1. canny 算法介绍

虽然 Canny 算法年代久远，但可以说它是边缘检测的一种标准算法，而且仍在研究中广泛使用。Canny 算法基于三个基本目标：

① 低错误率：所有的边缘都应该被找到，而且应该没有伪响应。也就是检测到的边缘必须是尽可能真实的边缘。

② 边缘点应该能被很好地定位。已定位边缘必须尽可能接近真实边缘。也就是有检测器标记为边缘的点和真实边缘的中心之间的距离应该最小。

③ 单一的边缘点响应。对于真实的边缘点。检测器仅应返回一个点。也就是真实边缘周围的局部最大数应该是最小的。这意味着仅存一个单一边缘点的位置，检测器不应该指出多个边缘像素。

Canny 算法工作的本质是从数学上表达了前面三个准则，并试图找到这些表达式的最优解。

2. Canny 算法的分析与设计

Canny 检测算法包含下面几个阶段：

1) 灰度化

灰度计算公式：（有多种算法，这里选取下式所示的一组 RGB 系数）

$$Gray[i, j] = 0.299 * R[i, j] + 0.587 * G[i, j] + 0.114 * B[i, j]$$

2) 高斯模糊

在实际的图片中，都会包含噪声。但有时候，图片中的噪声会导致图片中边缘信息的消失。对此的解决方案就是使用高斯平滑来减少噪声，即进行高斯模糊操作。该操作是一种滤波操作，与高斯分布有关，下面是一个二维的高斯函数，其中 (x, y) 为坐标，σ 为标准差：

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

进行高斯滤波之前，需要先得到一个高斯滤波器（kernel）。如何得到一个高斯滤波器？其实就是将高斯函数离散化，将滤波器中对应的横纵坐标索引代入高斯函数，即可得到对应的值。不同尺寸的滤波器，得到的值也不同，下面是 (2k+1)x(2k+1) 滤波器的计算公式 ：

$$H[i, j] = \frac{1}{2\Pi\sigma^2} e^{-\frac{(i-1-k)^2 + (j-k-1)^2}{2\sigma^2}}$$

在本次实验中，我使用的是 5×5 高斯滤波器，即：H[i,j]公式中的 k 取 2。

3) 计算图片梯度幅值

边缘是图像强度快速变化的地方，可以通过图像梯度幅值，即计算图像强度的一阶导数来识别这些地方。由于图片数据的数值是离散的，可以使用离散形式的导数来近似图片的梯度：

$$\frac{\partial \mathrm{f}}{\partial x}=\frac{f\left(x_{n+1}, y\right)-f\left(x_{n}, y\right)}{\Delta \mathrm{x}}$$

图片梯度幅值为：

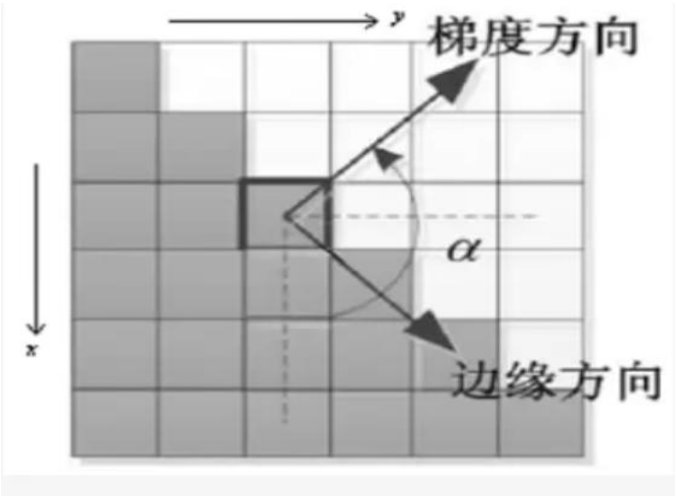
$$\mathrm{M}=\sqrt{(\frac{\partial f}{\partial x})^{2}+(\frac{\partial f}{\partial y})^{2}}$$

梯度方向为：

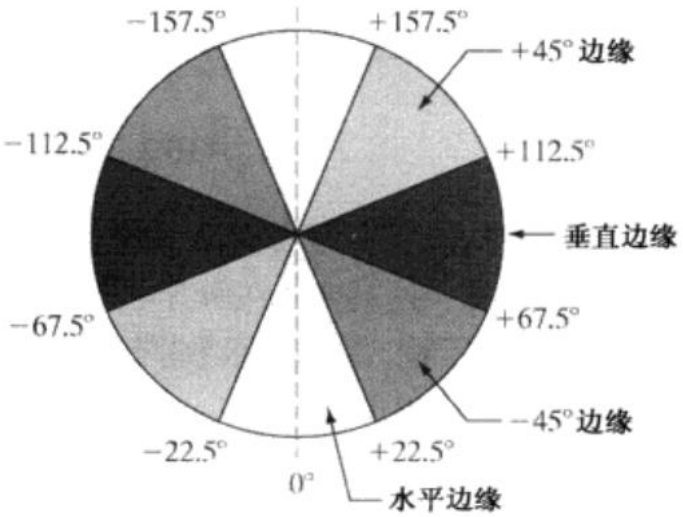
$$\Theta=\arctan^{-1}(\frac{\partial f}{\partial y} / \frac{\partial f}{\partial x})$$

4) 非极大值抑制

理想情况下，最终得到的边缘应该是很细的。因此，需要执行非极大值抑制以使边缘变细。原理很简单：遍历梯度矩阵上的所有点，并保留边缘方向上具有极大值的像素。该方法的本质是：指定边缘法线的许多离散方向（梯度向量）。



实际做法：定义四个方向，分别是水平、+45 度、垂直、-45 度。实际图像在某处的边缘方向是有无数种可能的，但我们必须把所有可能的边缘方向量化为上述的 4 个方向，故我们必须定义一个方向范围（包含 4 个等大小的子范围，每个子范围对应上述 4 个方向的一个），当边缘法线落在某个子范围就认为边缘法线的方向是该子范围对应的方向。



令 d1、d2、d3、d4 表示四个基本边缘方向：水平、+45 度、垂直、-45 度。对于以点（x，y）为中心的 3×3 邻域，假设，给出非抑制方案如下：

① 寻找点（x，y）的梯度方向最接近的基本方向 d_k；

② 只要点（x，y）的梯度幅度 M（x，y）小于沿着方向 d_k 的两个邻点中任意一个（因为是 3×3 邻域，所以有 2 个邻点）的梯度幅度，就令 $g_N(x，y)=0$（抑制）；否则令 $g_N(x，y)=M（x，y）$，这里 $g_N(x，y)$ 是非抑制后的图像。

5) 双阈值处理

阈值操作的操作对象是非抑制后的图像 $g_N(x，y)$，目的是减少伪边缘点。使用单阈值法对于其中低于该阈值的所有 $g_N(x，y)$ 值置 0，存在一个问题：如果阈值设置过低，则仍会存在一些伪边缘。如果阈值设置过高，则会删除实际上有效的边缘点。Canny 算法中，我们通过使用滞后阈值可以在一定程度上改善这一状况。两个阈值：一个低阈值 T_L 和一个高阈值 T_H。

非抑制后的图像 $g_N(x，y)$ 中大于 T_H 的任何边缘是真边缘，**保留**；而 T_L 以下的边缘肯定是非边缘，**舍弃**。位于这两个阈值之间的边缘会基于其连通性而分类

为边缘或非边缘，如果它们连接到“可靠边缘”像素（即：已被保留的像素），则它们被视为边缘的一部分。否则，不属于边缘，需要舍弃。

3. 测试示例

测试 1

图 1-1 原图

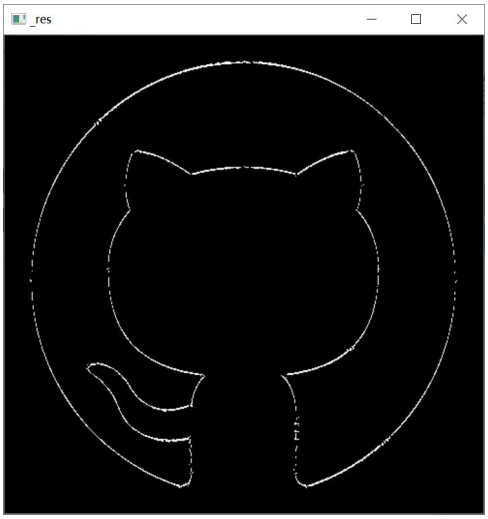


图 1-2 自实现的 Canny 算法效果

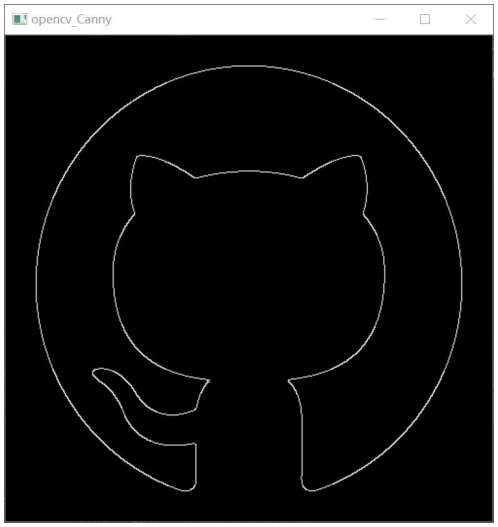


图 1-3 opencv 的 Canny 算法效果

测试 2

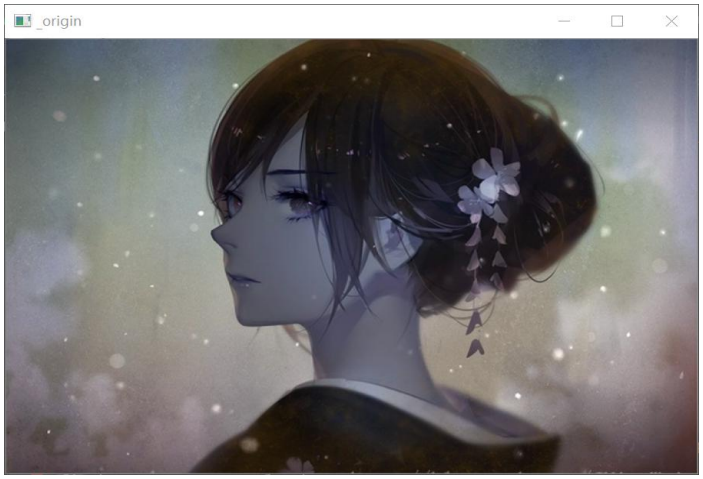


图 2-1 原图

图 2-2 自实现的 OTSU 算法结果

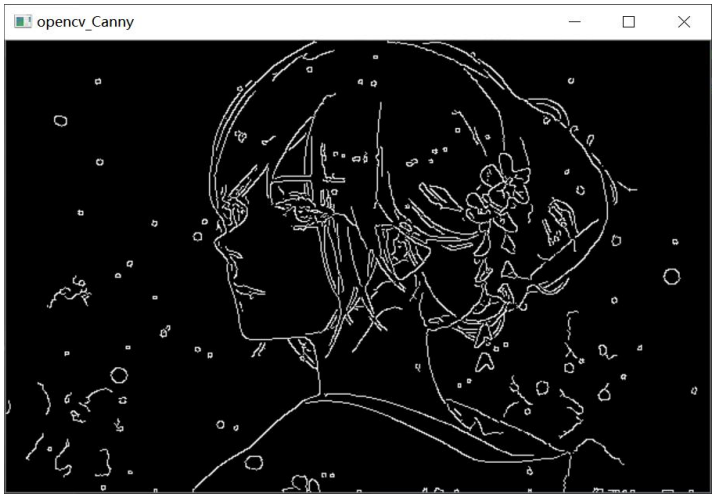


图 2-3 opencv 的 OTSU 算法

4. 对比分析

首先，分析测试 1：测试 1 的测试原图比较简单，不存在较多细节。对比图 1-1 和图 1-2，可以发现，对于简单的图形，自实现的 Canny 算法的效果还是不错的，虽然相比 opencv 自带的 Canny 算法，自实现的算法提取长边缘还存在一些缝隙，但提取出来的边缘效果大体上还是清晰的。

其次，分析测试 2：测试 2 的测试原图存在较多细节。对比图 2-1 和图 2-2，可以发现，自实现的 Canny 算法对长边缘的检测存在较多缝隙，导致最终得到的边缘图像看起来不够连续、不够丝滑，反观 opencv 自带的 Canny 算法，在长边缘的检测上表现得就很好。

总而言之，自实现的 Canny 算法对简单图像的边缘检测效果还是不错的，但是对于复杂图像的检测效果还有待改善。

对比自实现的 Canny 和 opencv 的 Canny 源码，我发现这两个算法在梯度计算上有所差异，opencv 在梯度计算上使用的是 Sobel 算子，而自实现的算法使用的则是普通的梯度计算公式。因此，梯度算法的不同和一些细节方面的不同处理造成了两个算法对图像处理的效果差异。

5. 源代码

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


# 函数：灰度化
def gray(img_url):
    """
    计算公式：
    Gray(i,j) = [R(i,j) + G(i,j) + B(i,j)] / 3
    or :
    Gray(i,j) = 0.299 * R(i,j) + 0.587 * G(i,j) + 0.114 * B(i,j)
    """
    # 读取图片资源
    img = plt.imread(img_url)
    # BGR 转换成 RGB 格式
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    # 灰度化
    gray_img = np.dot(img_rgb[..., :3], [0.299, 0.587, 0.114])
    # 返回灰度化后的图像
    return gray_img
```

```
# 函数：去除噪音 - 使用 5x5 的高斯滤波器
def smooth(gray_img):
    """
    要生成一个 (2k+1)x(2k+1) 的高斯滤波器：
    滤波器的各个元素计算公式如下：
    H[i, j] = (1/(2*pi*sigma**2))*exp(-1/2*sigma**2((i-k-1)**2 + (j-k-1)**2))
    """
    sigma1 = sigma2 = 1.4
    gau_sum = 0
    gaussian = np.zeros([5, 5])
    for i in range(5):
        for j in range(5):
            gaussian[i, j] = math.exp((-1 / (2 * sigma1 * sigma2)) * (np.square(i - 3)
                                                                      +
np.square(j - 3))) / (
                                                2 * math.pi * sigma1 * sigma2)
            gau_sum = gau_sum + gaussian[i, j]

    # 归一化处理
    gaussian = gaussian / gau_sum

    # 高斯滤波
    width, height = gray_img.shape
    gray_after_guasss = np.zeros([width - 5, height - 5])
    for i in range(width - 5):
        for j in range(height - 5):
            gray_after_guasss[i, j] = np.sum(gray_img[i:i + 5, j:j + 5] * gaussian)

    # 返回高斯滤波之后的图像
    return gray_after_guasss


# 函数：计算梯度幅值
def gradients(gray_smooth):
    width, height = gray_smooth.shape
    dx = np.zeros([width - 1, height - 1])
    dy = np.zeros([width - 1, height - 1])
    m = np.zeros([width - 1, height - 1])    # 梯度幅度
    theta = np.zeros([width - 1, height - 1])

    for i in range(width - 1):
        for j in range(height - 1):
            dx[i, j] = gray_smooth[i + 1, j] - gray_smooth[i, j]
```

```
            dy[i, j] = gray_smooth[i, j + 1] - gray_smooth[i, j]
            # 图像梯度幅作为图像强度
            m[i, j] = np.sqrt(np.square(dx[i, j]) + np.square(dy[i, j]))
            # 计算梯度方向 artan(dx/dy)
            theta[i, j] = math.atan(dx[i, j] / (dy[i, j] + 0.000000001))

    #  返回梯度值
    return dx, dy, m, theta


# 函数：非极大抑制
def do_nms(m, dx, dy):
    d = np.copy(m)
    width, height = m.shape
    nms = np.copy(d)
    nms[0, :] = nms[width - 1, :] = nms[:, 0] = nms[:, height - 1] = 0

    for i in range(1, width - 1):
        for j in range(1, height - 1):

            # 如果当前梯度为 0，该点就不是边缘点
            if m[i, j] == 0:
                nms[i, j] = 0

            else:
                grad_x = dx[i, j]  # 当前点 x 偏导
                grad_y = dy[i, j]  # 当前点 y 偏导
                grad_temp = d[i, j]  # 当前点 梯度

                # 导数方向趋向于 y 分量（如果 y 方向梯度值比较大，显然偏向 y 分量）
                if np.abs(grad_y) > np.abs(grad_x):
                    weight = np.abs(grad_x) / np.abs(grad_y)  # 权重
                    grad2 = d[i - 1, j]
                    grad4 = d[i + 1, j]

                    # 如果 x, y 方向导数符号一致
                    # 像素点位置关系
                    # g1 g2
                    #       c
                    #       g4 g3
                    if grad_x * grad_y > 0:
                        grad1 = d[i - 1, j - 1]
                        grad3 = d[i + 1, j + 1]
```

```
        # 如果 x，y 方向导数符号相反
        # 像素点位置关系
        #       g2 g1
        #       c
        # g3 g4
        else:
            grad1 = d[i - 1, j + 1]
            grad3 = d[i + 1, j - 1]

    # 如果 x 方向梯度值比较大
    else:
        weight = np.abs(grad_y) / np.abs(grad_x)
        grad2 = d[i, j - 1]
        grad4 = d[i, j + 1]

        # 如果 x, y 方向导数符号一致
        # 像素点位置关系
        #          g3
        # g2 c g4
        # g1
        if grad_x * grad_y > 0:

            grad1 = d[i + 1, j - 1]
            grad3 = d[i - 1, j + 1]

        # 如果 x，y 方向导数符号相反
        # 像素点位置关系
        # g1
        # g2 c g4
        #          g3
        else:
            grad1 = d[i - 1, j - 1]
            grad3 = d[i + 1, j + 1]

    # 利用 grad1-grad4 对梯度进行插值
    grad_temp1 = weight * grad1 + (1 - weight) * grad2
    grad_temp2 = weight * grad3 + (1 - weight) * grad4

    # 当前像素的梯度幅度是局部最大，则认为是边缘点
    if grad_temp >= grad_temp1 and grad_temp >= grad_temp2:
        nms[i, j] = grad_temp

    else:
        # 否则，认为不可能是边缘点，置 0（经过后面的阈值处理，这一部
```

```
分相当于被舍弃了）
                        nms[i, j] = 0
    #   返回非极大抑制后的结果
    return nms


#   函数：双阈值选取
def double_threshold(nms_res):
    weight, height = nms_res.shape
    res = np.zeros([weight, height])

    # 定义低高阈值（tl 和 th 一般 1:2 或 1:3）
    tl = 0.1 * np.max(nms_res)
    th = 0.3 * np.max(nms_res)

    # 利用双阈值进行处理
    for i in range(1, weight - 1):
        for j in range(1, height - 1):
            # 双阈值选取
            # 过小舍弃
            if nms_res[i, j] < tl:
                res[i, j] = 0
            # 过大保留
            elif nms_res[i, j] > th:
                res[i, j] = 1

            # 处理中间：连接到可靠边缘 ，则认为属于边缘
            elif (nms_res[i - 1, j - 1:j + 1] < th).any() \
                    or (nms_res[i + 1, j - 1:j + 1].any()
                        or (nms_res[i, [j - 1, j + 1]] < th).any()):
                res[i, j] = 1

    # 返回双阈值处理之后的图像
    return res


# 测试代码：测试自定义 canny 算法
# 1. 灰度化
_gray = gray(img_url='cannyTest1.jpg')
# 2. 高斯滤波
_smooth = smooth(gray_img=_gray)
# 3. 计算梯度
_dx, _dy, _M, _theta = gradients(gray_smooth=_smooth)
# 4. 非极大抑制
```

```
_NMS = do_nms(m=_M, dx=_dx, dy=_dy)
# 5. 双阈值处理
_double_threshold = double_threshold(nms_res=_NMS)
#  双阈值处理的结果即是最终结果
_res = _double_threshold

_origin_img = plt.imread('cannyTest1.jpg')
cv2.imshow('_origin', _origin_img)
cv2.imshow('_res', _res)
cv2.waitKey(0)
cv2.destroyAllWindows()
```