据作为训练样本，剩余 10000 个作为测试样本。我们应该将字母转为 ascii 字符，因为我们无法直接处理字母。

```
import cv2 as cv
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 = cv.ml.KNearest_create()
knn.train(trainData, cv.ml.ROW_SAMPLE, responses)
ret, result, neighbours, dist = knn.findNearest(testData, k=5)

correct = np.count_nonzero(result == labels)
accuracy = correct*100.0/10000
print( accuracy )
```

此程序给我的准确率为 93.22%。 同样，如果您想提高准确率，可以在每个级别迭代添加错误数据。

额外资源

练习

支持向量机（Support Vector Machine, SVM）

理解 SVM

对“SVM 是什么”有个基本了解

使用 SVM 进行手写数据识别

学习使用 OpenCV 中的 SVM 功能函数

理解 SVM

目标

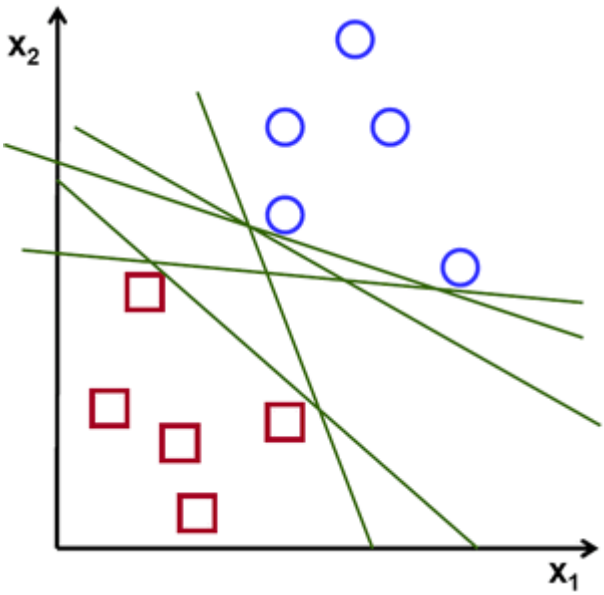
在本章

- 我们将对 SVM 有个直观理解

原理

线性可分离数据

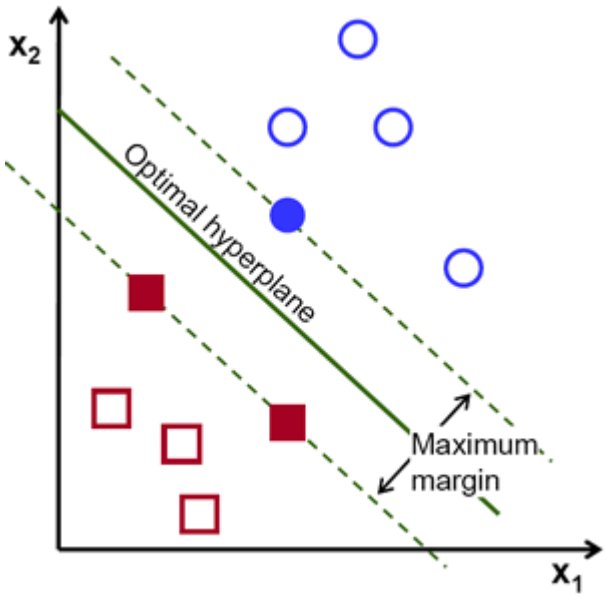
考虑下面的图像有两种类型的数据，红色和蓝色。在 kNN 中，对于一个测试数据，我们通常要计算它与所有训练样本的距离，并选择最短距离的那个。计算与所有样本的距离花费了大量时间，而且存储所有的训练样本要花费大量的内存。但是，对于图片中所给出的数据，我们真的需要这么多吗?



图像

换个思路。我们可以找到一条直线 **f(x)=ax$_1$ + bx$_2$ + c** 将数据分成两个区域。当我们得到一个新的测试数据 ***X*** 时，只需用 **f(x)** 代替它。如果 **f(*X*) > 0** ，则属于蓝色组，否则属于红色组。我们可以将此直线称为**决策边界**。这非常简单，并且用的内存很少。像这种能够用一条直线（或更高维度的超平面）分割成两部分的数据被称为**线性可分离的**。

在上图中，你可以看到很多这样的直线。我们将选哪一条？非常直观地说，该直线应该尽可能远离所有点。为什么呢？因为输入数据中可能存在噪声数据。这些噪声数据不应影响分类准确性。因此，选择最远的直线将提供更强的抗噪声能力。因此，SVM 所做的是找到与训练样本具有最大最小距离的直线（或超平面）。请参见下图中穿过中心的粗线。



图像

因此，为了找到决策边界，你需要训练数据。你需要训练所有数据吗？ 不。仅仅那些接近对方的数据就够了。在我们的图像中，它们是一个蓝色填充圆和两个红色填充正方形。我们可以称它们为**支持向量**，通过它们的线称为**支持平面**。它们足以找到我们的决策边界。我们无需为所有数据的量太大而担心。它有助于减少数据。

发生了什么事呢？首先，有两个超平面被发现最能代表数据。例如，蓝色数据由 $w^Tx + b_0 > 1$ 表示，而红色数据由 $w^Tx + b_0 < -1$ 表示，其中 **w** 是权重向量（ $w = [w_1,w_2,...,w_n]$ ）， **x** 是特征向量（ $x = [x_1,x_2,...,x_n]$ ）， b_0 是偏差。权重向量决定决策边界的方向，而偏差点决定其位置。现在，决策边界被定义为这些超平面的中间，因此表示为 $w^Tx + b_0 = 0$ 。从支持向量到决策边界的最小距离由 $distance_{support\ vectors} = 1/||w||$ 给出。间距是这个距离的两倍，我们需要最大化这个间距。即我们需要最小化一个新函数 $L(w,b_0)$ ，其中一些约束可以表示如下：

$$\min_{w,b_0} L(w, b_0) = \frac{1}{2}||w||^2 \text{ subject to } t_i(w^T x + b_0) \geq 1 \ \forall i$$

其中 ti 是每个类的标签， $t_i \in [-1,1]$ 。

非线性可分离数据

考虑到一些数据并不能用直线分成两部分。例如，一维数据，其中'X'在-3 和 +3，'O'在-1 和+1。显然，它不是线性可分的。但是有一些方法可以解决这类问题。 如果我们可以用函数 $f(x) = x^2$ 映射这个数据集，我们得到'X'为 9，'O'为 1，它们是线性可分的。

另外，我们可以将这个一维数据转换为二维数据。我们可以使用 $f(x) = (x, x^2)$ 函数来映射这些数据。然后'X'变为(-3,9)和(3,9)，而'O'变为(-1,1)和(1,1)。这也是线性可分的。简而言之，低维空间中的非线性可分数据在高维空间中变得线性可分离的可能性更大。

通常，这种方法是可行的，将 d 维空间中的点映射到某个 D 维空间 ($D > d$) 以检查线性可分离性的可能性。有一个想法有助于通过在低维输入（特征）空间中执行计算来计算高维（核）空间中的点积。我们可以通过以下示例来说明。

考虑二维空间中的两个点， **p = (p_1, p_2)** 和 **q = (q_1, q_2)** 。设 **Φ** 是一个映射函数，它将二维的点映射到三维空间，如下所示：

$$\phi(p) = (p_1^2, p_2^2, \sqrt{2}p_1 p_2)\phi(q) = (q_1^2, q_2^2, \sqrt{2}q_1 q_2)$$

让我们定义一个核函数 ***K(p,q)*** ，它在两点之间做一个点积，如下所示：

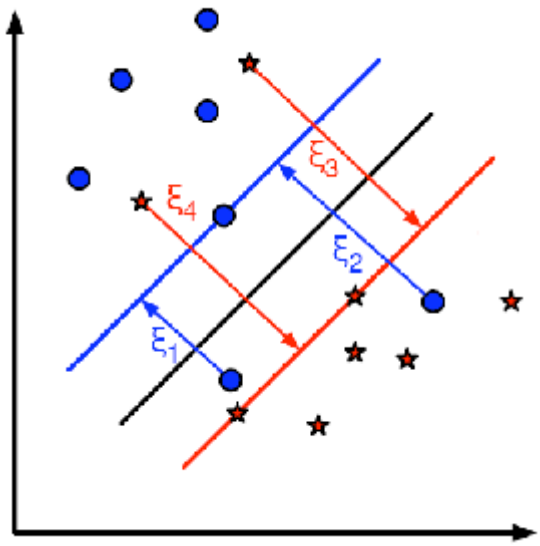
$$\begin{aligned} K(p,q) = \phi(p).\phi(q) &= \phi(p)^T\phi(q) \\ &= (p_1^2, p_2^2, \sqrt{2}p_1 p_2).(q_1^2, q_2^2, \sqrt{2}q_1 q_2) \\ &= p_1 q_1 + p_2 q_2 + 2p_1 q_1 p_2 q_2 \\ &= (p_1 q_1 + p_2 q_2)^2 \\ \phi(p).\phi(q) &= (p.q)^2 \end{aligned}$$

这意味着，在二维空间中使用平方点积可以实现三维空间中的点积。这可以应用于更高维度的空间。因此，我们可以从较低维度本身计算更高维度的特征。一旦我们映射它们，我们就会获得一个更高维的空间。

除了所有这些概念之外，还存在分类错误的问题。因此，仅仅找到具有最大间距的决策边界是不够的。我们还需要考虑分类错误的问题。有时，可能会找到间距较小的决策边界，但分类错误减少。无论如何，我们需要修改我们的模型，使其找到具有最大间距的决策边界，但分类错误较少。最小化标准修改为：

$$min\ ||w||^2 + C(distance\ of\ misclassified\ samples\ to\ their\ correct\ regions)$$

下图显示了这个概念。对于训练数据的每个样本，定义新参数 **ξi** 。它是从相应的训练样本到正确决策区域的距离。对于那些没有被错误分类的数据，它们会落在相应的支持平面上，因此它们的距离为零。



图像

所以新的优化问题是：

$$\min_{w,b_0} L(w,b_0) = ||w||^2 + C\sum_i \xi_i \text{ subject to } y_i(w^T x_i + b_0) \geq 1 - \xi_i \text{ and } \xi_i \geq 0\ \forall i$$

如何选择参数 C？很明显，这个问题的答案取决于训练数据的分布方式。虽然没有一般性答案，但考虑以下规则很有用：

- 较大的 C 值给出的解决方案具有较少的错误分类，但间距较小。当错误分类的代价高昂时，可以考虑此方案。因为优化的目标是最小化参数，所以较少的分类错误也是被允许的。
- 小的 C 值给出的解决方案具有更大的间距和更多的分类错误。在这种情况下，最小化并不会太关心求和项，因此它更侧重于寻找具有大间距的超平面。

额外资源

1. NPTEL 关于统计模式识别的说明，第 25-29 章

练习

使用 SVM 进行手写数据识别

目标

在本章

- 我们将再次学习手写数据 OCR，但是，使用 SVM 而不是 kNN。

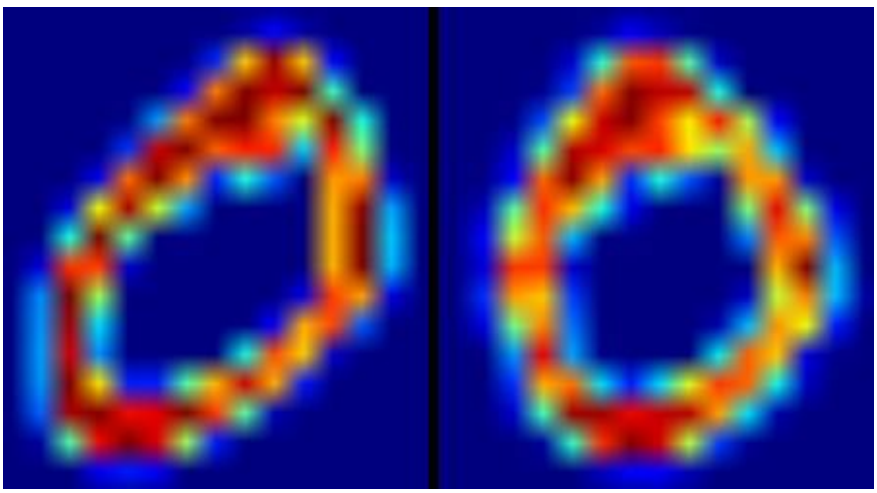
手写数字的 OCR

在 kNN 中，我们直接使用像素强度作为特征向量。这次我们将使用方向梯度直方图（HOG）作为特征向量。

在找 HOG 之前，我们使用其二阶矩来校正图像。所以我们首先定义一个函数 **deskew()**，它取一个数字图像并对其进行校正。下面是 deskew()函数：

```
def deskew(img):
    m = cv.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 = cv.warpAffine(img,M,(SZ, SZ),flags=affine_flags)
    return img
```

下图展示了应用于零图像的上述矫正函数。左图是原始图像，右图是矫正后的图像。



图像

接下来，我们必须找到每个单元的 HOG 描述符。为此，我们在 X 和 Y 方向找到每个单元的 Sobel 导数。然后，在每个像素处找到它们的大小和梯度方向。该梯度被量化为 0~16 间的整数。将此图像分为四个子方块。对于每个子方块，计算使用大小加权的方向的直方图（16 bins）。因此，每个子方块都会给你一个有 16 个值的向量。四个这样的向量（四个子方块）一起给出了有着 64 个值的特征向量。这是我们用来训练数据的特征向量。

```
def hog(img):
    gx = cv.Sobel(img, cv.CV_32F, 1, 0)
    gy = cv.Sobel(img, cv.CV_32F, 0, 1)
    mag, ang = cv.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_c
    hist = np.hstack(hists)     # hist is a 64 bit vector
    return hist
```

最后，与先前一样，我们首先将大数据集拆分为独立的单元。对每个数字，保留 250 个单元用于训练数据，剩余的 250 个数据被留下来用于测试。完整代码如下，你也可以从这里下载：

```
#!/usr/bin/env python

import cv2 as cv
import numpy as np

SZ=20
bin_n = 16 # Number of bins

affine_flags = cv.WARP_INVERSE_MAP|cv.INTER_LINEAR

def deskew(img):
    m = cv.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 = cv.warpAffine(img,M,(SZ, SZ),flags=affine_flags)
    return img

def hog(img):
    gx = cv.Sobel(img, cv.CV_32F, 1, 0)
    gy = cv.Sobel(img, cv.CV_32F, 0, 1)
    mag, ang = cv.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_c
    hist = np.hstack(hists)     # hist is a 64 bit vector
    return hist

img = cv.imread('digits.png',0)
if img is None:
    raise Exception("we need the digits.png image from samples/data here !")

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]

deskewed = [list(map(deskew,row)) for row in train_cells]
hogdata = [list(map(hog,row)) for row in deskewed]
trainData = np.float32(hogdata).reshape(-1,64)
responses = np.repeat(np.arange(10),250)[:,np.newaxis]

svm = cv.ml.SVM_create()
svm.setKernel(cv.ml.SVM_LINEAR)
svm.setType(cv.ml.SVM_C_SVC)
svm.setC(2.67)
svm.setGamma(5.383)

svm.train(trainData, cv.ml.ROW_SAMPLE, responses)
svm.save('svm_data.dat')

deskewed = [list(map(deskew,row)) for row in test_cells]
hogdata = [list(map(hog,row)) for row in deskewed]
testData = np.float32(hogdata).reshape(-1,bin_n*4)
result = svm.predict(testData)[1]

mask = result==responses
correct = np.count_nonzero(mask)
print(correct*100.0/result.size)
```

这种特殊技术给了我近 94%的准确率。你可以尝试为 SVM 的各种参数设置不同的值，以检查是否可以获得更高的精度。或者你也可以阅读该领域的技术论文并尝试实现它们。

额外资源

1. 方向梯度直方图视频

练习

1. OpenCV 示例里有个 digits.py，它对上述方法稍微做了改进，并获得了更好的效果。它还包含参考资料。阅读并理解它。

K-Means 聚类

理解 K-Means 聚类

阅读并对 K-Means 聚类有个直观的理解

OpenCV 中的 K-Means 聚类

现在让我们在 OpenCV 中尝试 K-Means 函数

理解 K-Means 聚类

目标

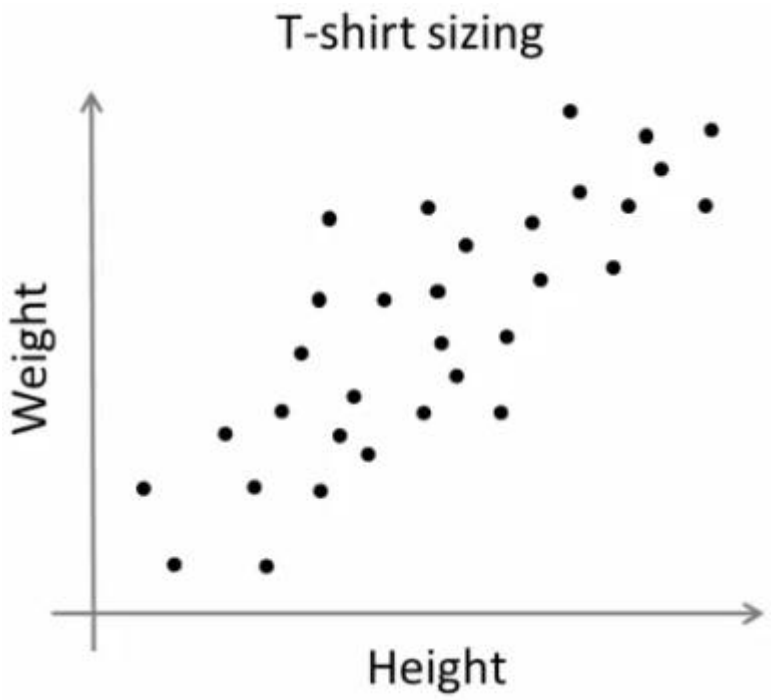
在本章中，我们将了解 K-Means 聚类的概念，以及它是如何工作的，等等。

原理

我们将用一个常用的例子来解释原理。

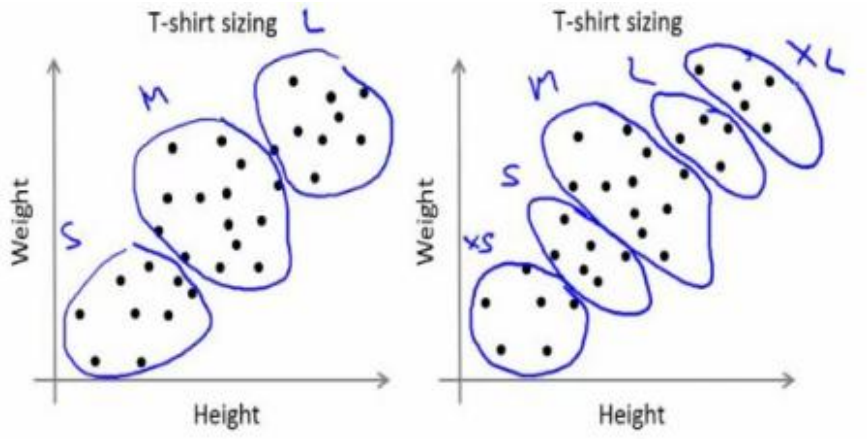
T 恤大小问题

举个例子，有一家公司将向市场推出一种新型 T 恤。显然，他们必须制造不同尺寸的衣服，以满足各种身材的人。为此，公司收集了一份人的身高和体重的数据，并将其绘制成图表，如下所示：



图像

公司不能制作所有尺寸的 T 恤。相反，他们将人们划分为小巧、中等和高大，并且仅制造这 3 种尺寸的衣服，以满足所有人的需求。将人群分为 3 组这种事情可以用 K-Means 聚类算法来完成，并且算法能够给我们找出最合适的 3 个尺寸，让所有人满意。如果不行，公司可以将人们分成更多的组，可能是 5 组，或者其它。查看下图：

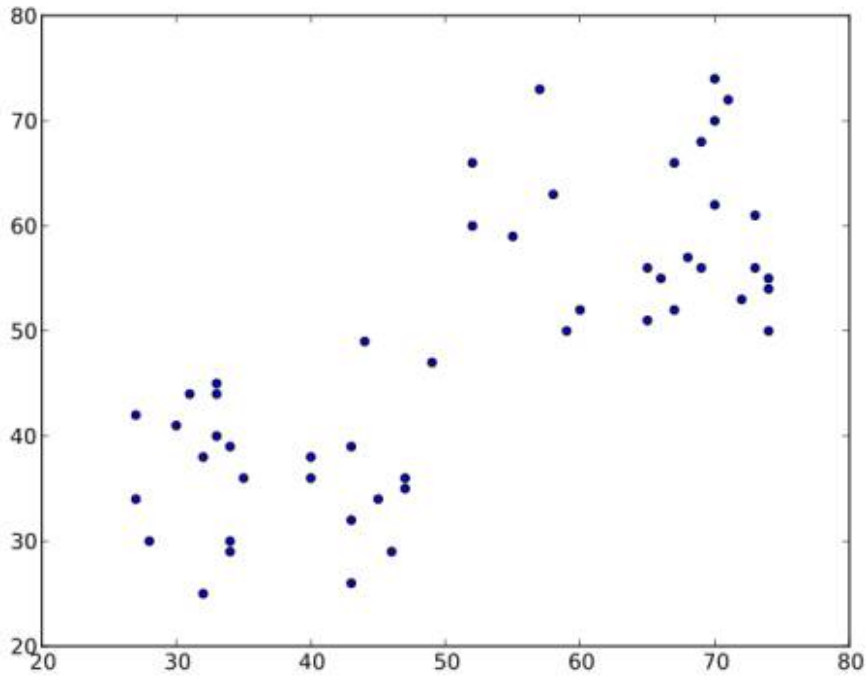


图像

如何工作的?

该算法是一个迭代过程，我们将在图像的帮助下逐步解释它。

考虑如下的一组数据（你可以将其视为 T 恤问题）。我们需要将这些数据分成两组。

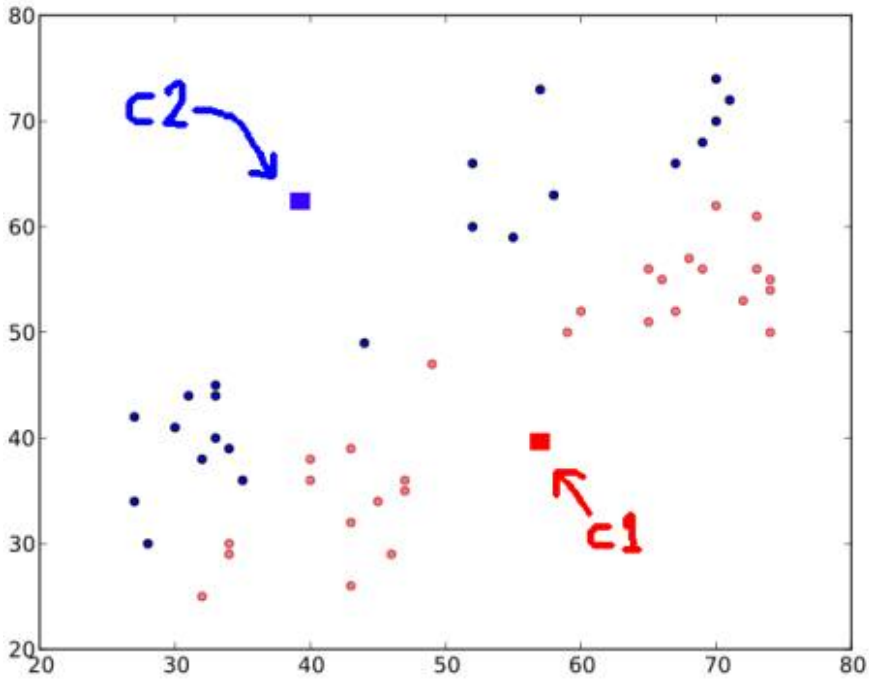


图像

步骤一： 算法随机选择两个中心点 ***C1*** 和 ***C2***（有时，任何两个数据都可以被选为中心点）。

步骤二： 计算从每个点到两个中心点的距离。如果测试数据更接近 ***C1***，则该数据标记为'0'。如果它更接近 ***C2***，则标记为'1'（如果有更多中心点，标记为'2'，'3'等）。

在这个例子中，我们用红色标记所有的'0'，用蓝色标记所有的'1'。在上述操作之后，我们得到了下图：

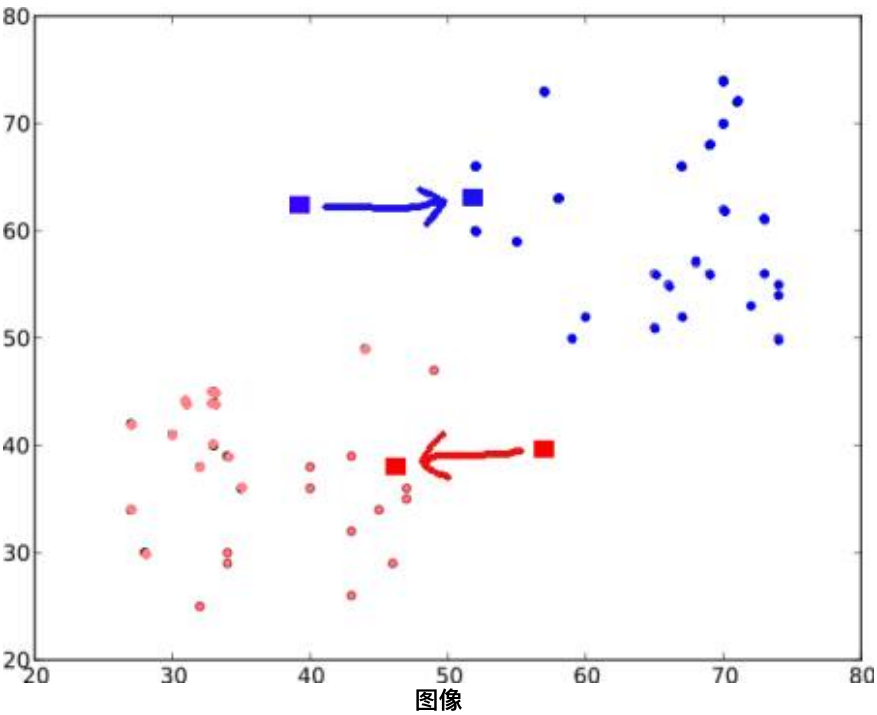


图像

步骤三： 接下来，我们分别计算所有蓝点和红点的平均值，这将是我们的新中心点。将 ***C1*** 和 ***C2*** 移过去。（要记住，这个图像所显示的并不是真正的值，也不是真正的规模，它仅用于演示）。

再来一次，使用新的中心点执行步骤二，并将数据标记为'0'和'1'。

然后我们得到了如下结果：

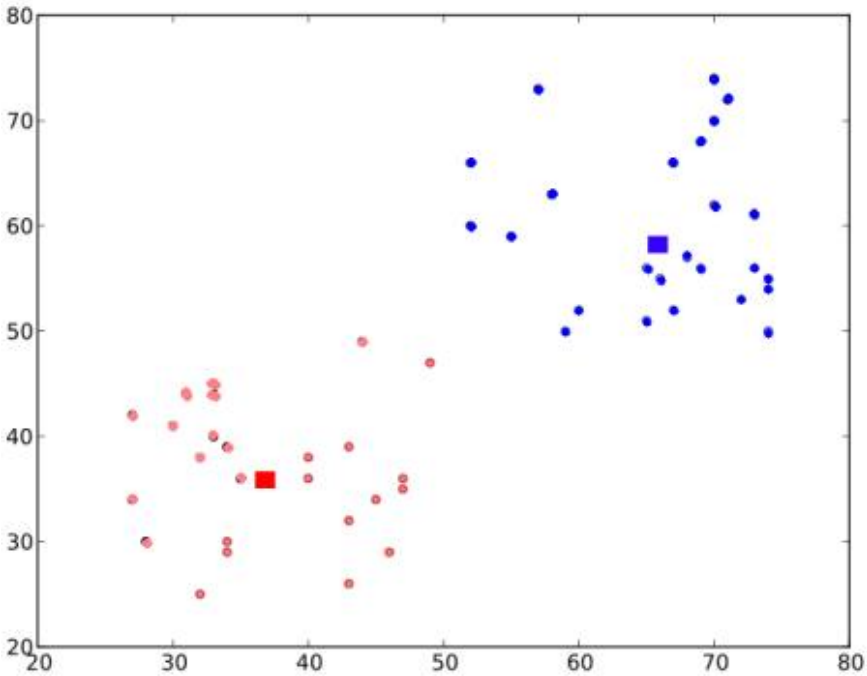


图像

现在，迭代 **步骤二** 和 **步骤三**，直到两个中心点收敛到固定点。*(或者可能根据我们所给的条件停止，如最大迭代次数，或达到特定的准去率等。)* **这些点使得测试数据与其对应的中心点之间的距离之和最小。** 或者简单地说，***C1↔红色点*** 和 ***C2↔蓝色点*** 之间的距离之和最小。

$$minimize \left[J = \sum_{All\ Red\_Points} distance(C1, Red\_Point) + \sum_{All\ Blue\_Points} distance(C2, Blue\_Point) \right]$$

最终结果大概如下所示：



图像

这只是对 K-Means 聚类的直观理解。有关更多详细信息和数学解释，请阅读任何规范的机器学习教科书或阅读额外资源中的链接。这只是 K-Means 聚类的顶层。这个算法有很多修改，如如何选择初始中心点，如何加速迭代过程等。

额外资源

1. 机器学习课程，Andrew Ng 教授的视频讲座（部分图片取自于此）

练习

OpenCV 中的 K-Means 聚类

目标

- 学习在 OpenCV 中使用 cv.kmeans() 函数进行数据聚类

了解参数

输入参数

1. **samples** : 应该是 np.float32 数据类型，并且每个功能都应该放在一个列中。
2. **nclusters(K)** : 结束时所需的集群数量
3. **criteria** : 迭代终止标准。满足此条件时，算法迭代停止。实际上，这是一个有 3 个元素的 tuple。它们是`( type, max_iter, epsilon )`:
   i. 终止标准的类型。 它有 3 个标志如下:
      - **cv.TERM_CRITERIA_EPS** - 如果达到指定的精度 epsilon，则停止算法迭代。
      - **cv.TERM_CRITERIA_MAX_ITER** - 在指定的迭代次数 *max_iter* 之后停止算法。
      - **cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER** - 当满足上述任何条件时停止迭代。
   ii. max_iter - 一个指定最大迭代次数的整数。
   iii. epsilon - 要求的精度
4. **attempts** : 用于指定使用不同初始标记执行算法的次数的标志。算法返回产生最佳紧凑性的标签。该紧凑性作为输出返回。
5. **flags** : 该标志用于指定初始中心的采用方式。通常会使用两个标志: **cv.KMEANS_PP_CENTERS** 和 **cv.KMEANS_RANDOM_CENTERS**。

输出参数

1. **compactness** : 每个点到其相应中心的平方距离之和。
2. **labels** : 标签数组（与前一篇文章中的'代码'相同），其中每个元素都标记为'0'，'1'.....
3. **centers** : 聚类中心数组

现在我们将通过三个例子看到如何应用 K-Means 算法。

1. 只有一个特征的数据

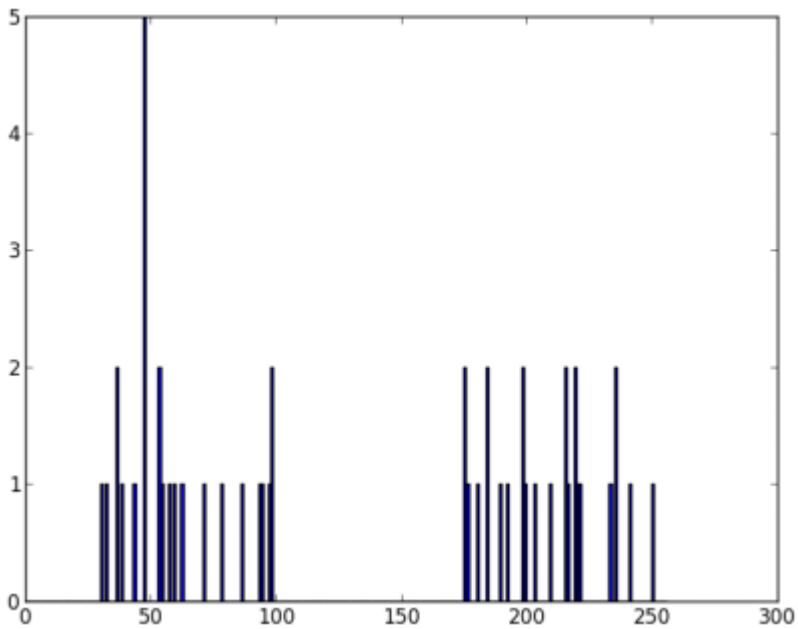
考虑一下，你有一组只有一个特征的数据，即一维。 例如，我们可以采用我们的 T 恤问题，只使用人的高度来决定 T 恤的大小。

因此，我们首先创建数据并在 Matplotlib 中绘制它。

```
import numpy as np
import cv2 as cv
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()
```

所以我们有'z'，这是一个大小为 50 的数组，值范围从 0 到 255。我已经将'z' 重塑为列向量。当存在多个特征时，它将更有用。然后我将类型数据转为 np.float32 。

我们得到以下图片：



图像

现在我们应用 KMeans 功能。在此之前，我们需要指定标准。我的标准是，每当运行 10 次迭代算法或达到 epsilon = 1.0 的精度时，就停止算法并返回答案。

```
# Define criteria = ( type, max_iter = 10 , epsilon = 1.0 )
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 10, 1.0)

# Set flags (Just to avoid line break in the code)
flags = cv.KMEANS_RANDOM_CENTERS

# Apply KMeans
compactness,labels,centers = cv.kmeans(z,2,None,criteria,10,flags)
```

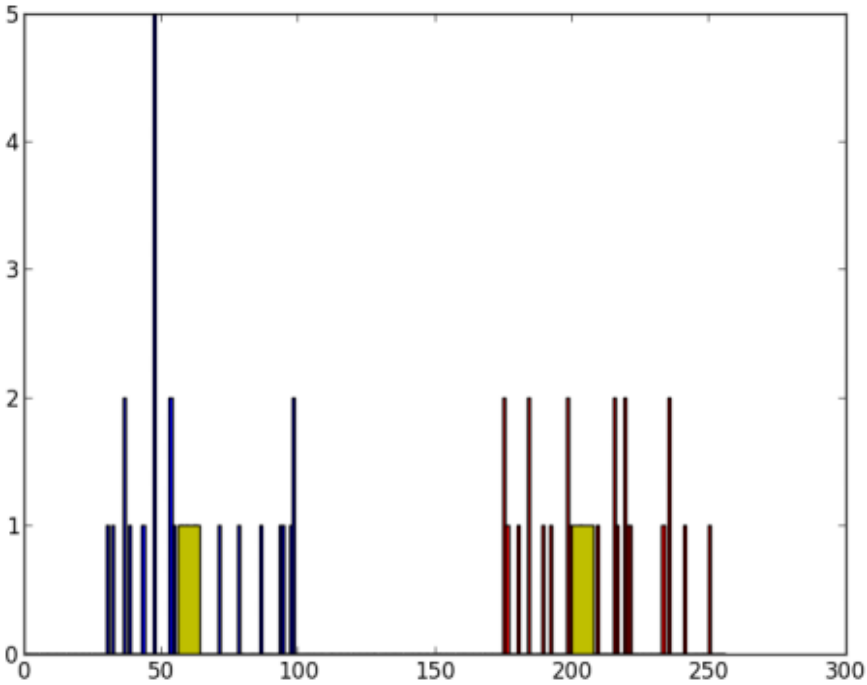
这为我们提供了紧凑性、标签和中心。 在这个例子下，我得到的中心为 60 和 207。标签将具有与测试数据相同的大小，其中每个数据将被标记为“0”，“1”，“2”等，具体取决于它们的中心。现在我们根据标签将数据拆分到不同的集群。

```
A = z[labels==0]
B = z[labels==1]
```

现在我们用红色绘制 A，用蓝色绘制 B，用黄色绘制中心。

```
# 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()
```

以下是我们得到的输出：



图像

2.具有多个特征的数据

在前面的例子中，我们只采取了 T 恤问题的高度。在这里，我们将采用高度和重量，即两个特征。

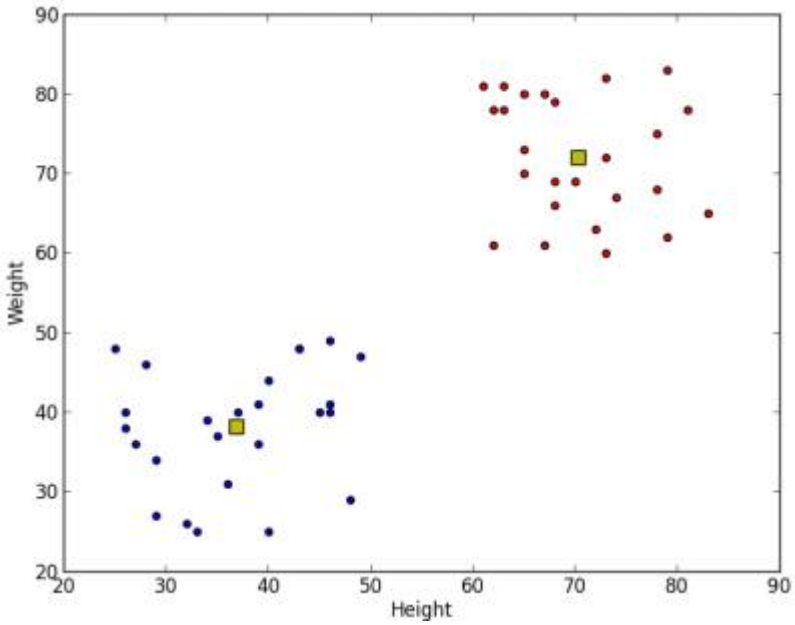
请记住，在前一种情况下，我们将数据制作为单个列向量。 每个特征排列在一列中，而每行对应一个输入测试样本。

举个例子，在本例中，我们将测试数据集的大小设为 50x2 ，这是 50 个人的高度和体重。第一列对应着 50 个人的高度，第二列对应着他们的重量。第一行包含两个元素，其中第一个元素是第一人的高度，第二个元素是他的重量。类似地，剩余的行对应于其他人的高度和重量。查看下图：

Features		
	Height	Weight
Person 1	H1	W1
Person 2	H2	W2
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
Person 50	H50	W50

图像

现在我直接转到代码： ```python import numpy as np import cv2 as cv 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 = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 10, 1.0) ret,label,center=cv.kmeans(Z,2,None,criteria,10,cv.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() ``` 以下是我们得到的输出：



图像

3.色彩量化

色彩量化是减少图像中颜色数量的过程。这样做的一个原因是减少内存。有时，某些设备可能具有限制，使得它只能产生有限数量的颜色。在那些情况下，也执行色彩量化。这里我们使用 k-means 聚类进行色彩量化。

这里没有什么新的概念需要解释。有 3 个特征，叫做 R、G、B。因此，我们需要将图像重塑为 Mx3 大小的数组（M 是图像中的像素数）。在聚类之后，我们将中心值（它也是 R、G、B）应用于所有像素，这样得到的图像将具有指定数量的颜色。

```
import numpy as np
import cv2 as cv

img = cv.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 = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 8
ret,label,center=cv.kmeans(Z,K,None,criteria,10,cv.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))

cv.imshow('res2',res2)
cv.waitKey(0)
cv.destroyAllWindows()
```

对于 K = 8，请参见下面的结果：

图像

额外资源

练习

图像去噪

目标

在这一章当中，

- 您将了解非局部均值去噪算法以消除图像中的噪声。
- 你会看到不同的功能，如 **cv.fastNlMeansDenoising（）** ，**cv.fastNlMeansDenoisingColored（）** 等。

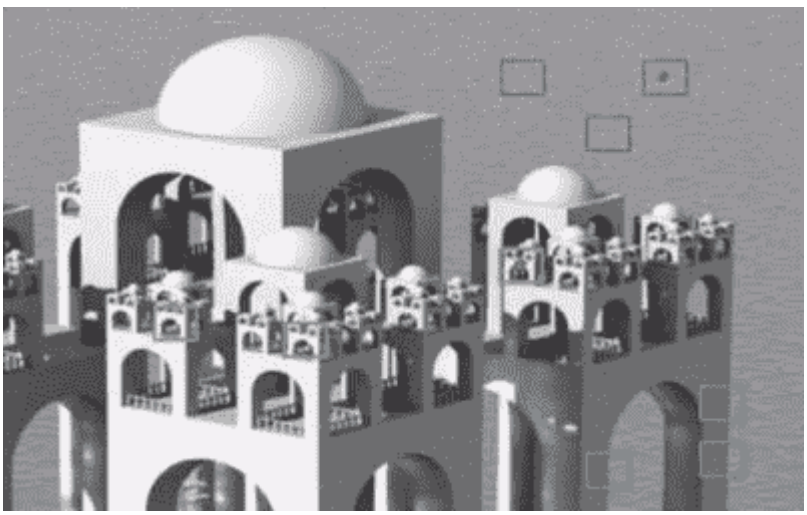
理论

在前面的章节中，我们已经看到许多图像平滑技术，如高斯模糊，中位模糊等，它们在一定程度上消除了少量噪声。在这些技术中，我们在像素周围采用了一个小邻域，并进行了一些操作，如高斯加权平均值，值的中值等，以替换中心元素。简而言之，像素处的噪声消除是其邻域的局部。

有噪音的财产。噪声通常被认为是零均值的随机变量。考虑一个有噪声的像素，$(p = p_0 + n)$ 其中 (p_0) 是像素的真值，(n) 是该像素中的噪声。您可以从不同的图像中获取大量相同的像素例如 $(N) $ 并计算它们的平均值。理想情况下，你应该得到 $(p = p_0)$，因为噪声的平均值为零。

您可以通过简单的设置自行验证。将静态相机固定在某个位置几秒钟。这将为您提供大量的帧或同一场景的大量图像。然后写一段代码来查找视频中所有帧的平均值（这对你来说应该太简单了）。比较最终结果和第一帧。你可以看到减少噪音。不幸的是，这种简单的方法对相机和场景运动不稳健。通常也只有一个嘈杂的图像。

所以想法很简单，我们需要一组类似的图像来平均噪音。考虑图像中的一个小窗口（比如 5x5 窗口）。机会很大，相同的补丁可能在图像中的其他位置。有时在它附近的一个小社区。如何一起使用这些类似的补丁并找到它们的平均值？对于那个特定的窗口，那很好。请参阅下面的示例图片：



图像中的蓝色斑块看起来很相似。绿色斑块看起来相似。所以我们拍摄一个像素，在它周围采取小窗口，在图像中搜索类似的窗口，平均所有窗口并用我们得到的结果替换像素。该方法是非局部均值去噪。与我们之前看到的模糊技术相比，它需要更多的时间，但结果非常好。更多详细信息和在线演示可以在其他资源的第一个链接中找到。

对于彩色图像，图像被转换为CIELAB 色彩空间，然后单独对 L 和 AB 分量进行去噪。

OpenCV 中的图像去噪

OpenCV 提供了这种技术的四种变体。

1. **cv.fastNlMeansDenoising（）** - 适用于单个灰度图像
2. **cv.fastNlMeansDenoisingColored（）** - 适用于彩色图像。
3. **cv.fastNlMeansDenoisingMulti（）** - 适用于短时间内拍摄的图像序列（灰度图像）
4. **cv.fastNlMeansDenoisingColoredMulti（）** - 与上述相同，但适用于彩色图像。

常见的论点是：

- h：参数决定滤波器强度。较高的 h 值可以更好地消除噪声，但也会删除图像的细节。 （10 个没问题）
- hForColorComponents：与 h 相同，但仅适用于彩色图像。 （通常与 h 相同）
- templateWindowSize：应该是奇数。 （推荐 7）
- searchWindowSize：应该是奇数。 （推荐 21）

有关这些参数的详细信息，请访问其他资源中的第一个链接。

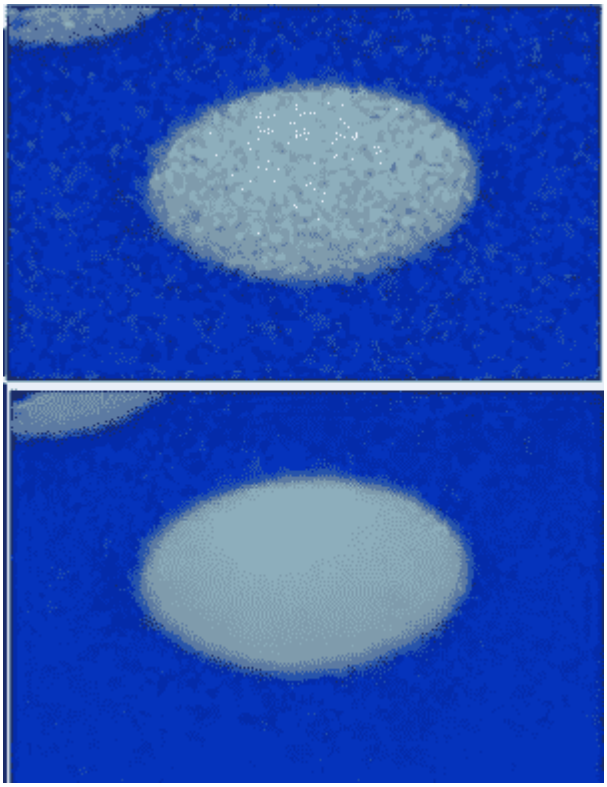
我们将在这里演示 2 和 3。休息留给你。

1. cv.fastNlMeansDenoisingColored（）

如上所述，它用于从彩色图像中去除噪声。 （噪音预计是高斯噪音）。请参阅以下示例：

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('die.png')

dst=cv.fastNlMeansDenoisingColored(img,None,10,10,7,21)

plt.subplot(121),plt.imshow(img)
plt.subplot(122),plt.imshow(dst)
plt.show()
```

下面是结果的缩放版本。我的输入图像的高斯噪声为 $(sigma = 25)$。看结果：

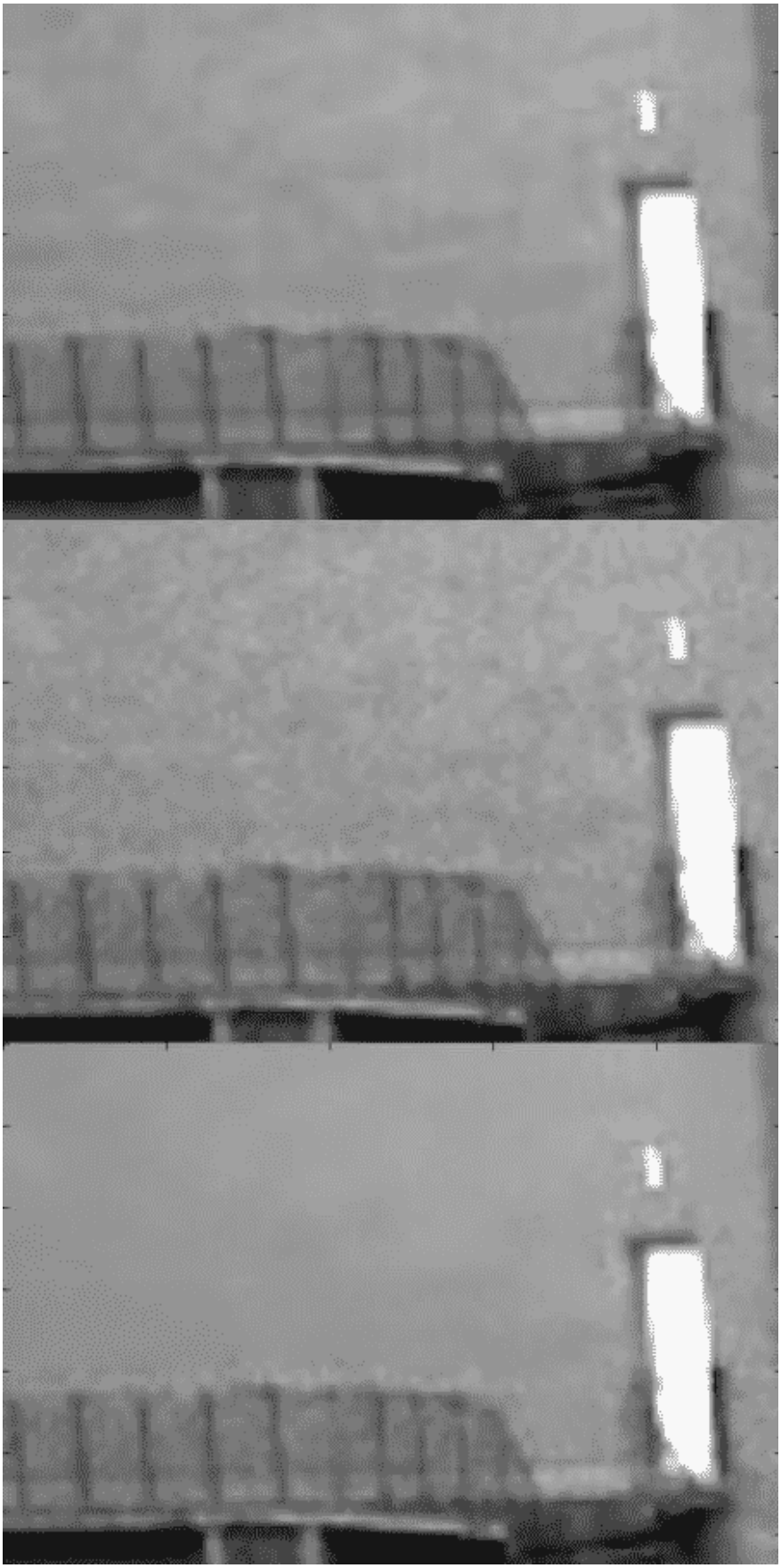


2. cv.fastNlMeansDenoisingMulti（）

现在我们将相同的方法应用于视频。第一个参数是嘈杂帧的列表。第二个参数 imgToDenoiseIndex 指定我们需要去噪的帧，因为我们在输入列表中传递了 frame 的索引。第三个是 temporalWindowSize，它指定了用于去噪的附近帧的数量。应该很奇怪。在这种情况下，使用总共 temporalWindowSize 帧，其中中心帧是要去噪的帧。例如，您传递了 5 个帧的列表作为输入。设 imgToDenoiseIndex = 2 和 temporalWindowSize = 3.然后使用 frame-1，frame-2 和 frame-3 对帧-2 进行去噪。我们来看一个例子吧。

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

cap = cv.VideoCapture('vtest.avi')
# create a list of first 5 frames
img = [cap.read()[1] for i in xrange(5)]

# convert all to grayscale
gray = [cv.cvtColor(i, cv.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 = cv.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()
```

下图显示了我们得到的结果的缩放版本：



计算需要相当长的时间。在结果中，第一图像是原始帧，第二图像是噪声图像，第三图像是去噪图像。

其他资源

1. http://www.ipol.im/pub/art/2011/bcm_nlm/ （它有详细信息，在线演示等。强烈建议访问。我们的测试图像是从这个链接生成的）
2. coursera 的在线课程（第一张图片来自这里）

演习

图像修复

目标

在这一章当中，

- 我们将学习如何通过一种称为修复的方法去除旧照片中的小噪音，笔画等
- 我们将在 OpenCV 中看到修复功能。

基本

大多数人会在家里放一些旧的退化照片，上面有一些黑点，一些笔画等。你有没有想过恢复它？我们不能简单地在绘画工具中擦除它们，因为它将简单地用白色结构替换黑色结构，这是没有用的。在这些情况下，使用称为图像修复的技术。基本思路很简单：用相邻像素替换那些坏标记，使其看起来像邻域。考虑下面显示的图像（取自维基百科）：

为此目的设计了几种算法，OpenCV 提供了两种算法。两者都可以通过相同的功能访问， **cv.inpaint（）**

第一种算法基于 2004 年由 Alexandru Telea 撰写的**“基于快速行进方法的图像修复技术”**。它基于快速行进方法。考虑图像中要修复的区域。算法从该区域的边界开始，然后进入区域内，逐渐填充边界中的所有内容。它需要在邻域像素周围的一个小邻域进行修复。该像素由邻域中所有已知像素的归一化加权和代替。选择权重是一个重要的问题。对于靠近该点的那些像素，靠近边界的法线和位于边界轮廓上的那些像素，给予更多的权重。一旦像素被修复，它将使用快速行进方法移动到下一个最近的像素。 FMM 确保首先修复已知像素附近的像素，这样它就像手动启发式操作一样工作。使用标志 cv.INPAINT_TELEA 启用此算法。

第二种算法基于 Bertalmio，Marcelo，Andrea L. Bertozzi 和 Guillermo Sapiro 在 2001 年的论文**“Navier-Stokes，流体动力学和图像和视频修补”**。该算法基于流体动力学和利用偏微分方程。基本原则是 heurisitic。它首先沿着已知区域的边缘行进到未知区域（因为边缘是连续的）。它继续等照片（连接具有相同强度的点的线，就像轮廓连接具有相同高度的点一样），同时在修复区域的边界处匹配渐变矢量。为此，使用来自流体动力学的一些方法。获得颜色后，填充颜色以减少该区域的最小差异。使用标志 cv.INPAINT_NS 启用此算法。

代码

我们需要创建一个与输入图像大小相同的掩码，其中非零像素对应于要修复的区域。其他一切都很简单。我的图像因一些黑色笔画而降级（我手动添加）。我用 Paint 工具创建了相应的笔触。

```
import numpy as np
import cv2 as cv

img = cv.imread('messi_2.jpg')
mask = cv.imread('mask2.png',0)
dst = cv.inpaint(img,mask,3,cv.INPAINT_TELEA)

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

请参阅下面的结果。第一张图像显示降级输入。第二个图像是面具。第三个图像是第一个算法的结果，最后一个图像是第二个算法的结果。

其他资源

1. Bertalmio，Marcelo，Andrea L. Bertozzi 和 Guillermo Sapiro。“Navier-stokes，流体动力学，图像和视频修复。”在计算机视觉和模式识别，2001 年.CVPR 2001. 2001 年 IEEE 计算机学会会议论文集，第一卷。 1，pp.I-355。 IEEE，2001。
2. Telea，Alexandru。“基于快速行进方法的图像修复技术。” Journal of graphics tools 9.1（2004）： 23-34。

演习

1. OpenCV 附带了一个关于 inpainting 的示例/ sample / python / inpaint.py 的交互式示例，试一试。
2. 几个月前，我观看了一个关于 Content-Aware Fill 的视频，这是 Adobe Photoshop 中使用的一种先进的修复技术。在进一步搜索时，我能够发现 GIMP 中已经存在相同的技术，具有不同的名称，“Resynthesizer”（您需要安装单独的插件）。我相信你会喜欢这项技术。

高动态范围（HDR）

目标

在本章中，我们将

- 了解如何从曝光序列生成和显示 HDR 图像。
- 使用曝光融合来合并曝光序列。

理论

高动态范围成像（HDRI 或 HDR）是一种用于成像和摄影的技术，可以再现比标准数字成像或照相技术更大的动态光度范围。虽然人眼可以适应各种光线条件，但大多数成像设备每通道使用 8 位，因此我们仅限于 256 级。当我们拍摄现实世界场景的照片时，明亮区域可能会过度曝光，而暗区域可能曝光不足，因此我们无法使用单次曝光捕捉所有细节。 HDR 成像适用于每通道使用 8 位以上（通常为 32 位浮点值）的图像，从而允许更宽的动态范围。

获得 HDR 图像的方法有很多种，但最常见的方法是使用不同曝光值拍摄的场景照片。要结合这些曝光，了解相机的响应函数是有用的，并且有估算它的算法。合并 HDR 图像后，必须将其转换回 8 位以在通常的显示器上查看。此过程称为色调映射。当场景或相机的物体在镜头之间移动时会出现额外的复杂性，因为应该注册和对齐具有不同曝光的图像。

在本教程中，我们展示了 2 种算法（Debevec，Robertson），用于从曝光序列生成和显示 HDR 图像，并演示了一种称为曝光融合（Mertens）的替代方法，该方法产生低动态范围图像，不需要曝光时间数据。此外，我们估计相机响应函数（CRF），这对许多计算机视觉算法具有重要价值。 HDR 流水线的每个步骤都可以使用不同的算法和参数来实现，因此请查看参考手册以查看所有步骤。

曝光顺序 HDR

在本教程中，我们将查看以下场景，其中我们有 4 个曝光图像，曝光时间为：15,2.5,1 / 4 和 1/30 秒。 （您可以从维基百科下载图像）

1.将曝光图像加载到列表中

第一阶段只是将所有图像加载到列表中。此外，我们还需要常规 HDR 算法的曝光时间。注意数据类型，因为图像应该是 1 通道或 3 通道 8 位（np.uint8），曝光时间需要为 float32，并且以秒为单位。

```
import cv2 as cv
import numpy as np

# Loading exposure images into a list
img_fn = ["img0.jpg", "img1.jpg", "img2.jpg", "img3.jpg"]
img_list = [cv.imread(fn) for fn in img_fn]
exposure_times = np.array([15.0, 2.5, 0.25, 0.0333], dtype=np.float32)
```

2.将曝光合并到 HDR 图像中

在这个阶段，我们将曝光序列合并为一个 HDR 图像，显示我们在 OpenCV 中有两种可能性。第一种方法是 Debevec，第二种方法是 Robertson。请注意，HDR 图像的类型为 float32，而不是 uint8，因为它包含所有曝光图像的完整动态范围。

```
# Merge exposures to HDR
imagemerge_debevec = cv.createMergeDebevec()
hdr_debevec = merge_debevec.process(img_list, times=exposure_times.copy())
merge_robertson = cv.createMergeRobertson()
hdr_robertson = merge_robertson.process(img_list, times=exposure_times.copy())
```

3. Tonemap HDR 图像

我们将 32 位浮点 HDR 数据映射到范围[0..1]。实际上，在某些情况下，值可以大于 1 或小于 0，因此请注意我们稍后将不得不剪切数据以避免溢出。

```
# Tonemap HDR image
tonemap1 = cv.createTonemapDurand(gamma=2.2)
res_debevec = tonemap1.process(hdr_debevec.copy())
tonemap2 = cv.createTonemapDurand(gamma=1.3)
res_robertson = tonemap2.process(hdr_robertson.copy())
```

4.使用 Mertens 融合合并曝光

在这里，我们展示了一种合并曝光图像的替代算法，我们不需要曝光时间。我们也不需要使用任何色调图算法，因为 Mertens 算法已经给出了[0..1]范围内的结果。

```
# Exposure fusion using Mertens
merge_mertens = cv.createMergeMertens()
res_mertens = merge_mertens.process(img_list)
```

5.转换为 8 位并保存

为了保存或显示结果，我们需要将数据转换为[0..255]范围内的 8 位整数。

```
# Convert datatype to 8-bit and save
res_debevec_8bit = np.clip(res_debevec*255, 0, 255).astype('uint8')
res_robertson_8bit = np.clip(res_robertson*255, 0, 255).astype('uint8')
res_mertens_8bit = np.clip(res_mertens*255, 0, 255).astype('uint8')

cv.imwrite("ldr_debevec.jpg", res_debevec_8bit)
cv.imwrite("ldr_robertson.jpg", res_robertson_8bit)
cv.imwrite("fusion_mertens.jpg", res_mertens_8bit)
```

结果

您可以看到不同的结果，但考虑到每个算法都有额外的额外参数，您应该适合获得所需的结果。最佳做法是尝试不同的方法，看看哪种方法最适合您的场景。

德普外阁(Debevec)：

罗伯逊(Robertson)：

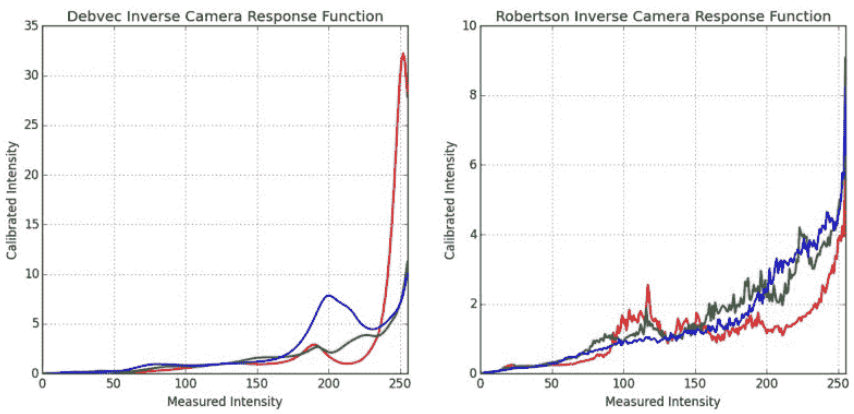
梅特内斯·福森(Mertenes Fusion)：

估计相机响应功能

相机响应功能（CRF）为我们提供了场景辐射与测量强度值之间的连接。 CRF 在一些计算机视觉算法中非常重要，包括 HDR 算法。在这里，我们估计逆相机响应函数并将其用于 HDR 合并。

```
# Estimate camera response function (CRF)
cal_debevec = cv.createCalibrateDebevec()
crf_debevec = cal_debevec.process(img_list, times=exposure_times)

hdr_debevec = merge_debevec.process(img_list,     times=exposure_times.copy(),response

cal_robertson = cv.createCalibrateRobertson()
crf_robertson = cal_robertson.process(img_list, times=exposure_times)

hdr_robertson = merge_robertson.process(img_list, times=exposure_times.copy(), respons
```

相机响应函数由每个颜色通道的 256 长度矢量表示。对于此序列，我们得到以下估计：



其他资源

1. Paul E Debevec 和 Jitendra Malik。从照片中恢复高动态范围辐射图。在 ACM SIGGRAPH 2008 课程中，第 31 页.ACM，2008。 [44]
2. Mark A Robertson，Sean Borman 和 Robert L Stevenson。通过多次曝光改善动态范围。在 Image Processing，1999。ICIP 99.会议录。 1999 年国际会议，第 3 卷，第 159-163 页。 IEEE，1999。 [166]
3. Tom Mertens，Jan Kautz 和 Frank Van Reeth。曝光融合。在计算机图形和应用程序，2007 年.PG'07。第 15 届太平洋会议，第 382-390 页。 IEEE，2007。 [136]
4. 来自 Wikipedia-HDR 的图片

演习

1. 尝试所有色调图算法： cv :: TonemapDrago ， cv :: TonemapDurand ， cv :: TonemapMantiuk 和 cv :: TonemapReinhard
2. 尝试更改 HDR 校准和色调映射方法中的参数。

使用 Haar Cascades 进行人脸检测

目标

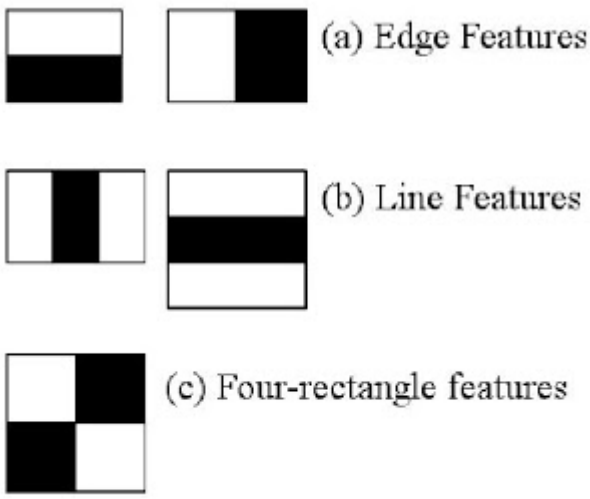
在本次会议中:

- 我们将看到使用基于 Haar 特征的级联分类器进行人脸检测的基础知识
- 我们将对眼睛检测等进行扩展。

基本

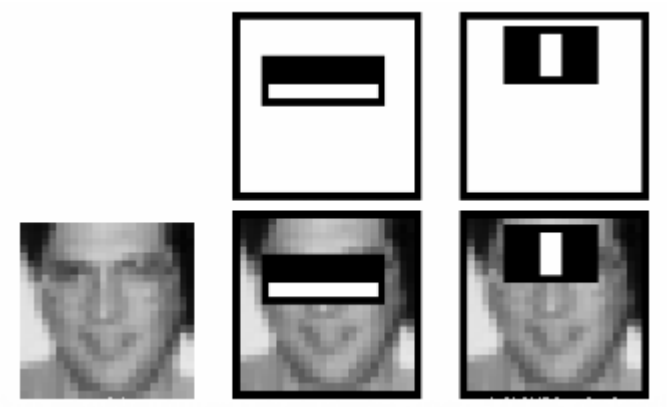
使用基于 Haar 特征的级联分类器的对象检测是 Paul Viola 和 Michael Jones 在他们的论文“使用简单特征的 Boosted 级联的快速对象检测”中提出的有效的对象检测方法。它是一种基于机器学习的方法，其中 a 级联功能是从许多正面和负面图像中训练出来的。然后它用于检测其他图像中的对象。

在这里，我们将使用面部检测。最初，该算法需要许多正图像（面部图像）和负图像（没有面部的图像）来训练分类器。然后我们需要从中提取特征。为此，使用下图所示的 Haar 特征。它们就像我们的卷积内核一样。每个特征是通过从黑色矩形下的像素之和减去白色矩形下的像素之和而获得的单个值。



现在，每个内核的所有可能大小和位置都用于计算大量功能。（想象一下它需要多少计算？即使 24x24 窗口也会产生超过 160000 个特征）。对于每个特征计算，我们需要找到白色和黑色矩形下的像素之和。为了解决这个问题，他们引入了积分图像。无论图像多大，它都会将给定像素的计算减少到仅涉及四个像素的操作。不错，不是吗？它使事情超级快。

但在我们计算的所有这些功能中，大多数都是无关紧要的。例如，请考虑下面的图像。第一行显示了两个很好的功能。选择的第一个特征似乎集中在眼睛区域通常比鼻子和脸颊区域更暗的属性。选择的第二个特征依赖于眼睛比鼻梁更暗的特性。但适用于脸颊或任何其他地方的窗户是无关紧要的。那么我们如何从 160000 多个功能中选择最佳功能呢？这是由 Adaboost 实现的。



为此，我们在所有训练图像上应用每个特征。对于每个特征，它会找到最佳阈值，将面部分类为正面和负面。显然，会出现错误或错误分类。我们选择具有最小错误率的特征，这意味着它们是最准确地对面部和非面部图像进行分类的特征。（过程并不像这样简单。每个图像在开始时给予相同的权重。在每次分类之后，错误分类图像的权重增加。然后进行相同的处理。计算新的错误率。还有新的权重。继续处理，直到达到所需的精度或错误率或找到所需的特征数量）。

最终的分类器是这些弱分类器的加权和。它被称为弱，因为它不能单独对图像进行分类，但与其他图像一起形成一个强大的分类器。该报称，即使是 200 种功能也能提供 95%的检测精度。他们的最终设置有大约 6000 个功能。（想象一下，从 160000 多个功能减少到 6000 个功能。这是一个很大的收获）。

所以现在你拍了一张照片。每个 24x24 窗口。应用 6000 功能。检查它是否是面部。哇..是不是有点低效和耗时？是的。作者有一个很好的解决方案。

在图像中，大部分图像是非面部区域。因此，有一个简单的方法来检查窗口是否不是面部区域是一个更好的主意。如果不是，请一次丢弃，不要再处理。相反，要关注可能有面孔的区域。这样，我们花费更多时间检查可能的面部区域。

为此，他们引入了 **Cascade of Classifiers** 的概念。这些功能不是在窗口上应用所有 6000 个功能，而是分组到不同的分类器阶段并逐个应用。（通常前几个阶段将包含很少的功能）。如果窗口在第一阶段失败，则将其丢弃。我们不考虑其余的功能。如果通过，则应用第二阶段功能并继续该过程。通过所有阶段的窗口是面部区域。那个计划怎么样！

作者的探测器具有 6000 多个特征，其中 38 个阶段在前五个阶段具有 1 个，10 个，25 个，25 个和 50 个特征。（上图中的两个特征实际上是 Adaboost 中最好的两个特征）。据作者说，平均每个子窗口评估了 6000 多个特征中的 10 个特征。

因此，这是 Viola-Jones 面部检测工作原理的简单直观解释。阅读本文以获取更多详细信息，或查看“其他资源”部分中的参考资料。

OpenCV 中的 Haar-cascade 检测

OpenCV 配有训练器和探测器。如果您想为汽车，飞机等任何对象训练自己的分类器，您可以使用 OpenCV 创建一个。它的全部细节在这里给出：级联分类器训练。

在这里我们将处理检测。OpenCV 已经包含许多面部，眼睛，微笑等预先训练的分类器。这些 XML 文件存储在 opencv/data/haarcascades/文件夹 中。让我们用 OpenCV 创建一个面部和眼睛探测器。

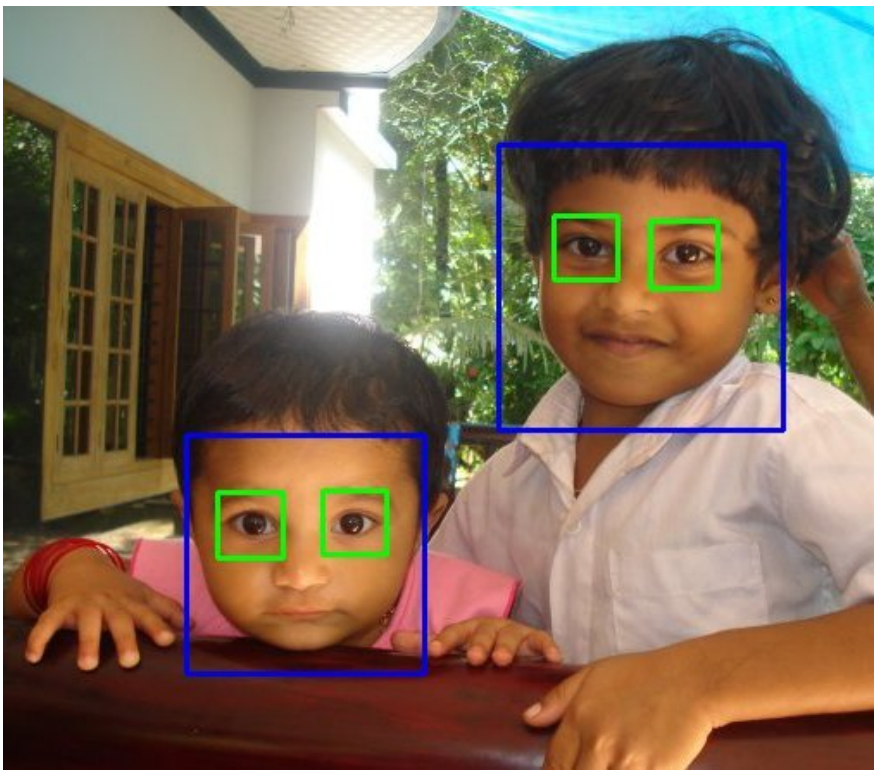
首先，我们需要加载所需的 XML 分类器。然后以灰度模式加载输入图像（或视频）。

```
import numpy as np
import cv2 as cv
face_cascade = cv.CascadeClassifier('haarcascade_frontalface_default.xml')
eye_cascade = cv.CascadeClassifier('haarcascade_eye.xml')
img = cv.imread('sachin.jpg')
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
```

现在我们在图像中找到面孔。如果找到了面，它会将检测到的面的位置返回为 Rect（x，y，w，h）。一旦我们获得这些位置，我们就可以为脸部创建投资回报率，并在此投资回报率上应用眼睛检测（因为眼睛总是在脸上!!!）。

```
faces = face_cascade.detectMultiScale(gray, 1.3, 5)
for (x,y,w,h) in faces:
    cv.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:
        cv.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)
cv.imshow('img',img)
cv.waitKey(0)
cv.destroyAllWindows()
```

结果如下所示：



其他资源

- 关于 人脸检测和跟踪 的视频讲座
- Adam Harvey 关于人脸探测的有趣采访

练习

如何生成 OpenCV-Python 绑定?

在 OpenCV 中，所有算法都是用 C ++实现的。但是这些算法可以用于不同的语言，如 Python，Java 等。这可以通过绑定生成器实现。这些生成器在 C++和 Python 之间架起了一座桥梁，使用户能够从 Python 调用 C++函数。要全面了解后台发生的情况，需要熟悉 Python / C 的 API。有关将 C++函数扩展到 Python 的简单示例可以在官方 Python 文档中找到[1]。因此，通过手动编写包装函数将 OpenCV 中的所有函数扩展到 Python 是一项非常耗时的任务。所以 OpenCV 以更智能的方式完成它。 OpenCV 使用一些位于 `modules/python/src2` 中的 Python 脚本从 C++头文件自动生成这些包装函数。我们将研究他们的工作。

首先， `modules/python/CMakeFiles.txt` 是一个 CMake 脚本，它检查要扩展到 Python 的模块。它会自动检查所有要扩展的模块并获取它们的头文件。这些头文件包含该特定模块的所有类、函数、常量等的列表。

其次，将这些头文件传递给 Python 脚本 `modules/python/src2/gen2.py` 。这是 Python 绑定生成器脚本。它调用另一个 Python 脚本 `modules/python/src2/hdr_parser.py` 。这是头解析器脚本。此标头解析器将完整的头文件拆分为小的 Python 列表。因此，这些列表包含有关特定函数、类等的所有详细信息。例如，将解析函数以获取包含函数名称，返回类型，输入参数，参数类型等的列表。最终列表包含所有函数的详细信息，枚举该头文件中的结构、类、等等。

但是头解析器不解析头文件中的所有函数/类。开发人员必须指定应将哪些函数导出到 Python。为此，在这些声明的开头添加了某些宏，这使得标头解析器能够识别要解析的函数。这些宏由编程特定功能的开发人员添加。简而言之，开发人员决定哪些函数应该扩展到 Python，哪些不是。这些宏的详细信息将在下一个会话中给出。

因此，头解析器返回解析函数的最终大列表。我们的生成器脚本（gen2.py）将为头解析器解析的所有函数/类/枚举/结构创建包装器函数（您可以在 `build/modules/python/` 文件夹中编译这些头文件作为 pyopencv*generated*.h 文件）。但是可能有一些基本的 OpenCV 数据类型，如 Mat，Vec4i，Size。它们需要手动扩展。例如，Mat 类型应该扩展为 Numpy 数组，Size 应该扩展为两个整数的元组等。类似地，可能有一些复杂的结构/类/函数等需要手动扩展。所有这些手动包装函数都放在 `modules/python/src2/cv2.cpp` 中。

所以现在唯一剩下的就是编译这些包装文件，它们为我们提供了 cv2 模块。所以当你在 Python 中调用一个函数，比如 `res = equalizeHist(img1, img2)` 时，你会传递两个 numpy 数组，你期望另一个 numpy 数组作为输出。所以这些 numpy 数组转换为 cv::Mat，然后在 C++中调用 **equalizeHist()** 函数。最终结果，res 将被转换回 Numpy 数组。简而言之，几乎所有操作都是用 C++完成的，这使我们的速度几乎与 C++相同。

所以这是 OpenCV-Python 绑定生成方式的基本版本。

如何将新模块扩展到 Python?

头解析器基于添加到函数声明中的一些包装器宏来解析头文件。枚举常量不需要任何包装器宏。它们是自动包装的。但是剩余的函数、类等需要包装器宏。

函数使用 `CV_EXPORTS_W` 宏进行扩展。下面是一个例子。

```
CV_EXPORTS_W void equalizeHist( InputArray src, OutputArray dst );
```

头解析器可以理解关键字 InputArray、OutputArray 等的输入和输出参数。但有时，我们可能需要硬编码输入和输出。为此，使用了诸如 `CV_OUT` 、 `CV_IN_OUT` 等宏。

```
CV_EXPORTS_W void minEnclosingCircle( InputArray points,
                                      CV_OUT Point2f& center, CV_OUT float& radius );
```

对于大型类，也使用 `CV_EXPORTS_W` 。要扩展类方法，使用 `CV_WRAP` 。类似地， `CV_PROP` 用于类字段。

```
class CV_EXPORTS_W CLAHE : public Algorithm
{
public:
    CV_WRAP virtual void apply(InputArray src, OutputArray dst) = 0;
    CV_WRAP virtual void setClipLimit(double clipLimit) = 0;
    CV_WRAP virtual double getClipLimit() const = 0;
}
```

重载的函数可以使用 CV_EXPORTS_AS 进行扩展。但是我们需要传递一个新名称，以便在 Python 中每个函数都使用这个名称来调用。以下面的积分函数为例。有三个函数可用，因此在 Python 中每个函数都有一个后缀。类似地， `CV_WRAP_AS` 可以用于包装重载的方法。

```
CV_EXPORTS_W void integral( InputArray src, OutputArray sum, int sdepth = -1 );

CV_EXPORTS_AS(integral2) void integral( InputArray src, OutputArray sum,
                                        OutputArray sqsum, int sdepth = -1, int sqdept

CV_EXPORTS_AS(integral3) void integral( InputArray src, OutputArray sum,
                                        OutputArray sqsum, OutputArray tilted,
                                        int sdepth = -1, int sqdepth = -1 );
```

使用 `CV_EXPORTS_W_SIMPLE` 扩展小类/结构。这些结构体通过值传递给 c++函数。例子有关键点，匹配等。它们的方法由 `CV_WRAP` 扩展，字段由 `CV_PROP_RW` 扩展。

```
class CV_EXPORTS_W_SIMPLE DMatch
{
public:
    CV_WRAP DMatch();
    CV_WRAP DMatch(int _queryIdx, int _trainIdx, float _distance);
    CV_WRAP DMatch(int _queryIdx, int _trainIdx, int _imgIdx, float _distance);
    CV_PROP_RW int queryIdx; // query descriptor index
    CV_PROP_RW int trainIdx; // train descriptor index
    CV_PROP_RW int imgIdx;   // train image index
    CV_PROP_RW float distance;
};
```

其他一些小类/结构可以使用 `CV_EXPORTS_W_MAP` 导出，并将其导出到 Python 本地字典。矩()就是一个例子。

```
class CV_EXPORTS_W_MAP Moments
{
public:
    CV_PROP_RW double  m00, m10, m01, m20, m11, m02, m30, m21, m12, m03;
    CV_PROP_RW double  mu20, mu11, mu02, mu30, mu21, mu12, mu03;
    CV_PROP_RW double  nu20, nu11, nu02, nu30, nu21, nu12, nu03;
};
```

这些是 OpenCV 中可用的主要扩展宏。通常，开发人员必须将适当的宏放到适当的位置。Rest 由生成器脚本完成。有时，可能会出现生成器脚本无法创建包装器的异常情况。这类函数需要手动处理，为此编写您自己的 `pyopencv_*.hpp` 扩展头文件，并将它们放到模块的 misc/python 子目录中。但是大多数情况下，根据 OpenCV 编码指南编写的代码将被生成器脚本自动包装。

更高级的情况涉及为 Python 提供 c++接口中不存在的额外特性，如额外的方法、类型映射或提供默认参数。稍后我们将以 `UMat` 数据类型作为此类情况的一个示例。首先，要提供特定于 python 的方法， `CV_WRAP_PHANTOM` 的使用方式与 `CV_WRAP` 类似，不同之处是它以方法头作为参数，您需要在自己的 pyopencv_*中提供方法体。进行扩展。queue()和 `UMat::context()` 就是这种虚方法的一个例子，这种虚方法在 c++接口中不存在，但是在 Python 端处理 OpenCL 功能时需要用到。其次，如果已经存在的数据类型可以映射到您的类，那么最好使用 `CV_WRAP_MAPPABLE` (以源类型作为参数)来表示这种容量，而不是构建您自己的绑定函数。最后，如果需要一个默认参数，但在本机 c++接口中没有提供，那么可以将其作为 `CV_WRAP_DEFAULT` 参数提供给 Python 端。根据下面的 `UMat::getMat` 示例:

```
class CV_EXPORTS_W UMat
{
public:
    // You would need to provide `static bool cv_mappable_to(const Ptr<Mat>& src, Ptr<
    CV_WRAP_MAPPABLE(Ptr<Mat>);
    /! returns the OpenCL queue used by OpenCV UMat.
    // You would need to provide the method body in the binder code
    CV_WRAP_PHANTOM(static void* queue());
    // You would need to provide the method body in the binder code
    CV_WRAP_PHANTOM(static void* context());
    CV_WRAP_AS(get) Mat getMat(int flags CV_WRAP_DEFAULT(ACCESS_RW)) const;
};
```