上机实验十一
实验报告

刘瀚文
517030910294

2018 年 12 月 4 日

目录

Part I

实验十一

Chapter 1

实验准备

1.1 环境配置

1.1.1 OpenCV

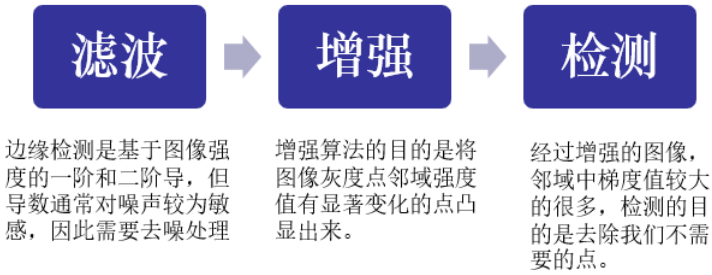
使用上次安装好的 OpenCV 库。
环境说明：Python 2.7.15 OpenCV 3.4.3
使用 Ubuntu 14.04 VMware 14

1.2 背景知识

1.2.1 边缘检测

图象的边缘指图象局部区域亮度变化显著的部分，该区域的灰度剖面一般可以看作是一个阶跃，即灰度值在很小的区域内急剧的变化。实现图像的边缘检测，主要用离散化梯度逼近函数根据二维灰度矩阵梯度向量来寻找图像灰度矩阵的灰度跃变位置，然后在图像中将这些位置的点连起来就构成了所谓的图像边缘。

1.2.2 概括性的检测步骤



1.2.3 Canny 算法原理

灰度化

通常摄像机获取的是彩色图像，而检测的首要步骤是进行灰度化，以 RGB 格式彩图为例，一般的灰度化方法有两种：

方式 1：Gray = (R + G + B)/3;

方式 2：Gray=0.299R+0.587G+0.114B;（参数考虑到人眼的生理特点）

高斯滤波

图像高斯滤波的实现可以用两个一维高斯核分别两次加权实现，也可以通过一个二维高斯核一次卷积实现。离散化的一维高斯函数与二维高斯函数如下：确定参数就可以得到一维核向量与二维核向量：

$$K = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x*x}{2\sigma*\sigma}}$$

离散一维高斯函数

$$K = \frac{1}{2\pi\sigma*\sigma} e^{-\frac{x*x+y*y}{2\sigma*\sigma}}$$

离散二维高斯函数

注：在求得高斯核后，要对整个核进行归一化处理

灰一阶偏导的有限差分来计算梯度的幅值和方向

由于我使用的是 Sobel 算子，在背景知识中就不阐述 sobel 算子，在这里阐明 Canny 算子。

Canny算法使用的算子

Canny算法中，所采用的卷积算子较为简单，为 $s_x=\begin{bmatrix}-1 & 1\\ -1 & 1\end{bmatrix}, s_y=\begin{bmatrix}1 & 1\\ -1 & -1\end{bmatrix}$

其x向、y向的一阶偏导数矩阵，梯度幅值以及梯度方向的数学表达式为：

$$P[i,j]=(f[i,j+1]-f[i,j]+f[i+1,j+1]-f[i+1,j])/2$$
$$Q[i,j]=(f[i,j]-f[i+1,j]+f[i,j+1]-f[i+1,j+1])/2$$
$$M[i,j]=\sqrt{P[i,j]^2+Q[i,j]^2}$$
$$\theta[i,j]=\arctan(Q[i,j]/p[i,j])$$

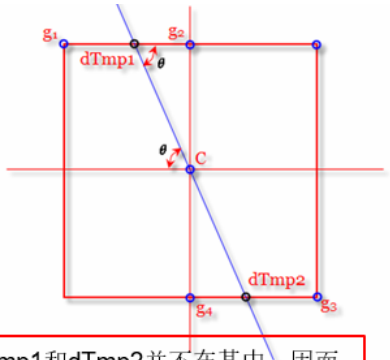
求出这几个矩阵后，就可以进行下一步的检测过程。

对梯度幅值进行非极大值抑制

图像梯度幅值矩阵中的元素值越大，说明图像中该点的梯度值越大。在 Canny 算法中，非极大值抑制是进行边缘检测的重要步骤，是指寻找像素点局部最大值，将非极大值点所对应的灰度值置为 0，从而可以剔除掉一大部分非边缘点。

右图为判断像素点C的灰度值在邻域内是否为最大的原理。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。



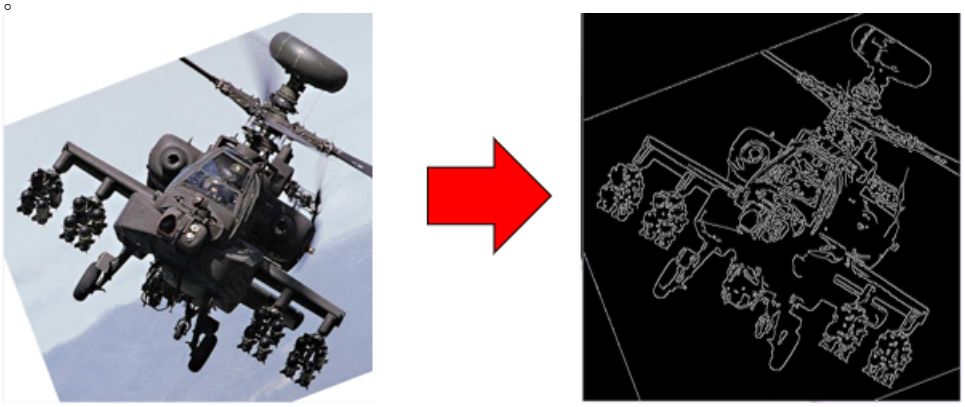
注意：我们只能得到C点邻域的8个点，而dTmp1和dTmp2并不在其中，因而需要根据g1和g2对dTmp1进行插值，根据g3和g4对dTmp2进行插值，这要用到其梯度方向，即上文Canny算法中要求解梯度方向矩阵的原因。

双阈值算法检测和连接边缘

Canny 算法中减少假边缘数量的方法是采用双阈值法。选择两个阈值，根据高阈值得到一个边缘图像，这样一个图像含有很少的假边缘，但是由于阈值较高，产生的图像边缘可能不闭合，为解决此问题采用了另外一个低阈值。

在高阈值图像中把边缘链接成轮廓，当到达轮廓的端点时，该算法会在

断点邻域点中寻找满足低阈值的点，再根据此点收集新的边缘，直到整个图像边缘闭合。



Chapter 2

Exercise

181204 更新在这部分之前，想向老师解释一下，由于我原本以为操作的使之前实验的两张照片，所以在第一次完成报告的时候就以之前两个图片作为基准，并以其中一张效果较好的图片详细展示了操作过程中每一步得到的图片结果。后来在提交时才发现 *dataset* 中给出的是新的图片，幸好我的代码都是自己写的，所以就用新的图片重新完成了实验，在报告最后的对比环节中加入 *dataset* 中图片的对比结果以示完成实验。但是中间过程还是保留原本的图片的展示。这个先跟老师解释一下，给老师添麻烦了。

2.1 要求

请对 dataset 文件夹中的图片进行 Canny 边缘检测，并与 OpenCV 库中自带 Canny 检测结果进行对比，报告中应包含实验原理，算法，流程，结果，最好有自己的心得体会和讨论，源程序附在报告最后。

可选取不同梯度幅值算子与阈值获得更优的检测性能。

2.2 实验过程

在本次实验中将在实验过程中结合代码及每一步骤得到的图片 (以 img2 为例)，阐明我的实验思路的同时完成实验。

2.2.1 对原始图像进行灰度化

```
# 1
img = cv2.imread("img2.png", 0)
# cv2.imshow("David Stark's Image", img)
```

2.2.2 对图像进行高斯滤波

直接使用 PPT 后面的高斯滤波函数，注意（9*9）的选择。

高斯滤波函数

OpenCV 中的高斯模糊函数 Gause_img= cv2.GaussianBlur(img,(3,3),0) 给定的参数是高斯矩阵的尺寸和标准差这里 (3, 3) 表示高斯矩阵的长与宽都是 3，标准差取 0 时 OpenCV 会根据高斯矩阵的尺寸自己计算。

```
# 2
Guass_img = cv2.GaussianBlur(img, (9, 9), 0)
cv2.imshow("David Stark's Guass_img", Guass_img)
```

使用 [9,9] 矩阵得到的高斯模糊图像

2.2.3 用一阶偏导的有限差分来计算梯度的幅值和方向

在实验中，我使用的是 Sobel 算子，Sobel 算子也可以应用 PPT 最后给出的公式进行操作。

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：

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

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

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

```
# 3
x = cv2.Sobel(Guass_img, cv2.CV_16S, 1, 0)
y = cv2.Sobel(Guass_img, cv2.CV_16S, 0, 1)

absX = cv2.convertScaleAbs(x)
absY = cv2.convertScaleAbs(y)

dst = cv2.addWeighted(absX, 0.5, absY, 0.5, 0)
```

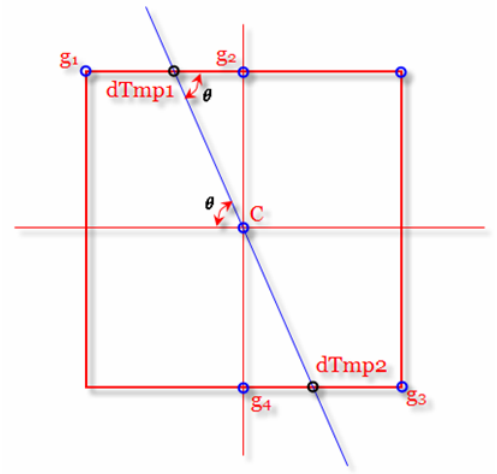
水平方向应用 Sobel 算子得到的 absX

竖直方向应用 Sobel 算子得到的 absY

综合 x-y 方向得到的 dst

2.2.4 对梯度幅值进行非极大值抑制

图像梯度幅值矩阵中的元素值越大，说明图像中该点的梯度值越大，但这不能说明该点就是边缘。非极大值抑制就很关键了，寻找像素点局部最大值，将非极大值点所对应的灰度值置为 0，这样剔除掉一大部分非边缘的点。



根据示意图可知，要进行非极大值抑制，就首先要确定像素点 C 的灰度值在其 8 值邻域内是否为最大。示意图中蓝色的线条方向为 C 点的梯度方向，这样就可以确定其局部的最大值肯定分布在这条线上，也表示除了 C 点外，梯度方向的交点 dTmp1 和 dTmp2 这两个点的值也可能会是局部最大值。因此，判断 C 点灰度与这两个点灰度大小即可判断 C 点是否为其邻域内的局部最大灰度点。

如果经过判断，C 点灰度值小于这两个点中的任一个，那就说明 C 点不是局部极大值，那么则可以排除 C 点为边缘。这就是非极大值抑制的工作原理。

```
# 4
absX = absX + 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

# cv2.imshow("David Stark's New dst_img", dst)
```

对梯度幅值进行非极大值抑制得到的 newDst

这部分的操作中由于 numpy 提供了很方便的矩阵除法运算，所以直接使用 numpy 中的地板除进行计算，同时将结果输出为 int 形式。这样尽可能保证精度的同时，也能提高运算的效率。同时注意到在过程中，梯度值有为 0 的现象，所以在原有梯度值的基础上加上一个 1e-8 的小量，方便计算。

剩下的操作就是对于这部分操作的思路的实现，实质就是比较中心点和加权获得的 dTmp1，dTmp2 的大小，进行非极大值抑制。按照思路，进行四种 k 值的判断就可以得到正确的结果。

发现在理解上需要注意的两点

非最大抑制是回答这样一个问题 “当前的梯度值在梯度方向上是一个局部最大值吗?”所以, 要把当前位置的梯度值与梯度方向上两侧的梯度值进行比较。

梯度方向垂直于边缘方向 但实际上，我们只能得到 C 点邻域的 8 个点的值，而 dTmp1 和 dTmp2 并不在其中，要得到这两个值就需要对该两个点两端的已知灰度进行线性插值，也即根据图 1 中的 g1 和 g2 对 dTmp1

进行插值，根据 g3 和 g4 对 dTmp2 进行插值，这要用到其梯度方向，这是上文算法中求解梯度方向矩阵的原因。

完成非极大值抑制后 会得到一个二值图像，非边缘的点灰度值均为 0，可能为边缘的局部灰度极大值点可设置其灰度为 128。根据下文的具体测试图像可以看出，这样一个检测结果还是包含了很多由噪声及其他原因造成的假边缘。因此还需要进一步的处理，就是第五步中的用双阈值算法检测和连接边缘。

2.2.5 用双阈值算法检测和连接边缘

Canny 算法中减少假边缘数量的方法是采用双阈值法。选择两个阈值（高阈值和低阈值进行划分），根据高阈值得到一个边缘图像，这样一个图像含有很少的假边缘，但是由于阈值较高，产生的图像边缘可能不闭合，为解决这样一个问题采用了另外一个低阈值。

在高阈值图像中把边缘链接成轮廓，当到达轮廓的端点时，该算法会在断点的 8 邻域点中寻找满足低阈值的点，再根据此点收集新的边缘，直到整个图像边缘闭合。

这一步我觉得是边缘探测中最有意思的一部分 我思考了好多天如何写出并优化我的代码，我书写了两个版本的代码，最后选择用了第二个版本的代码。

我的代码的思路就是用 TH1 & TH2 将图像分为三个部分，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，并且补充进去作为新的边缘，（代码的目标）实现图像边缘的闭合。再循环完成操作的时候，将高阈值 (形成的边缘) 置为 255，输出白色的边缘。

```
# 5
th1 = 32
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("David Stark's Data", dst)
```

经过我的代码得到的边缘识别图像

在这部分中需要对 TH1 & TH2 的大小进行调整以产生最有效果的边界图像。经过我的尝试，一般选取 TH1 / TH2 = 0.3 - 0.5 的区间比较好。

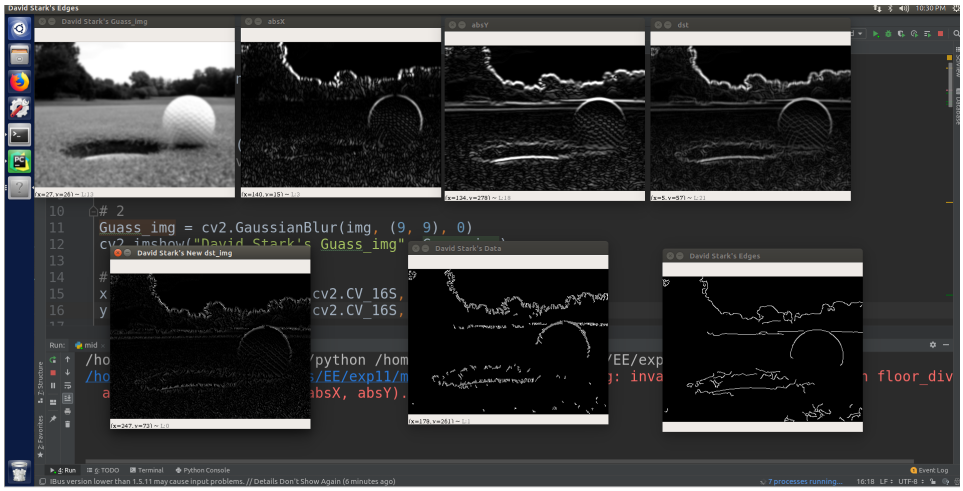
2.2.6 最后和 Canny 算法直接进行比较

使用 OpenCV 的 Canny 函数进行输出。

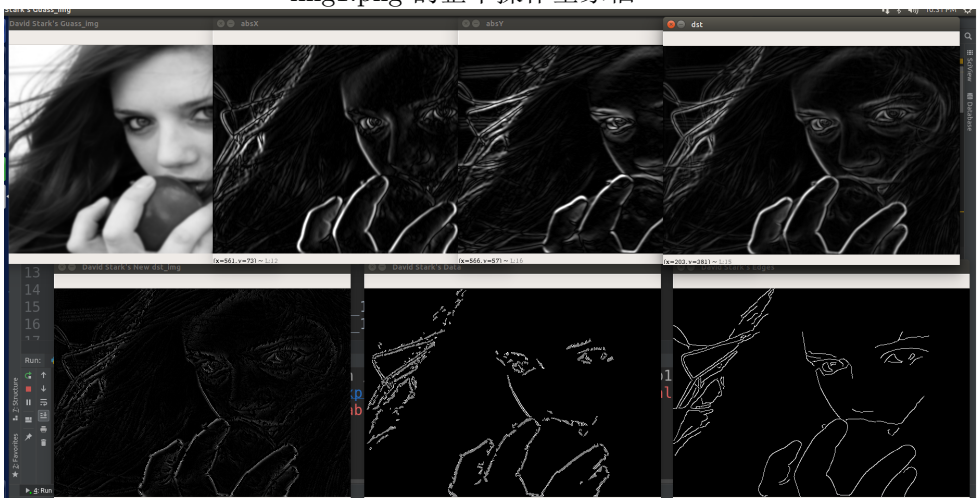
```
edges = cv2.Canny(Guass_img, 50, 150, apertureSize=3)
```

2.3 结果展示

由于之前已经在实验步骤中展示了尽可能好的边缘识别的效果，为了节约报告的空间，在结果展示部分将展示大的全家福，以展示效果及完成情况。



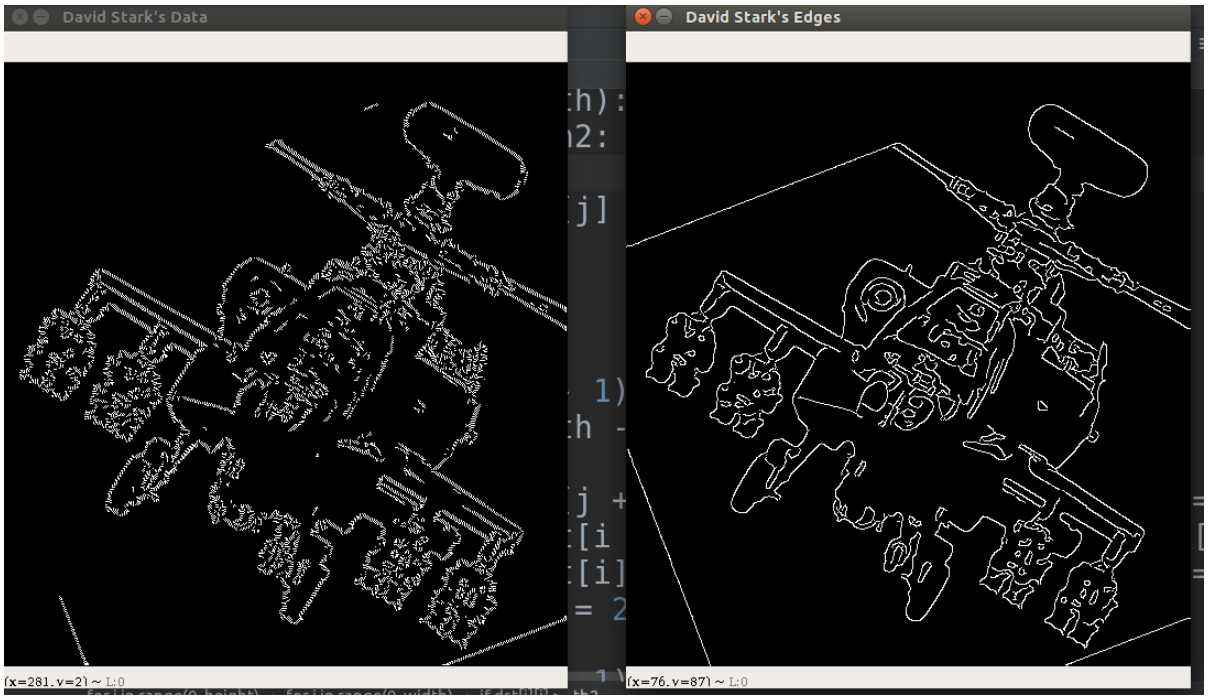
img1.png 的整个操作全家福



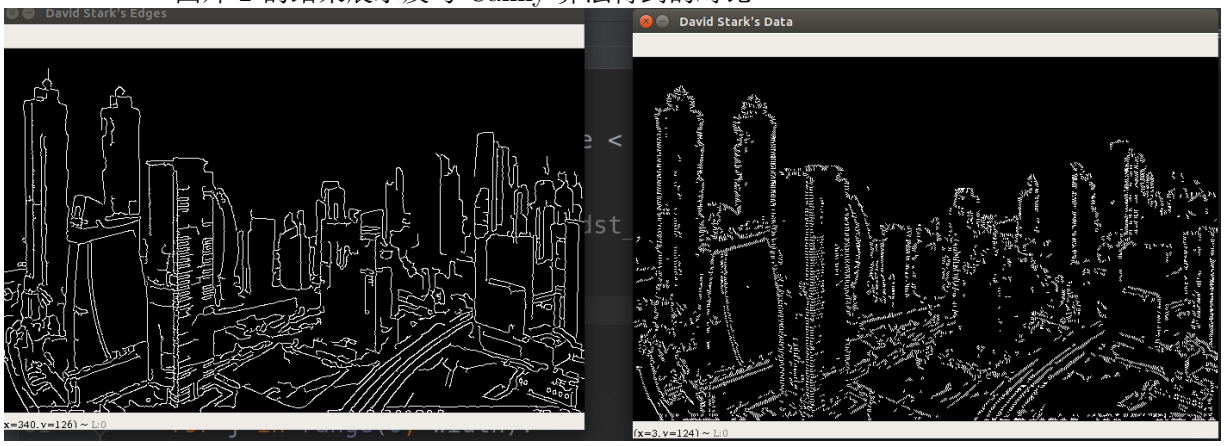
img2.png 的整个操作全家福

181204 更新在这里展示 dataset 的结果

图片 1 的结果展示及与 Canny 算法得到的对比



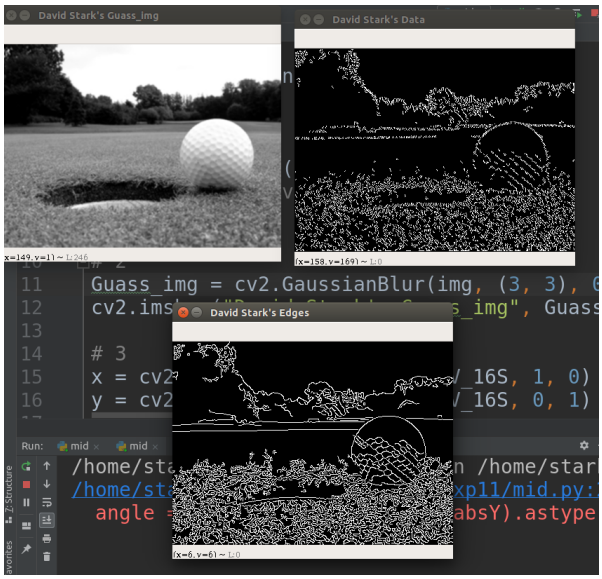
图片 2 的结果展示及与 Canny 算法得到的对比



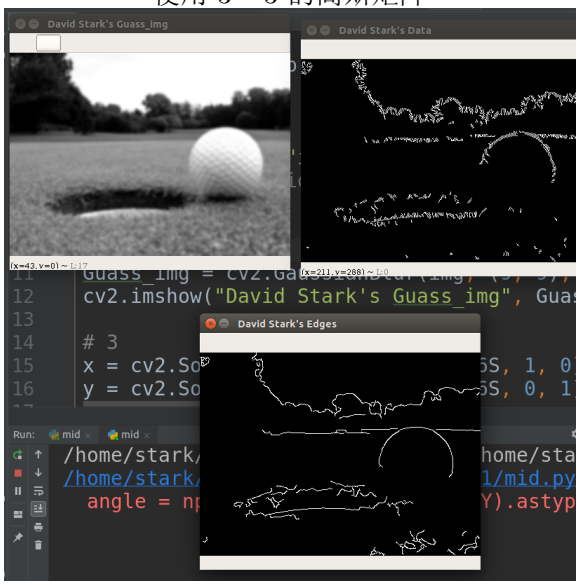
图片 3 的结果展示及与 Canny 算法得到的对比

2.4 初步实验完成后对实验的分析、优化与提高

2.4.1 高斯模糊矩阵大小的选择的对比



使用 3 * 3 的高斯矩阵



使用 9 * 9 的高斯矩阵

对图像进行高斯滤波，听起来很玄乎，其实就是根据待滤波的像素点及其邻域点的灰度值按照一定的参数规则进行加权平均。这样可以有效滤去理想图像中叠加的高频噪声。

通常滤波和边缘检测是矛盾的概念，抑制了噪声会使得图像边缘模糊，这会增加边缘定位的不确定性；而如果要提高边缘检测的灵敏度，同时对噪声也提高了灵敏度。

实际工程经验表明，高斯函数确定的核可以在抗噪声干扰和边缘检测精确定位之间提供较好的折衷方案。

在本次实验中，明显选择较大的高斯矩阵会有更好的效果。

2.4.2 高斯滤波函数的自己实现

```
#生成二维高斯分布矩阵
gaussian = np.zeros([5, 5])
for i in range(5):
  for j in range(5):
    gaussian[i,j] = math.exp(-1/2 * (np.square(i-3)/np.square(sigma1) + (np.
                                     square(j-3)/np.square(sigma2)))) / (2*
                                     math.pi*sigma1*sigma2)
    sum = sum + gaussian[i, j]

gaussian = gaussian/sum
# print(gaussian)

def rgb2gray(rgb):
  return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])


# 高斯滤波
gray = rgb2gray(img)
W, H = gray.shape
new_gray = np.zeros([W-5, H-5])
for i in range(W-5):
for j in range(H-5):
  new_gray[i,j] = np.sum(gray[i:i+5,j:j+5]*gaussian)# 与高斯矩阵卷积实现滤波

# plt.imshow(new_gray, cmap="gray")
```

2.4.3 使用不同梯度幅值算子

使用最基本的梯度的方法来计算

```
W1, H1 = new_gray.shape
dx = np.zeros([W1-1, H1-1])
dy = np.zeros([W1-1, H1-1])
d = np.zeros([W1-1, H1-1])

for i in range(W1-1):
  for j in range(H1-1):
    dx[i,j] = new_gray[i, j+1] - new_gray[i, j]
    dy[i,j] = new_gray[i+1, j] - new_gray[i, j]
    d[i, j] = np.sqrt(np.square(dx[i,j]) + np.square(dy[i,j]))
    # 图像梯度幅值作为图像强度值

# plt.imshow(d, cmap="gray")
```

使用算子

由于使用不同的梯度算子进行操作，代码的实质差不多，就不再呈现代码。将侧重点放在不同算子的学习和比较上。

为什么我们要用算子来进行梯度的计算 因为边缘的实质就是图像上灰度级变化很快的点的集合。

那我们如何计算出这些变化率很快的点 自然而然我们会想到用数学上的导数：连续函数上某点斜率，导数越大表示变化率越大，变化率越大的地方就越是“边缘”，但是在计算机中不常用，因为在斜率 90 度的地方，导数无穷大，计算机很难表示这些无穷大的东西。

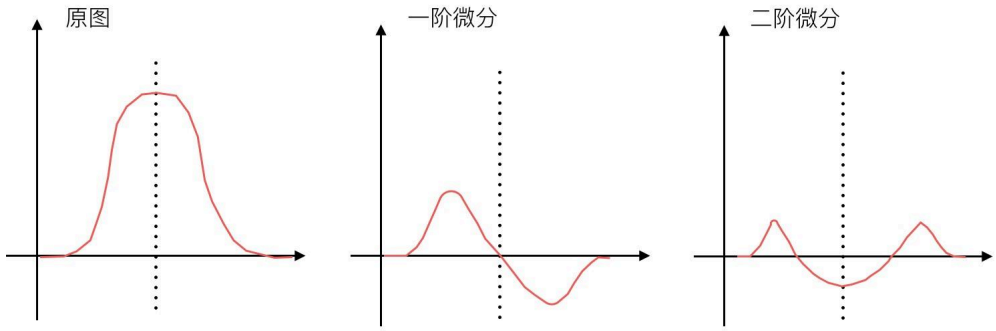
那我们就得使用方便计算机的方法来操作，使用微分。连续函数上 x 变化了 dx，导致 y 变化了 dy，dy 值越大表示变化的越大，那么计算整幅图像的微分，dy 的大小就是边缘的强弱了。微分与导数的关系：dy = f ’(x) dx

在这种思路的大方向上，我们产生了三种算子——Prewitt 算子、Sobel 和 Laplace 算子。

Prewitt 算子 Prewitt 算子就是在二维情况下对于一阶差分的计算。左右对称形成 [-1, 0, 1] (一维) 的矩阵。

Sobel 算子　Sobel 算子是在 Prewitt 的基础上进行对中心点的重点考虑。对中心点进行加权之后形成了 Sobel 算子。分别计算偏 x 方向的 Gx，偏 y 方向的 Gy，然后通过计算得到的 G(x, y)，就是 sobel 边缘检测后的图像了。

Laplace 算子　拉普拉斯是用二阶差分计算边缘的，在连续函数的情况下在一阶微分图中极大值或极小值处，认为是边缘。在二阶微分图中极大值和极小值之间的过 0 点，被认为是边缘。



拉普拉斯算子推导：

一阶差分：f ’(x) = f(x) - f(x - 1)
二阶差分：f ’(x) = (f(x + 1) - f(x)) - (f(x) - f(x - 1))
化简后：f ’(x) = f(x - 1) - 2 f(x)) + f(x + 1)
提取前面的系数：[1, -2, 1]

二维的情况下，同理可得:

$$f'(x, y) = -4\,f(x, y) + f(x-1, y) + f(x+1, y) + f(x, y-1) + f(x, y+1)$$

并且考虑斜对角的情况得到普拉斯算子。

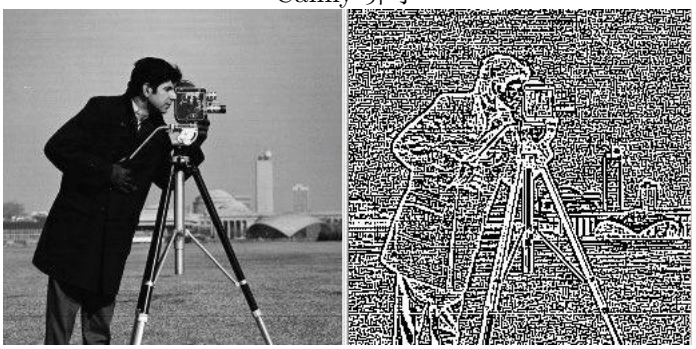
$$\begin{matrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{matrix}$$

三个算子区别　sobel 产生的边缘有强弱，抗噪性好；
laplace 对边缘敏感，可能有些是噪声的边缘，也被算进来了；
canny 产生的边缘很细，可能就一个像素那么细，没有强弱之分。

三种算子的对比图：

Sobel 算子

Canny 算子



Laplace 算子

2.4.4 双阈值算法的高低阈值选择

两个阈值（高阈值、低阈值）的选择会影响图片的边缘细节。

在这里只阐述高低阈值对于细节的影响，而不展示图片啦。

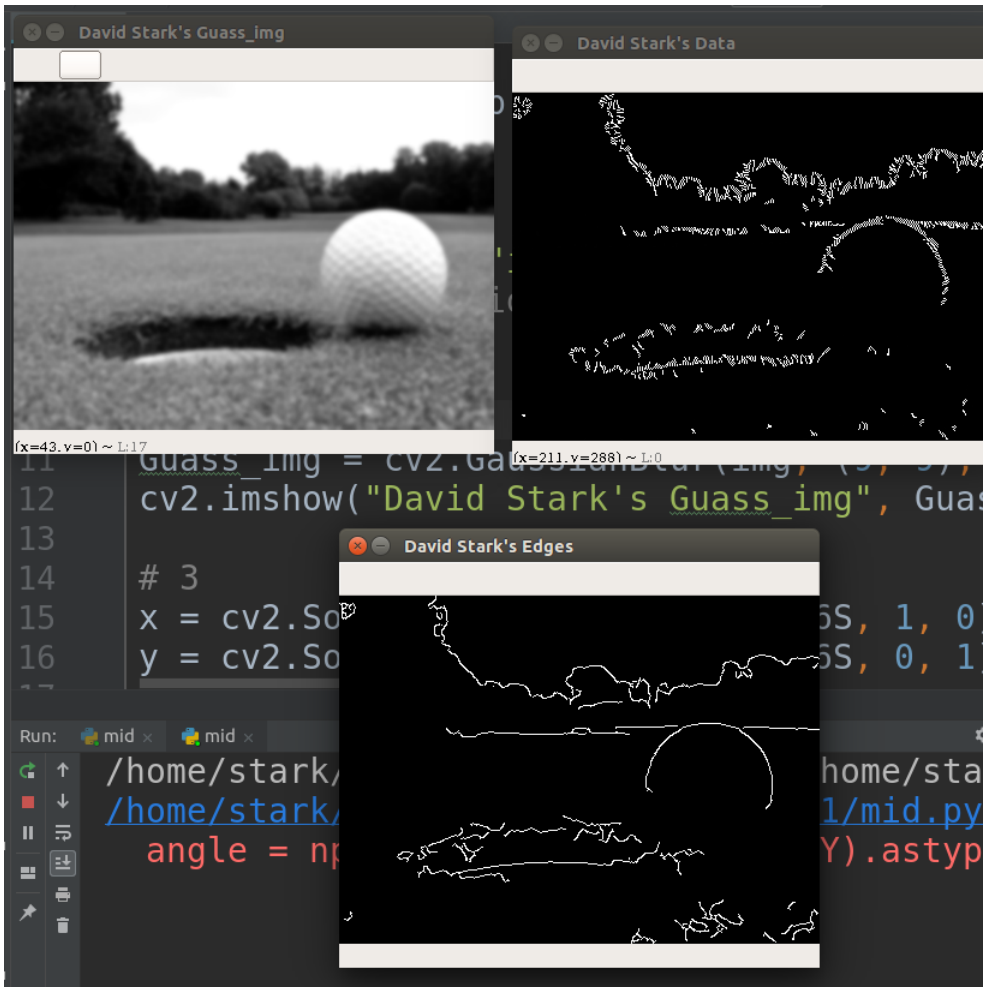
高阈值限制了在连接边缘前的细节，高阈值越高就会留存更少的边缘点（正确率肯定会更高）但是就会限制边缘的条数，影响完成后图片的边缘。若是高阈值越低，就会留存更土的图片上的点，那么随之而来的就是正确率的降低（就是假边缘的条数上升）

低阈值则是起到一个过滤噪声的作用。低阈值越高，背景中的噪声留存的越少，但是可用的信息也相对会减少。而选择一个低的低阈值，有可能会引入太多的背景噪声，使图像的边缘不明显。

一般的选择在实验操作中也阐明了，相对值关系 TH1 / TH2 = 0.3 - 0.5，而绝对大小的关系就是在操作中自行进行调节。在实验的最后我在总结的过程中，又发现了一种方法，可以选取高低阈值为原来的（即前一步中）最大值的一个比例，这样可以同时考虑到原图像的信息，而进行更有效的判断。

2.4.5 我做出的结果与 Canny 直接做出图形的比较及思考

img1



img1 中的结果对比

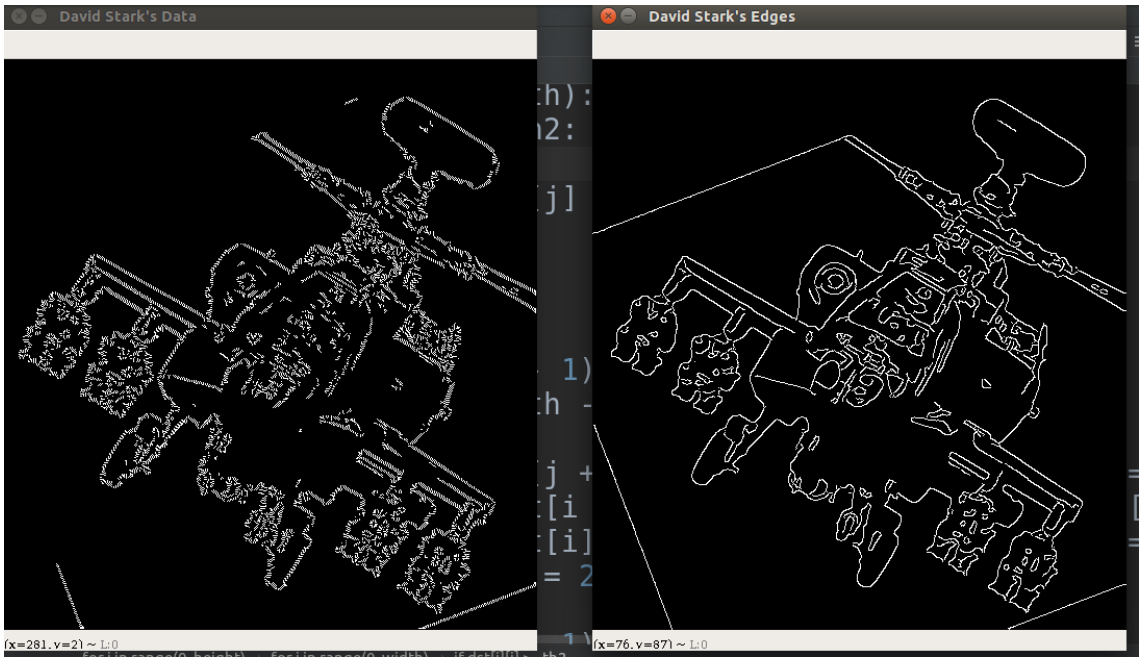
img2

我自己代码的结果图像

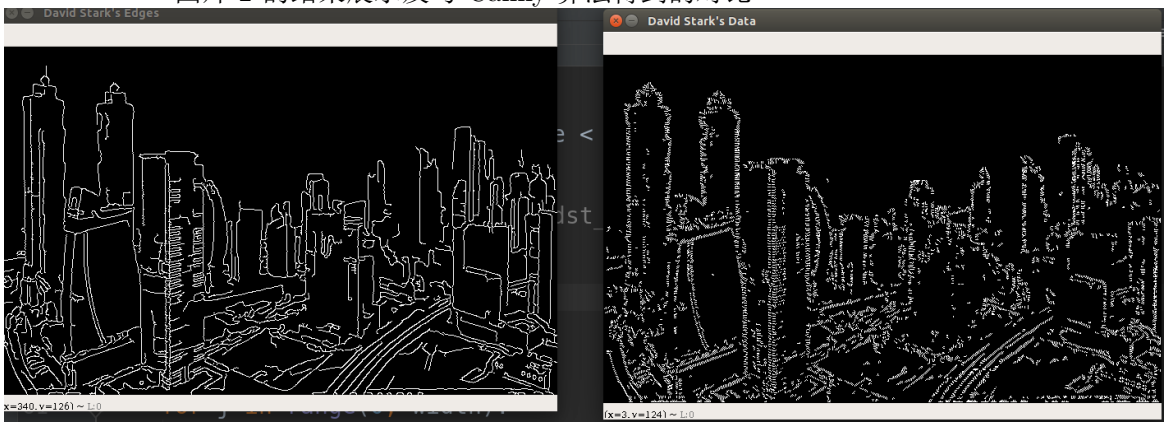
Canny 得到的图像

181204 更新在这里加入和上面相同的比较图片

图片 1 的结果展示及与 Canny 算法得到的对比



图片 2 的结果展示及与 Canny 算法得到的对比



图片 3 的结果展示及与 Canny 算法得到的对比

思考产生的不同结果　也是在前面的算子比较中进行了阐释:

Sobel 产生的边缘有强弱，抗噪性好；

Canny 产生的边缘很细，可能就一个像素那么细，没有强弱之分。

我的操作　使用的是 Sobel 算子，可能是算子的因素导致的差异。

2.5 源码

```
import cv2
import numpy as np
import copy

# 1
img = cv2.imread("img1.png", 0)
# cv2.imshow("David Stark's Image", img)


# 2
Guass_img = cv2.GaussianBlur(img, (3, 3), 0)
cv2.imshow("David Stark's Guass_img", Guass_img)

# 3
x = cv2.Sobel(Guass_img, cv2.CV_16S, 1, 0)
y = cv2.Sobel(Guass_img, cv2.CV_16S, 0, 1)

absX = cv2.convertScaleAbs(x)
absY = cv2.convertScaleAbs(y)

dst = cv2.addWeighted(absX, 0.5, absY, 0.5, 0)

# cv2.imshow("absX", absX)
# cv2.imshow("absY", absY)
# cv2.imshow("dst", dst)

# 4
absX = absX + 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

# cv2.imshow("David Stark's New dst_img", dst)

# 5
th1 = 32
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("David Stark's Data", dst)

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

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

Part II

实验总结

这次实验深入学习了 Canny 算法，并且用自己的代码水平进行了重构，很有趣！

在这次实验中，由于大部分都是自己的代码，所以对以自己的代码水平也是有了一次提高。对于操作的思路，进行代码思路的转化，进行代码的实现。很锻炼人的能力，也激起了我对视觉处理的兴趣。

期待在下一次实验中学习更多的知识！