第十一次实验报告

学号：518030910308
姓名：刘文轩

一、实验准备

1、 实验环境介绍

操作系统：ubuntu 14.04
语言：Python 2
IDE：Pycharm 2019.2.3

2、 实验目的

2.1 了解 Canny 边缘检测的概念
2.2 学会使用 OpenCV 自带的 Canny 边缘检测
2.2 学会自己编写 Canny 边缘检测的算法

3、实验思路

3.1 将获取的图片灰度化
3.2 进行高斯滤波，这使得单独的一个像素噪声在经过高斯滤波的图像上变得几乎没有影响
3.3 计算图像灰度值的梯度
3.4 对梯度非极大值进行抑制
3.5 使用双阈值算法检测和连接边缘

二、实验过程

1、 自行设计 Canny 边缘检测算法

1.1 灰度化

灰度化的方式非常简单，我们只需要在读入图片时加入额外的参数“0”指定通道就可以了：

```
1.  img = cv2.imread("1.jpg", 0)
```

1.2 高斯滤波

数据预处理工作是非常重要的，图像处理也不例外。这里使用高斯滤波进行图像去噪，比如 blur = cv2.GaussianBlur(img, (3,3), 0)，处理后的图像与原始图像相比稍微有些模糊。这样单独的一个像素噪声在经过高斯滤波的图像上变得几乎没有影响：

```
1.  Guass_img = cv2.GaussianBlur(img, (3, 3), 0)
2.  cv2.imshow("Guass_img", Guass_img)
```

1.3 灰一阶偏导的有限差分来计算梯度的幅值和方向

在本次实验中，我使用了 Sobel 算子，

$$s_x=\begin{bmatrix}-1&0&1\\-2&0&2\\-1&0&1\end{bmatrix}, s_y=\begin{bmatrix}1&2&1\\0&0&0\\-1&-2&-1\end{bmatrix} \quad K=\begin{bmatrix}a_0&a_1&a_2\\a_7&[i,j]&a_3\\a_6&a_5&a_4\end{bmatrix}$$

上式三个矩阵分别为该算子的 x 向卷积模板、y 向卷积模板以及待处理点的邻域点标记

矩阵，据此可用数学公式表达其每个点的梯度幅值为：

$$G[i,j]=\sqrt{s_x^2+s_y^2}$$

$$s_x=(a_2+2a_3+a_4)-(a_0+2a_7+a_6)$$
$$s_y=(a_0+2a_1+a_2)-(a_6+2a_5+a_4)$$

需要注意的是，我们在调用函数计算 Sobel 算子时，Sobel 函数求导后会有负值和>255 的值，因而需先使用 16 位有符号的数据类型，即 cv2.CV_16S，再转为 unit8 类型。

同时，在 OpenCV-Python 中，使用 Sobel 的算子的函数原型如下：

```
1.  dst = cv2.Sobel(src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderTy
    pe]]]]])
```

前四个是必须的参数：

第一个参数是需要处理的图像；

第二个参数是图像的深度，-1 表示采用的是与原图像相同的深度。目标图像的深度必须大于等于原图像的深度；

dx 和 dy 表示的是求导的阶数，0 表示这个方向上没有求导，一般为 0、1、2。

其后是可选的参数：

ksize 是 Sobel 算子的大小，必须为 1、3、5、7。

scale 是缩放导数的比例常数，默认情况下没有伸缩系数；

delta 是一个可选的增量，将会加到最终的 dst 中，同样，默认情况下没有额外的值加到 dst 中；

borderType 是判断图像边界的模式。这个参数默认值为 cv2.BORDER_DEFAULT。

此部分完整相关代码如下：

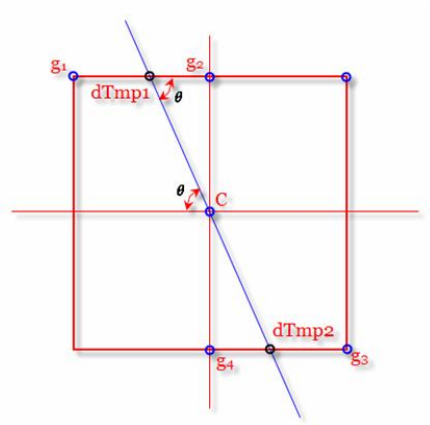
```
1.  x = cv2.Sobel(Guass_img, cv2.CV_16S, 1, 0)
2.  y = cv2.Sobel(Guass_img, cv2.CV_16S, 0, 1)
3.
4.  absX = cv2.convertScaleAbs(x)
5.  absY = cv2.convertScaleAbs(y)
6.
7.  dst = cv2.addWeighted(absX, 0.5, absY, 0.5, 0)
```

1.4 对梯度幅值进行非极大值抑制

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

这一部分的运算直接使用了 numpy 自带的矩阵相除的功能，但是由于发现 Y 方向的梯度矩阵中包含了 0 值，为了避免程序报警告，将 Y 方向的矩阵的每一个值都加入一个极小值 1e-8。

接下来就是进行非极大值的幅值抑制。

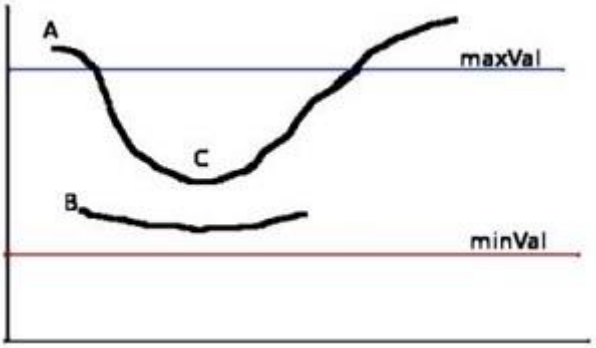


正如图片所示，蓝色线条方向为 C 点的梯度方向，C 点局部的最大值则分布在这条线上。即除 C 点外，还需要判定 dTmp1 和 dTmp2 两点灰度值。若 C 点灰度值小于两点中任一个，则 C 点不是局部极大值，因而可以排除 C 点为边缘点。此部分的完整代码如下：

```
absX = absX
absY = absY + 1e-8
angle = np.floor_divide(absX, absY).astype(int)

height = len(dst)
width = len(dst[0])

for i in range(1, height - 2):
    for j in range(1, width - 2):
        k = angle[i][j]
        if k >= 1:
            dTmp1 = (1 - 1 / k) * dst[i - 1][j] + (1 / k) * dst[i - 1][j + 1]
            dTmp2 = (1 - 1 / k) * dst[i + 1][j] + (1 / k) * dst[i + 1][j - 1]
        elif 0 <= k < 1:
            dTmp1 = k * dst[i - 1][j + 1] + (1 - k) * dst[i][j + 1]
            dTmp2 = k * dst[i + 1][j - 1] + (1 - k) * dst[i][j - 1]
        elif -1 <= k < 0:
            dTmp1 = -k * dst[i - 1][j - 1] + (1 + k) * dst[i][j - 1]
            dTmp2 = -k * dst[i + 1][j + 1] + (1 + k) * dst[i][j + 1]
        elif k < -1:
            dTmp1 = (1 + 1 / k) * dst[i - 1][j] - (1 / k) * dst[i - 1][j - 1]
            dTmp2 = (1 + 1 / k) * dst[i + 1][j] - (1 / k) * dst[i + 1][j + 1]

        value = dst[i][j]
        if value < dTmp1 or value < dTmp2:
            dst[i][j] = 0
```

1.5 双阈值算法检测和连接边缘

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



A 高于阈值 maxVal 所以是真正的边界点，C 虽然低于 maxVal 但高于 minVal 并且与 A 相连，所以也被认为是真正的边界点。而 B 就会被抛弃，因为它不仅低于 maxVal 而且不与真正的边界点相连。

因此我在设计代码时，使用了 th1 与 th2 将图像中的像素点分为了三个部分：低阈值（0）、中阈值（1）、高阈值（2）。在这三部分中，以高阈值形成的图像为基础，在图像中的边缘进行在中阈值的对周围点的搜索，并且将其补充进去作为新的边缘。

完成操作后，将高阈值置为 255，输出白色的边缘。

```
th1 = 30
th2 = 70

for i in range(0, height):
    for j in range(0, width):
        if dst[i][j] >= th2:
            dst[i][j] = 2
        elif th1 < dst[i][j] < th2:
            dst[i][j] = 1
        else:
            dst[i][j] = 0

for i in range(1, height - 1):
    for j in range(1, width - 1):
        if dst[i][j] == 1:
            if dst[i + 1][j + 1] == 2 or dst[i + 1][j - 1] == 2 or dst[i - 1][j + 1] == 2 \
                    or dst[i - 1][j - 1] == 2 or dst[i + 1][j] == 2 or dst[i - 1][j] == 2 \
                    or dst[i][j + 1] == 2 or dst[i][j - 1] == 2:
                dst[i][j] = 2

for i in range(1, height - 1):
```

```
    for j in range(1, width - 1):
        if dst[i][j] == 2:
            dst[i][j] = 255

cv2.imshow("Data", dst)

edges = cv2.Canny(Guass_img, 50, 150, apertureSize=3)
cv2.imshow("Edges", edges)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

2、实现检测

2.1 OpenCV 实现 Canny 边缘检测

用自带的函数实现 Canny 边缘检测十分简单，只需要将高斯模糊后的图形，直接传递给对应的函数即可，

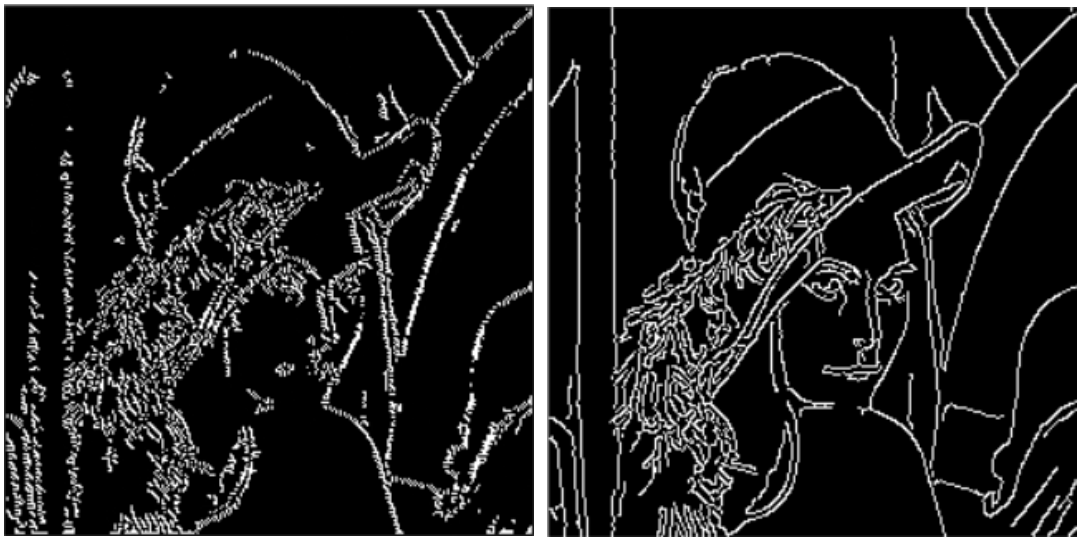
```
edges = cv2.Canny(Guass_img, 50, 150, apertureSize=3)
cv2.imshow("Edges", edges)
```

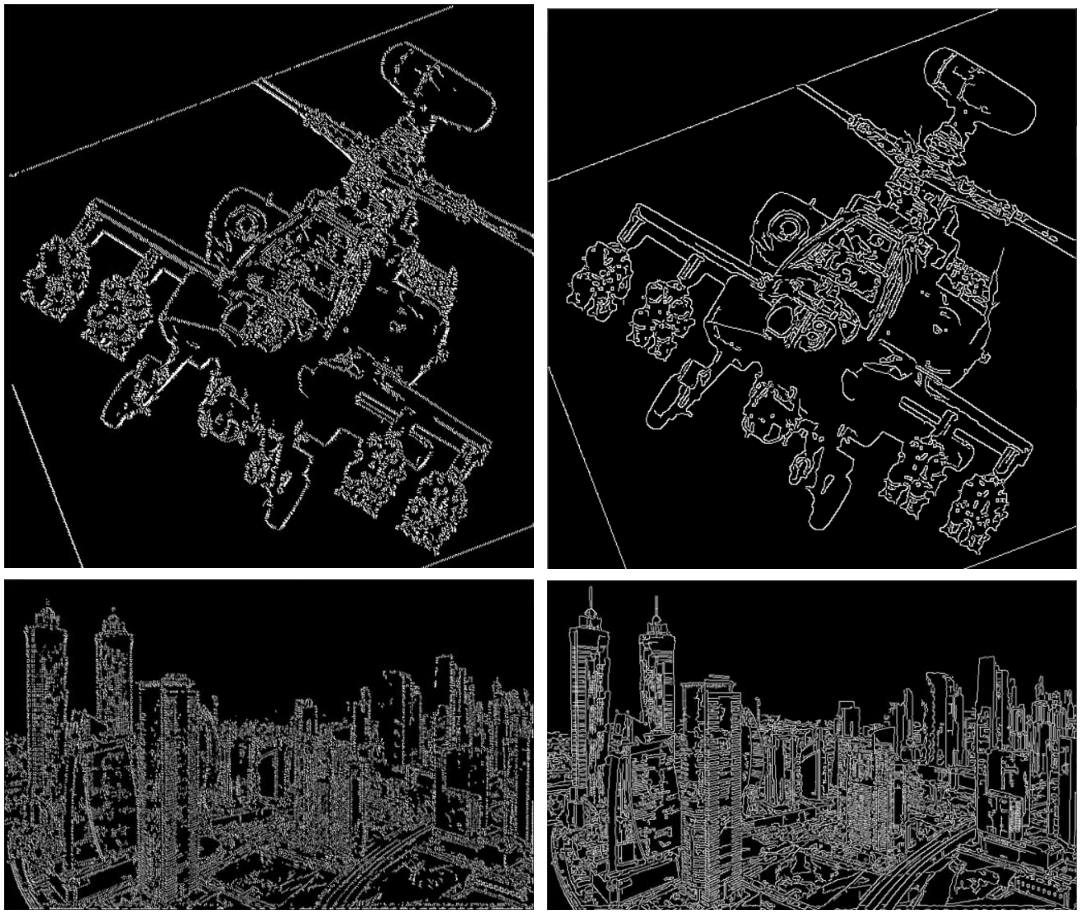
其中 Canny 的第一个参数是传入的图片，第二和第三个参数是低阈值和高阈值的取值。

2.2 结果对比

我们的三张原图如下：

对比我自行设计的代码的结果（左）与 OpenCV 自带函数的结果（右），分别如下：





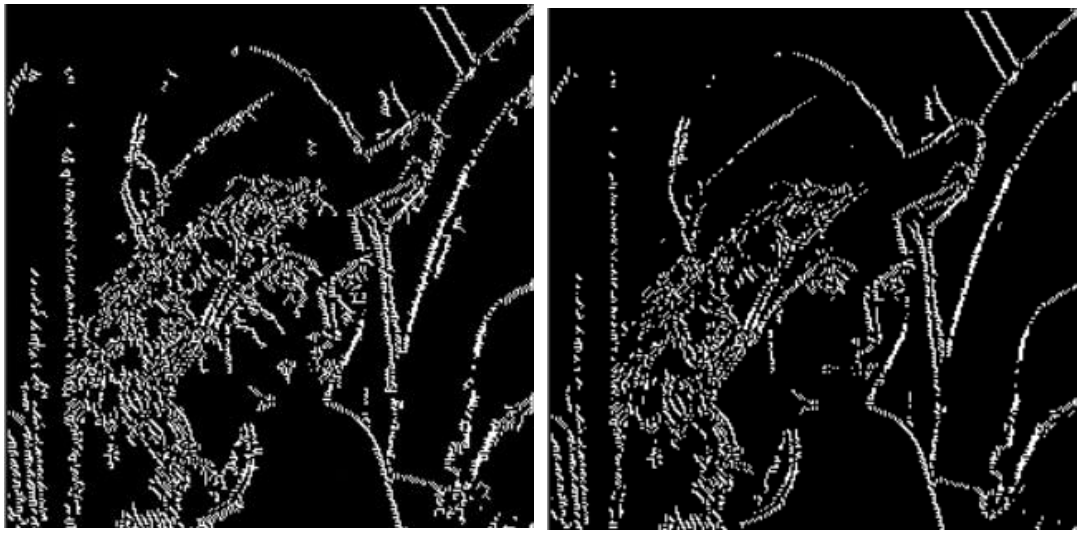
由图可见，大体上我的代码的实现结果已经相当不错，只是在图像的连续性上与 OpenCV 自带的函数运行效果有一定差距。

3、 有关双阈值选择的拓展

3.1 低阈值的选择

我在上面自己设计的代码中，选择的阈值为 30 和 70。下面我们以图片 1 为例，探讨不同阈值的影响。

此部分中，保持高阈值为 70 不变，改变低阈值为 10（左）以及 60（右），结果如图:

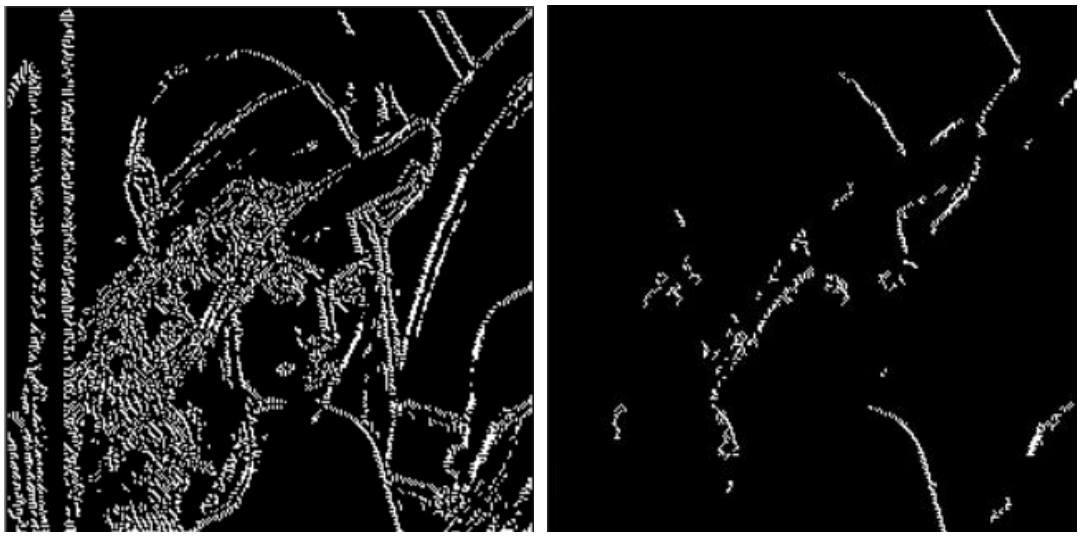


可见，低阈值调低能使图像连续，但会引入过多噪点。

然而，低阈值调高使得噪点消除，但会使图像不够连续。

3.2 高阈值的选择

在此部分中，我们保持低阈值为 30 不变，将高阈值设置为 40（左）与 200（右）的结果如图所示：



可见，高阈值设置过低会使得大量非边缘的像素点被误识别，设置过高会使得图像边缘很不完整。

3.3 实验结论

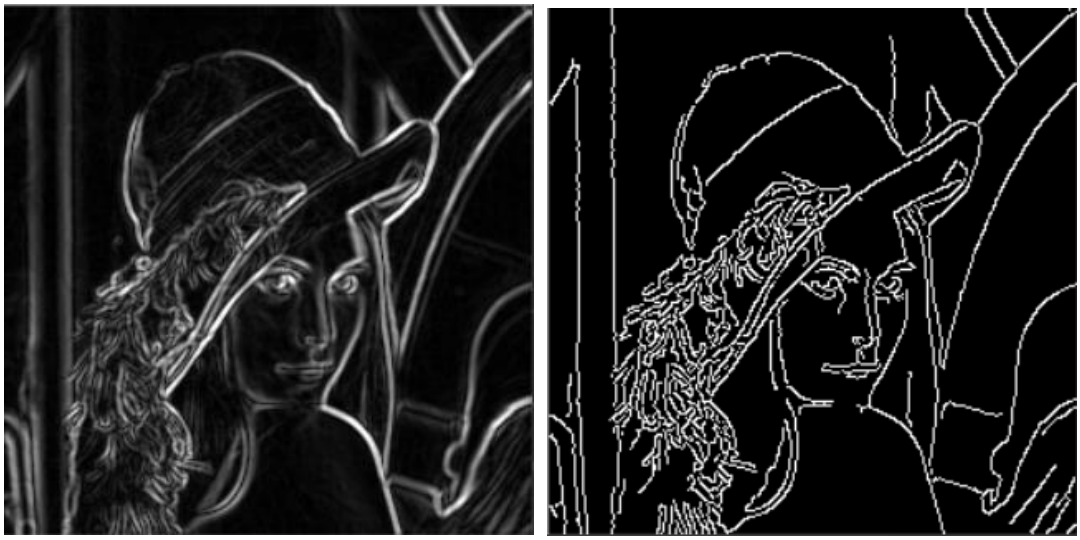
综上可见，低高阈值的设置都要合理恰当，一般两者的相对大小应该在 0.3 到 0.5 之间，绝对大小需要不断调整。

4、 有关算子选择的拓展

4.1 两种算子的选择与对比

这里我们对比我使用的 Sobel 算子（左）与 Canny 算法（右）自带的算子。

两者获得的边缘图对比如下：



结合之前的实验结果，我们得到可以得到如下的结论：

Sobel 算子得到的图像边缘有粗细，但是较模糊，识别不够精准。

Canny 算子得到的图像轮廓清晰，但是边缘没有粗细，没有深度感。

这两种算子，在实际应用中，具体的选择，我们要结合自身情况而定。

三、实验总结

1、实验概述

本次实验的主要任务，可以总结为自己实现 Canny 边缘检测的算法，并与 OpenCV 原算法相对比。

2、感想总结

在这次的实验中，学会了的东西有很多，其中最重要的就是提高了自己处理问题、收集相关资料、解决问题的能力，这在我们将来的学习和工作生活中都是很重要的。而具体细化开来，在本次实验中：

2.1 学会了自己动手设计一个 Canny 边缘检测算法

2.2 学会了如何使用 cv2 自带的函数来实现 Canny 边缘检测

3、创新点

在进行双阈值算法检测边缘时，我想到了使用了 th1 与 th2 将图像中的像素点分为了三个部分：低阈值（0）、中阈值（1）、高阈值（2）。在这三部分中，以高阈值形成的图像为基础，在图像中的边缘进行在中阈值的对周围点的搜索，并且将其补充进去作为新的边缘。完成操作后，将高阈值置为 255，输出白色的边缘。

这样实现的双阈值检测很好的利用了两个参数，并且达到了相对令人满意的结果。

4、遇到的问题

在使用矩阵除法时，矩阵中有元素为 0，会遇到如下的 Warning：

```
1.  RuntimeWarning: divide by zero encountered in floor_divide
2.  angle = np.floor_divide(absX, absY).astype(int)
```

我想到通过给 absY 中的每一个元素加上极小量 1e-8，这样既避免了上述问题，又不至于影响实验结果。