长安大学
CHANG'AN UNIVERSITY

学生实验报告

实验课名称：数字图像处理

实验项目名称：图像分割(python 实现)

专 业 名 称：人工智能

班　　级：　2022240401

学　　号：　2022905226

学 生 姓 名：　郝志阳

教 师 姓 名：王伟

实验地点:计算中心机房 WM2204

实验日期：2024.10.13

一、实验名称

图像分割(python 实现)

二、实验目的与要求

本次实验主要实现并掌握图像分割的相关算法；编程实现大津阈值分割算法，并理解其最佳阈值设定的准则；编程实现基于梯度算子的边缘检测算法，并理解各个算子对图像卷积的效果及原理；编程实现霍夫检测，并深刻理解霍夫变换的数学原理。

三、实验内容

1. 大津阈值分割算法设计与实现（20 分）
2. 边缘检测
   a. 利用 Prewitt 算子边缘检测
   b. 利用 Sobel 算子边缘检测
   c. 利用 Log 算子边缘检测
   d. 利用 Canny 算子边缘检测(可调用 cv 函数)
3. 霍夫变换
   a. 霍夫变换检测直线（手写）
   b. 霍夫变换检测圆（选做）

四、实验仪器与设备

联想笔记本电脑。 13 代 i7 处理器，32G 内存。 4070 显卡。

五、实验原理

5.1 大津阈值分割法

大津阈值分割法是一种**自动**确定图像分割阈值的算法。这种方法通过**最大化类间方差**来确定最佳阈值,实现图像的二值化、阈值化。

5.1.1 基本原理

假设我们有一张灰度图像,里面有深色的物体和浅色的背景。大津法的核心思想就是:找到一个最佳的灰度值**作为阈值**,将图像中的像素分成两组(前景和背景),使得这两组之间的**差异最大**。

对于一个图像，图像的灰度级范围为$[0, L-1]$,总像素数为N。

1. 对于阈值t,我们有:
    - 背景像素概率: $w_0(t) = \sum_{i=0}^{t} p\,(i)$
    - 前景像素概率: $w_1(t) = \sum_{i=t+1}^{L-1} p\,(i) = 1 - w_0(t)$

    其中 p(i)是灰度级 i 的概率。
2. 背景和前景的平均灰度值:

$$\mu_0(t) = \frac{\sum_{i=0}^{t} i \cdot p(i)}{w_0(t)}, \quad \mu_1(t) = \frac{\sum_{i=t+1}^{L-1} i \cdot p(i)}{w_1(t)}$$

3. 总体平均灰度值:

$$\mu = w_0(t) \cdot \mu_0(t) + w_1(t) \cdot \mu_1(t)$$

4. 类间方差: $\sigma_B^2(t) = w_0(t) \cdot w_1(t) \cdot [\mu_0(t) - \mu_1(t)]^2$
5. 最佳阈值:

$$t^* = \arg\max_{0 \le t < L} \sigma_B^2(t)$$

通过公式可以看出，要求出最大类间方差，我们只需要知道四个参数：**背景像素概率w_0，前景像素概率w_1，背景平均灰度值μ_0，前景平均灰度值μ_1**。

所以实际上实现大津阈值法可以归结为以下的几个步骤:

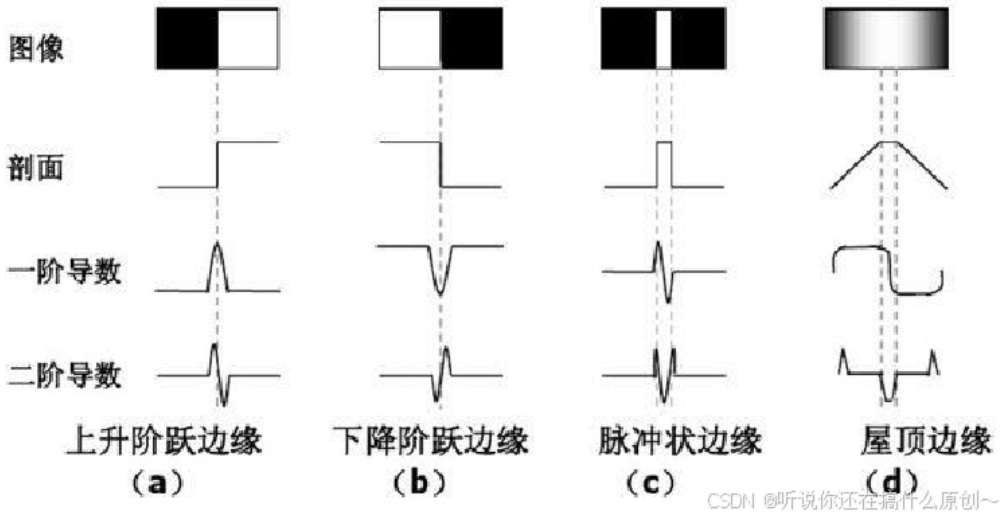
1. **计算图像直方图**: 统计图像中每个灰度级出现的频率。
2. **遍历所有可能的阈值**: 对于每个可能的阈值$t(0 \le t \le 255)$:
    - 将像素分为两类:小于等于t的为背景,大于t的为前景。
    - 计算两类像素的**平均灰度值和出现概率**。
    - 计算类间方差。
3. **选择最佳阈值**: 选择使类间方差最大的阈值作为最佳阈值。

对于类间方差最大时的灰度值，实际上此时也对应着**类内方差最小**。

根据上述原理我们也可以推出一些该方法的**优劣**:

- 首先就是大津阈值法实际上对**双峰直方图**的图像效果很好。因为此时更能体现两类的分界。
- 其次是该方法是一个全自动的划分阈值的方法，不需要人为干预。
- 但是大津阈值法对噪声很敏感，另外只考虑灰度值,没有考虑像素的空间分布信息。

5.2 边缘检测



5.2.1 基本原理

图 1：边缘类型及其导数

边缘是指图像周围**像素灰度有阶跃变化或屋顶状变化**的像素集合。

边缘具有**方向和幅度**两个特征:

- 沿边缘走向，像素值变化比较**平缓**。
- 垂直于边缘走向，像素值变化比较**剧烈**。

边缘可以分为两种:

- 一种是**阶跃性边缘**，两边的像素灰度值有着明显的不同;
- 另一种为**屋顶状边缘**，位于灰度值从增加到减少的变化转折点。
- 对于阶跃性边缘，**二阶方向导数在边缘处成零交叉**，对于屋顶状边缘，**二阶方向导数在边缘处取极值**。

在图（a）中:

- 对灰度值剖面的**一阶导数在图像由暗变明的位置处有一个向上的阶跃**，在其他位置为零。这表明可用**一阶导数的幅度值来检测边缘的存在**，幅度峰值一般对应边缘位置。
- 对灰度值剖面的二阶导数在一阶导数的阶跃上升区有一个向上的脉冲，在一阶导数阶跃下降区有一个向下的脉冲。在这两个阶跃之间有一个过零点，它的**位置正对应原始图像中边缘的位置，所以可用二阶导数过零点检测边缘位置**，而二阶导数在过零点附近的**符号**确定边缘像素在图像边缘的暗区或明区。

5.2.2 图像梯度

边缘检测的核心概念是图像梯度。梯度是一个向量,它指向图像中灰度变化最大的方向。在二维图像中,梯度有两个分量:

1. x 方向梯度 (Gx): 表示水平方向上的灰度变化
2. y 方向梯度 (Gy): 表示垂直方向上的灰度变化
3. 对于数字图像，简单的梯度表达式可以表示为:

$$G_x = I(x+1, y) - I(x-1, y)$$

$$G_y = I(x, y+1) - I(x, y-1)$$

- 梯度幅值: $G = \sqrt{G_x^2 + G_y^2}$
- 梯度方向: $\theta = \arctan\left(\frac{G_y}{G_x}\right)$

在边缘检测中,**算子**是一种用于计算图像梯度的数学工具。它通常是一个小的矩阵,用于与图像的局部区域进行**卷积运算**,从而得到该区域的梯度信息。（注：滤波器、掩膜、掩码、核、卷积核、模板、窗口、算子等，其实都是一个东西，在不同的领域叫法不同。在信号领域称为滤波器，在数学里称为核、算子，在图像处理领域称为掩膜、掩码，在深度学习领域称为卷积核）总之对于任意一个算子，我们使用它的目的就是**与图像进行卷积运算，计算图像梯度**。

一般处理图像时会将算子做如下处理:

1. 将算子(通常是 3x3 或 5x5 的矩阵)在图像上滑动。
2. 在每个位置,将算子与图像的对应区域进行**卷积运算**。
3. 卷积的结果**作为中心像素的新值,表示该点的梯度**。

算子主要有以下作用:

- 计算图像梯度: 识别图像中强度变化最显著的区域。
- 突出边缘: 增强图像中的边缘信息,使轮廓更加清晰。
- 降噪: 某些算子(如 Sobel)在计算梯度的同时也有平滑图像的作用,可以减少噪声的影响。
- 方向性检测: 不同的算子可以对特定方向(如水平、垂直或对角线)的边缘更敏感。

5.2.3 Prewitt 算子

Prewitt 算子是一种用于边缘检测的一阶微分算子,通过计算图像在水平和垂直方向上的**灰度差分**来检测边缘。

5.2.3.1 卷积核

Prewitt 算子使用两个 3x3 的卷积核,分别用于检测**水平**和**垂直**方向的边缘:

水平方向卷积核:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

垂直方向卷积核:

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

卷积核分别对应于**图像在 x 和 y 方向的一阶偏导数的离散近似**。

5.2.3.2 边缘强度和方向

可以用上述的 prewitt 算子对图像进行卷积运算从而得到梯度:

1. 水平梯度: $Gx = I(x+1, y) - I(x-1, y)$
2. 垂直梯度: $Gy = I(x, y+1) - I(x, y-1)$

在此基础上计算:

- 边缘强度: $G = \sqrt{Gx^2 + Gy^2}$
- 边缘方向: $\theta = \arctan(Gy/Gx)$

进一步确定边缘位置。

5.2.4 Sobel 算子

5.2.4.1 卷积核

Sobel 算子同样使用两个 3x3 的卷积核:

水平方向卷积核:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

垂直方向卷积核:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

与 Prewitt 算子相比,Sobel 算子在中心位置给予了更大的权重。对噪声有一定的**平滑作用**,相比 Prewitt 算子更不敏感。

5.2.4.2 边缘强度和方向

计算方法与 Prewitt 算子相同:

1. 水平梯度: $Gx = I(x+1,y) - I(x-1,y) + 2[I(x+1,y+1) - I(x-1,y-1)]$
2. 垂直梯度: $Gy = I(x,y+1) - I(x,y-1) + 2[I(x+1,y+1) - I(x-1,y-1)]$

边缘强度和方向的计算与 Prewitt 算子相同。

5.2.5 LoG (Laplacian of Gaussian) 算子

LoG 算子,全称为高斯拉普拉斯算子,是一种结合了**高斯滤波和拉普拉斯算子**的边缘检测方法。它首先对图像进行高斯平滑以减少噪声,然后应用拉普拉斯算子来检测边缘。

5.2.5.1 原理

1. **高斯滤波**: 首先,使用高斯函数对图像进行平滑处理,减少噪声的影响。高斯函数在二维空间中的表达式为:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

   其中,**σ** 控制了高斯函数的宽度。

2. **拉普拉斯算子**: 拉普拉斯算子是一个二阶微分算子,用于检测图像中的亮度急剧变化区域。二维拉普拉斯算子定义为:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

   该算子用卷积核可以表示为 3x3 或 5x5 的矩阵。表示为 5x5 的卷积核如下:

$$\begin{bmatrix} -2 & -4 & -4 & -4 & -2 \\ -4 & 0 & 8 & 0 & -4 \\ -4 & 8 & 24 & 8 & -4 \\ -4 & 0 & 8 & 0 & -4 \\ -2 & -4 & -4 & -4 & -2 \end{bmatrix}$$

3. **LoG 算子**: LoG 算子将高斯滤波和拉普拉斯算子结合，对 图像进行卷积运算。

$$LoG(x,y) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2+y^2}{2\sigma^2}\right] e^{-\frac{x^2+y^2}{2\sigma^2}}$$

LoG 算子对噪声具有较强的抵抗能力，但是计算复杂度相对高一些。

5.2.6 Canny 边缘检测算子

Canny 边缘检测是一种多阶段的边缘检测算法，它被广泛认为是一种**最优**的边缘检测方法，能够提供良好的检测、定位和最小响应。

5.2.6.1 原理

Canny 算法的目标是找到图像中存在的最优边缘。往往通过以及几个步骤来实现：

1. **高斯滤波**： 首先，使用高斯滤波器来平滑图像，减少噪声。高斯滤波器的二维形式为：

   $$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

   其中，σ控制了滤波器的"模糊"程度。

2. **计算梯度**： 使用类似 **Sobel 算子**的方法计算图像的梯度大小和方向。但是往往采用 5x5 的卷积核。

3. **非极大值抑制**： 沿着边缘方向，如果像素不是局部最大值，**则将其抑制（设为 0）**。这一步"细化"了边缘。

4. **双阈值处理**： 使用两个阈值 T_{low} 和 T_{high}：

   - 如果像素值 > T_{high}，则认为是强边缘。
   - 如果 T_{low} < 像素值 < T_{high}，则认为是弱边缘。
   - 如果像素值 < T_{low}，则抑制。

5. **边缘跟踪**： 通过滞后技术连接边缘。强边缘点被立即确认为边缘，而弱边缘点只有在与强边缘点相连时才被确认为边缘。

5.3 霍夫变换

霍夫变换是一种用于检测图像中**特定形状**(如直线、圆)的特征提取技术。它的主要优点是能够**在噪声环境下**有效地识别不完整的形状。

5.3.1 基本原理

霍夫变换利用点和线之间的对偶性，将图像空间中直线上离散的像素点通过参数方程映射为**霍夫空间中的曲线**，并将**霍夫空间中多条曲线的交点**作为直线方程的参数映射为图像空间中的直线。给定直线的参数方程，可以利用霍夫变换来检测图像中的直线。 直线空间和霍夫空间的映射关系如下：

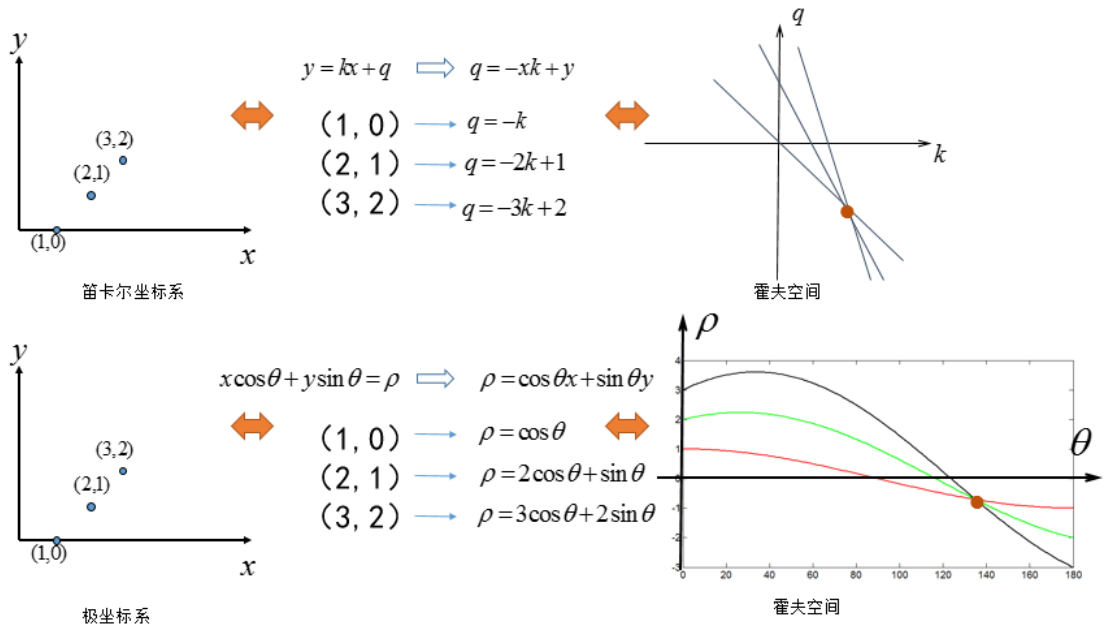


图 2：直线空间和霍夫空间的映射关系

为了避免对于直线垂直造成的斜率无限大无法表示的问题，通常我们采用**极坐标系**来表示直线。 转化为极坐标下的参数方程:

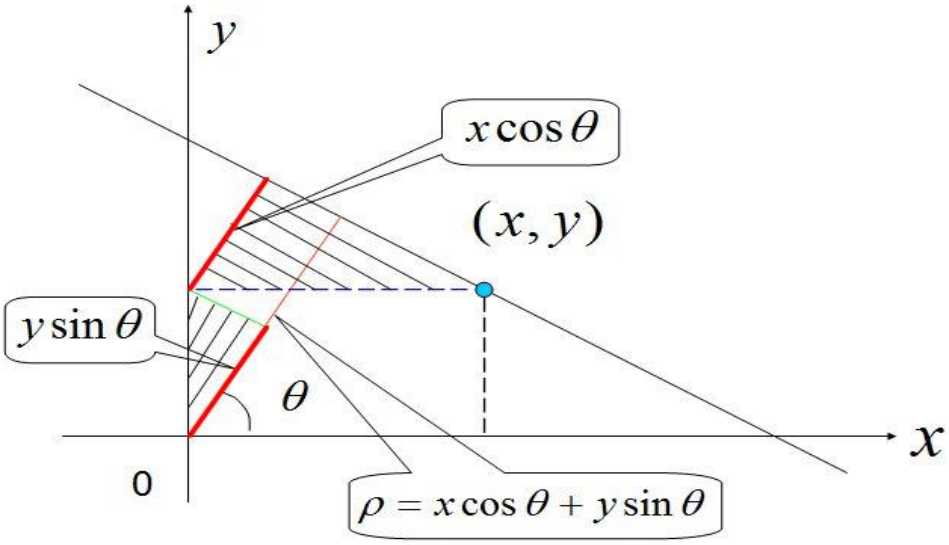


图 3：转化为极坐标下的参数方程

图像空间中的点，对应了霍夫空间中的曲线。 曲线的交点确定了一组参数，能够描述图像空间中的特定直线。

点和线的对偶性:

1. 图像空间中的点，对应霍夫空间中的直线。

2. 图像中的直线，对应霍夫空间中的点。
3. 共点的直线，在霍夫空间中对应的点在一条直线上。
4. 共线的点，在霍夫空间中对应的直线交与一点。

六、实验过程及代码

6.1 大津阈值分割算法

6.1.1 实现过程

6.1.1.1 NumPy 实现

1. 计算图像直方图:
    - 使用 `cv.calcHist` 函数计算图像的直方图。
2. 遍历所有可能的阈值:
    - 使用 NumPy 的 **`cumsum`** 函数计算累积和,高效得到不同阈值下的像素分布。
    - 利用 NumPy 的广播机制计算平均灰度值u_0和u_1,避免了显式的循环。
3. 计算类间方差:
    - 用 NumPy 的向量化操作计算类间方差。
4. 找出最佳阈值:
    - 用 `np.argmax` 函数找出使类间方差最大的阈值。
5. 应用阈值:
    - 用 NumPy 的布尔索引快速地对图像进行二值化。

NumPy 实现中的核心部分:

```
weight1 = np.cumsum(hist)
weight2 = total - weight1
mean1 = np.cumsum(hist * np.arange(256)) / (weight1 + 1e-10)
mean2 = (np.cumsum(hist * np.arange(256))[-1] - np.cumsum(hist * np.ara
nge(256))) / (weight2 + 1e-10)
variance = weight1 * weight2 * (mean1 - mean2) ** 2
max_t = np.argmax(variance)
```

其中

- `weight1` 表示灰度值小于等于当前阈值的像素比例。(概率累计和，表示像素是此类的概率)
- `weight2` 表示剩余像素的比例,即大于当前阈值的像素比例。
- `mean1` 表示灰度值小于等于当前阈值的像素的平均灰度值。
- `mean2` 表示灰度值大于当前阈值的像素的平均灰度值。
- `variance` 表示类间方差。

- max_t 表示使类间方差最大的阈值。

6.1.1.2 OpenCV 实现

1. 直接调用 cv.threshold 函数,指定 cv.THRESH_OTSU 参数。
2. 函数返回最佳阈值和分割后的图像。

OpenCV 实现的核心部分:

```
max_t, new_img = cv.threshold(img, 0, 255, cv.THRESH_OTSU)
```

6.1.1.3 具体计算过程

假设有一个小的 4x4 灰度图像,灰度值范围为 0-7:

$$\begin{bmatrix} 2 & 5 & 5 & 3 \\ 1 & 4 & 6 & 2 \\ 3 & 5 & 7 & 1 \\ 2 & 3 & 4 & 6 \end{bmatrix}$$

- **计算直方图**

首先,统计每个灰度值的出现次数:

灰度值	0	1	2	3	4	5	6	7
频数	0	2	3	2	3	3	2	1

- **遍历所有可能的阈值**

以阈值$t = 3$为例:

1. 计算权重: $w0 = (2 + 3 + 2)/16 = 7/16\, w1 = (3 + 3 + 2 + 1)/16 = 9/16$
2. 计算平均灰度值: $\mu 0 = (1 * 2 + 2 * 3 + 3 * 2)/7 = 12/7\, \mu 1 = (4 * 3 + 5 * 3 + 6 * 2 + 7 * 1)/9 = 44/9$
3. 计算类间方差: $\sigma_B^2 = w0 * w1 * (\mu 0 - \mu 1)^2$ = (7/16) * (9/16) * (12/7 - 44/9)^2 ≈ 1.37

- **对所有可能的阈值重复 2,找出使类间方差最大的阈值**

最终结果:

阈值	0	1	2	3	4	5	6	7
σB^2	0.0	0.22	0.81	1.37	1.42	1.22	0.56	0.0

最大的类间方差出现在阈值$t = 4$时,因此$t = 4$就是最佳阈值。

- **应用阈值**

最后,使用这个阈值对图像进行二值化:

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

0 表示小于等于阈值的像素(背景),1 表示大于阈值的像素(前景)。

6.2 边缘检测

6.2.1 Prewitt 算子

采用了三种方法实现 Prewitt 算子:

1. **NumPy FFT（快速傅里叶变换）实现**:
    - 定义 Prewitt 算子的水平和垂直方向卷积核。
    - 使用 `np.pad` 对输入图像进行边缘填充。
    - 利用 `np.fft.fft2` 对填充后的图像和卷积核进行快速傅里叶变换(FFT)。
    - **在频域中进行乘法运算**，相当于空间域的卷积。
    - 使用 `np.fft.ifft2` 将结果转回空间域。
    - 计算梯度幅值并归一化。

```
img_fft = np.fft.fft2(img_pad)
gx = np.real(np.fft.ifft2(img_fft * kernel_x_fft))[1:-1, 1:-1]
gy = np.real(np.fft.ifft2(img_fft * kernel_y_fft))[1:-1, 1:-1]
```

上述代码中，`img_pad` 是填充后的图像，`kernel_x_fft` 和 `kernel_y_fft` 分别是水平和垂直方向卷积核的傅里叶变换。可以看到，首先将图像和卷积核变换到频域，然后在频域中进行乘法运算，`gx` 和 `gy` 是最终的梯度图像。`np.real` 函数用于提取实部，`np.fft.ifft2` 函数用于将结果转回空间域。

2. **SciPy 卷积实现**:
    - 定义相同的卷积核。
    - 使用 `scipy.signal.convolve2d` 函数直接进行二维卷积。
    - 计算梯度幅值并归一化。

```
gx = convolve2d(img, kernel_x, mode='same', boundary='symm')
gy = convolve2d(img, kernel_y, mode='same', boundary='symm')
```

这里直接使用 scipy.signal.convolve2d 函数进行卷积运算，mode='same'表示输出与输入相同大小，boundary='symm'表示对边缘进行对称填充。分别对水平和竖直方向进行卷积运算得到梯度结果。

3. **OpenCV 实现**:
    - 使用 cv.getDerivKernels 获取 Prewitt 算子的分离卷积核。
    - 应用 cv.sepFilter2D 进行分离卷积操作。
    - 使用 cv.magnitude 计算梯度幅值。

```
img_prewittx = cv.sepFilter2D(img, cv.CV_32F, kernelx[0], kernelx[1])
img_prewitty = cv.sepFilter2D(img, cv.CV_32F, kernely[0], kernely[1])
```

6.2.2 Sobel 算子

Sobel 算子的实现过程与 Prewitt 算子非常相似,主要区别在于**卷积核的定义**。

1. **NumPy FFT 实现**:
    - 过程与 Prewitt 算子相同,只是使用 Sobel 的卷积核。
2. **SciPy 卷积实现**:
    - 使用 scipy.signal.convolve2d 函数,但应用 Sobel 的卷积核。
3. **OpenCV 实现**:
    - 直接使用 cv.Sobel 函数,简化了实现过程。

NumPy 和 SciPy 的实现与 Prewitt 相似,只是卷积核不同。OpenCV 实现的核心代码:

```
sobelx = cv.Sobel(img, cv.CV_64F, 1, 0, ksize=3)
sobely = cv.Sobel(img, cv.CV_64F, 0, 1, ksize=3)
```

6.2.2.1 具体计算过程

假设有以下 3x3 的图像块:

$$\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}$$

使用 Sobel 算子来计算中心像素(值为 50)的梯度。

1. 应用 Sobel 算子

Sobel 水平方向卷积核:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Sobel 垂直方向卷积核:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

2. 计算水平梯度Gx

Gx = (30 * 1 + 60 * 2 + 90 * 1) - (10 * 1 + 40 * 2 + 70 * 1) \\ = (30 + 120 + 90) - (10 + 80 + 70) = 240 - 160 = 80

3. 计算垂直梯度Gy

Gy = (70 * 1 + 80 * 2 + 90 * 1) - (10 * 1 + 20 * 2 + 30 * 1) \\ = (70 + 160 + 90) - (10 + 40 + 30) = 320 - 80 = 240

4. 计算梯度幅值

G = sqrt(Gx^2 + Gy^2) \\ = sqrt(80^2 + 240^2) \\ = sqrt(6400 + 57600) ≈ 252.98

5. 计算梯度方向

θ = arctan(Gy / Gx) \\ = arctan(240 / 80) \\ = arctan(3) ≈ 71.57°

通过上述计算可知:

- **梯度幅值(约 253)表示在这个位置的边缘强度**。值越大,表示边缘越明显。
- **梯度方向(约 71.57°)表示边缘朝向**。(朝向右上方)。

6.2.3 LoG (Laplacian of Gaussian) 算子

6.2.3.1 实现过程

LoG 算子的实现结合了高斯滤波和拉普拉斯算子:

1. **NumPy 实现**:
   - 定义 LoG 卷积核。
   - 使用 **FFT 方法**进行卷积(类似 Prewitt 和 Sobel 的 NumPy 实现)。
   - 寻找零交叉点来确定边缘。
2. **SciPy 实现**:
   - 使用 `scipy.signal.convolve2d` 函数与 LoG 卷积核进行卷积。
3. **OpenCV 实现**:
   - 先使用 `cv.GaussianBlur` 进行高斯模糊。
   - 然后使用 `cv.Laplacian` 应用拉普拉斯算子。

OpenCV 实现的核心代码:

```
img_blur = cv.GaussianBlur(img, (3, 3), 0)
new_img = cv.Laplacian(img_blur, cv.CV_64F, ksize=5)
```

6.2.3.2 具体计算过程

假设有一个 7×7 的图像块：

$$\begin{bmatrix} 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 20 & 20 & 20 & 20 & 20 & 10 \\ 10 & 20 & 50 & 50 & 50 & 20 & 10 \\ 10 & 20 & 50 & 100 & 50 & 20 & 10 \\ 10 & 20 & 50 & 50 & 50 & 20 & 10 \\ 10 & 20 & 20 & 20 & 20 & 20 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 \end{bmatrix}$$

1. LoG 卷积核

使用 **σ = 1.4**（高斯标准差）的 5×5 LoG 卷积核：

$$\begin{bmatrix} -2 & -4 & -4 & -4 & -2 \\ -4 & 0 & 8 & 0 & -4 \\ -4 & 8 & 24 & 8 & -4 \\ -4 & 0 & 8 & 0 & -4 \\ -2 & -4 & -4 & -4 & -2 \end{bmatrix}$$

2. 应用 LoG 卷积

计算中心像素（值为 100）的 LoG 响应。将卷积核与图像块中心 5×5 区域对应位置的值相乘并求和：

$$\begin{aligned} LoG_{response} = \quad & (-2 \times 20) + (-4 \times 50) + (-4 \times 50) + (-4 \times 50) + (-2 \times 20) \\ & +(-4 \times 20) + (0 \times 50) + (8 \times 50) + (0 \times 50) + (-4 \times 20) \\ & +(-4 \times 50) + (8 \times 50) + (24 \times 100) + (8 \times 50) + (-4 \times 50) \\ & +(-4 \times 20) + (0 \times 50) + (8 \times 50) + (0 \times 50) + (-4 \times 20) \\ & +(-2 \times 20) + (-4 \times 50) + (-4 \times 50) + (-4 \times 50) + (-2 \times 20) \end{aligned}$$

计算结果：

$$LoG_{response} = 1600$$

LoG 响应为**正值 1600**，这表示在这个位置检测到了显著的亮度变化。中心像素（100）比周围像素亮得多，形成了一个局部的峰值，这被 LoG 算子识别为一个潜在的边缘或角点。

σ 是高斯函数的标准差，它在 LoG 算子中起着关键作用：

- 控制平滑程度：较大的σ值会产生更宽的高斯分布，从而增加平滑效果，有助于抑制噪声，但可能会模糊一些细节。

- 影响检测尺度：σ值决定了 LoG 算子对不同尺度特征的敏感度。**较小的σ值适合检测精细的边缘和细节**，而较大的σ值则更适合检测大尺度的边缘和结构。
- 决定卷积核大小：通常，**卷积核的大小设置为(6σ+1)×(6σ+1)**，向下取整到最近的奇数。

6.2.4 2.4 Canny 边缘检测

6.2.4.1 实现过程

Canny 边缘检测算法是最复杂的，采用了 OpenCV 实现。

1. **OpenCV 实现**:
    - 直接使用 `cv.Canny` 函数,简化了实现过程。

OpenCV 实现的核心代码:

```
new_img = cv.Canny(img, 100, 200)
```

其中，100 是低阈值，200 是高阈值。主要作用是**抑制噪声**，同时尽可能多地保留边缘信息。

6.3 霍夫变换

6.3.1 霍夫变换求解霍夫域图像

霍夫变换的核心思想是将图像空间中的直线检测问题**转换为参数空间中的点检测问题**。

1. **边缘检测预处理**:

    ```
    img = cv.threshold(img, 127, 255, cv.THRESH_BINARY)[1]
    img = 255 - img
    ```

    - 首先对输入图像进行**二值化处理**,将灰度图像转换为只有黑白两种颜色的图像。
    - 阈值设为 127,高于此值的像素设为 255(白色),低于此值的设为 0(黑色)。
    - 然后进行反转操作(255 - img),使边缘像素为白色(255),背景为黑色(0)。
2. **参数空间初始化**:

    ```
    rows, cols = img.shape[:2]
    diagonal = int(np.sqrt(rows**2 + cols**2))
    hg_rows, hg_cols = 180, diagonal * 2
    ```

    - 计算图像对角线长度 `diagonal`,这决定了ρ的范围。
    - 设置霍夫空间的尺寸:

- θ的范围是 0 到 179 度,共 180 个值。
- ρ的范围是-diagonal 到+diagonal,总长度为 diagonal * 2。

3. **找出边缘点**:

```
y_idxs, x_idxs = np.nonzero(img)
```

   - 使用 np.nonzero()找出所有非零(即边缘)像素的坐标。

4. **计算所有可能的θ值**:

```
thetas = np.deg2rad(np.arange(hg_rows))
cos_thetas = np.cos(thetas)
sin_thetas = np.sin(thetas)
```

   - 生成 0 到 179 度的所有角度,并转换为弧度。
   - 预先计算所有角度的余弦和正弦值,以提高后续计算效率。

5. **计算ρ值**:

```
rhos = np.round(x_idxs[:, None] * cos_thetas + y_idxs[:, None] *
sin_thetas).astype(int)
rhos += diagonal
```

   - 对每个边缘点,计算所有可能θ值对应的ρ值。
   - 公式: ρ = x * cos(θ) + y * sin(θ)
   - [:, None]用于广播操作,使每个点的坐标与所有θ值进行计算。
   - 加上 diagonal 是为了将ρ的范围从[-diagonal, +diagonal]调整到[0, 2*diagonal]。

6. **累加器投票**:

```
hough_img = np.zeros((hg_rows, hg_cols), dtype=np.int32)
np.add.at(hough_img, (np.arange(hg_rows)[None, :], rhos), 1)
```

   - 创建一个二维数组 hough_img 作为累加器。
   - 使用 np.add.at()函数进行投票:
     - 对于每个边缘点和每个θ值,在对应的(θ, ρ)位置加 1。
   - 这个操作相当于对每条可能的直线进行投票。

最终的 hough_img 就是霍夫域图像。其中每个点(θ, ρ)的值表示原图中可能存在的直线的"可能性"。值越高的点,表示对应的直线在原图中出现的可能性越大。

好的,我会更详细地解释霍夫变换检测直线的过程。

6.3.2 霍夫变换检测直线

6.3.2.1 NumPy 实现

1. **霍夫变换**:

```
hough_img = hough_transform(img)
```

- 使用之前定义的 hough_transform 函数计算霍夫空间。
- hough_img 是一个二维数组,其中每个元素(θ, ρ)的值表示对应直线的"投票数"。

2. **峰值检测**:

```
threshold = np.max(hough_img) // 2
peaks = np.argwhere(hough_img > threshold)
```

- 设置阈值为霍夫空间中最大值的一半。这是一个经验值,可以根据需要调整。
- 使用 np.argwhere 找出所有超过阈值的点的坐标。这些点对应可能的直线。

3. **直线绘制**:

```
for theta, rho in peaks[:num]:
    a = np.cos(np.deg2rad(theta))
    b = np.sin(np.deg2rad(theta))
    x0 = a * (rho - diagonal)
    y0 = b * (rho - diagonal)
    x1 = int(x0 + 1000 * (-b))
    y1 = int(y0 + 1000 * (a))
    x2 = int(x0 - 1000 * (-b))
    y2 = int(y0 - 1000 * (a))

    cv.line(img_color, (x1, y1), (x2, y2), (0, 0, 255), 2)
```

- 遍历前 num 个峰值点(最强的直线)。
- 将霍夫空间的(θ, ρ)转换回图像空间的直线参数:
  - a = cos(θ), b = sin(θ)是直线的方向向量。
  - (x0, y0)是直线上的一点,通过 rho = x*cos(θ) + y*sin(θ) 计算得到。
  - 注意这里减去 diagonal 是为了将ρ的范围从[0, 2*diagonal]调整回[-diagonal, +diagonal]。
- 计算直线上的两个点(x1, y1)和(x2, y2):
  - 这里使用了 1000 作为延长因子,确保直线跨越整个图像。
- 使用 OpenCV 的 line 函数在原图上绘制直线。

6.3.2.2 OpenCV 实现

```
lines = cv.HoughLines(255 - img, 1, np.pi / 180, 100)
if lines is not None:
    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(hough_img, (x1, y1), (x2, y2), (0, 0, 255), 2)
```

1. **霍夫变换**:
    - `cv.HoughLines` 函数直接执行霍夫变换和峰值检测。
    - 参数说明:
        - `255 - img`: 输入的边缘图像(需要反转,因为 OpenCV 期望边缘为白色)。
        - `1`: ρ的分辨率,单位为像素。
        - `np.pi / 180`: θ的分辨率,单位为弧度。
        - `100`: 累加器阈值参数。只有获得足够投票的直线才会被返回。
2. **直线参数转换和绘制**:
    - 遍历检测到的每条直线。
    - 将(ρ, θ)转换为直线上的点坐标。
    - 使用 `cv.line` 函数绘制直线。

6.3.3 霍夫变换检测圆

霍夫变换检测圆的原理与检测直线类似,但**参数空间从 2D 变为 3D(圆心 x,y 坐标和半径 r)**。由于计算复杂度高,使用 OpenCV 的 `HoughCircles` 函数来实现。以下是详细的步骤说明:

1. **图像预处理**:

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

    - 应用高斯模糊减少噪声。
    - 参数`(5, 5)`指定高斯核的大小,`0` 表示自动计算标准差。
    - 这一步很重要,因为它可以减少假阳性检测。

```
edges = cv.Canny(img_blur, 50, 150, apertureSize=3)
```

    - 使用 Canny 边缘检测算法提取边缘。
    - `50` 和 `150` 分别是低阈值和高阈值。
    - `apertureSize=3` 指定 Sobel 算子的大小。
2. **圆检测**:

```
circles = cv.HoughCircles(edges, cv.HOUGH_GRADIENT, dp=1, minDist
=100,
                          param1=100, param2=30, minRadius=10, ma
xRadius=100)
```

- cv.HoughCircles 函数执行霍夫圆变换。
- 参数说明:
  - edges: 输入的边缘图像。
  - cv.HOUGH_GRADIENT: 检测方法,这是目前唯一可用的方法。
  - dp=1: 累加器分辨率与图像分辨率的比率。dp=1 时它们相同,dp=2 时累加器的分辨率是图像的一半。
  - minDist=100: 检测到的圆之间的最小距离。
  - param1=100: Canny 边缘检测的高阈值。
  - param2=30: 累加器阈值。值越小,检测到的圆越多(包括假阳性)。
  - minRadius=10, maxRadius=100: 圆半径的最小值和最大值。

3. **结果处理和绘制**:

```
if circles is not None:
    circles = np.uint16(np.around(circles))
    for i in circles[0, :]:
        cv.circle(new_img, (i[0], i[1]), i[2], (0, 255, 0), 2)
        cv.circle(new_img, (i[0], i[1]), 2, (0, 0, 255), 3)
```

- 检查是否检测到圆。
- 将浮点数坐标转换为整数。
- 遍历每个检测到的圆:
  - i[0], i[1]是圆心坐标,i[2]是半径。
  - 绘制圆周: cv.circle(new_img, (i[0], i[1]), i[2], (0, 255, 0), 2)
    - (0, 255, 0)是绿色,2 是线宽。
  - 绘制圆心: cv.circle(new_img, (i[0], i[1]), 2, (0, 0, 255), 3)
    - (0, 0, 255)是红色,3 是点的大小。

6.3.4 霍夫圆变换

1. 对于边缘图像中的每个点,以该点为圆心,在参数空间中绘制可能的圆。
2. 这些圆在 3D 参数空间(x, y, r)中相交的点,对应于原图中可能存在的圆。
3. 使用累加器数组记录每个(x, y, r)组合获得的"投票"。
4. 选择投票数超过阈值的(x, y, r)组合作为检测到的圆。

七、实验结果与分析

7.1　大津阈值法图像分割

大津阈值法处理时间:

类型	处理时间（ms）
手写实现	1.009
opencv 实现	0.005 以内

处理效果:

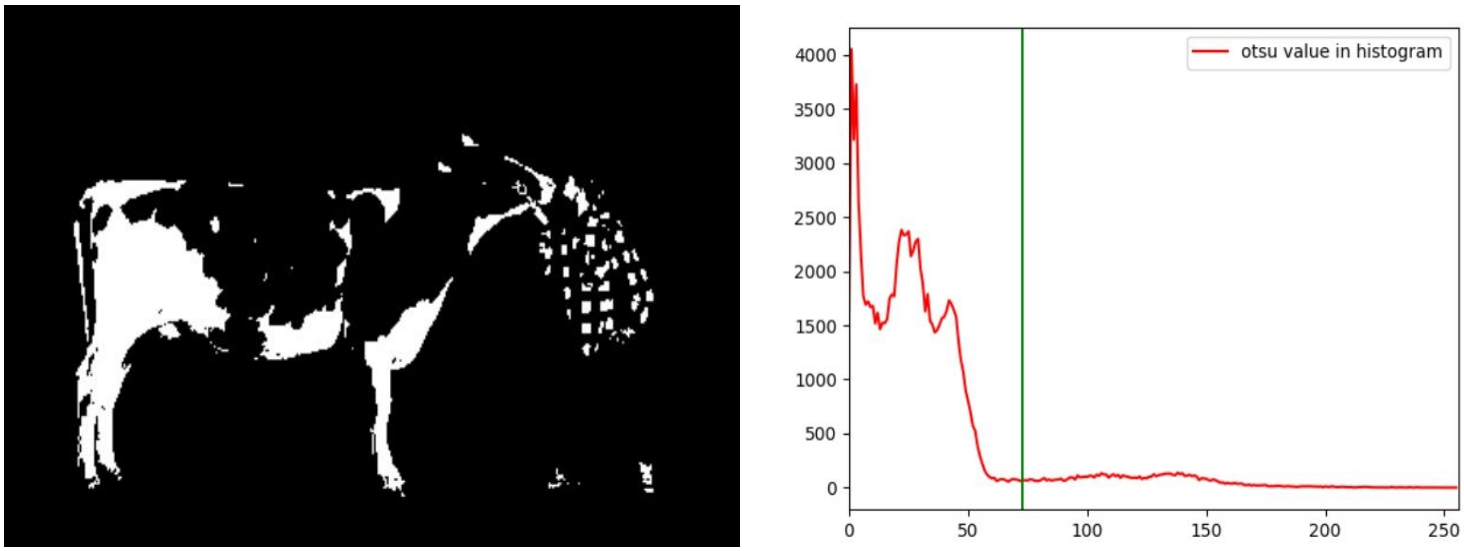


图 4：大津阈值图像分割效果

7.2　边缘检测

各边缘检测处理时间:

算子	实现方式	处理时间（ms）
prewiit	numpy-FFT	208.593
	scipy-convolve2d	43.614
	opencv	5.005
sobel	numpy-FFT	205.982
	scipy-convolve2d	43.000
	opencv	9.001
LoG	numpy-FFT	99.527

算子	实现方式	处理时间（ms）
	scipy-convolve2d	48.507
	opencv	8.600
Canny	opencv	2.175

处理效果:

图 5：prewiit 算子和 sobel 算子处理效果

图 6：loG 算子和 canny 算子处理效果

7.3 霍夫变换

霍夫变换处理时间:

变换类型	处理时间（ms）
手写实现直线检测	20.081
opencv 实现直线检测	1.000
opencv 实现圆检测	6.492

处理效果:

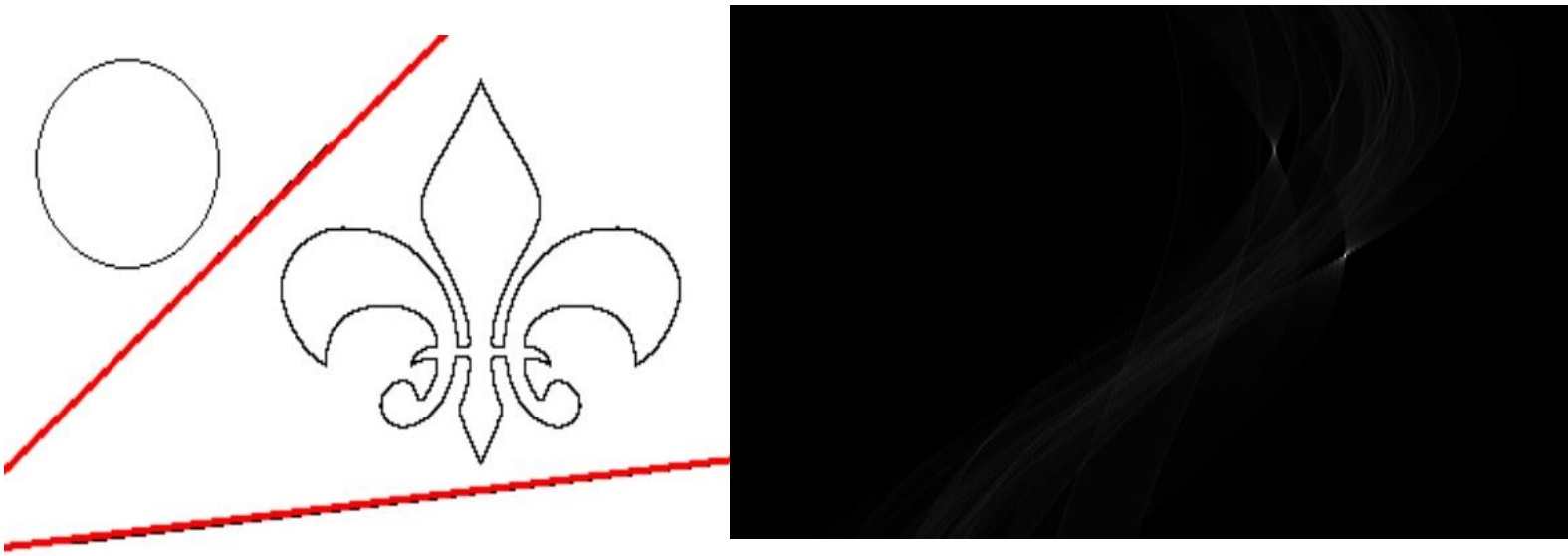


图 7：numpy 实现霍夫变换直线检测和霍夫域

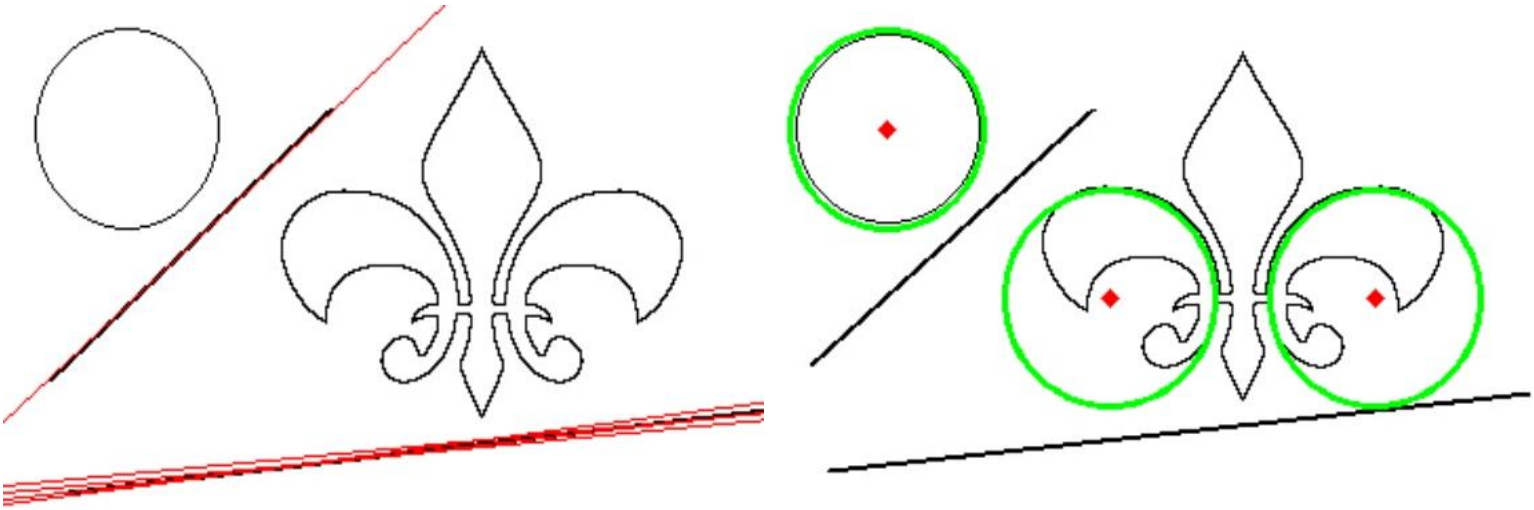


图 8：opencv 实现霍夫变换直线检测和圆检测效果

八、实验总结及心得体会

1. **算法优化**的重要性

做边缘检测时,我比较了用 NumPy、SciPy 和 OpenCV 实现的 Prewitt、Sobel 和 LoG 算子的运行时间。结果显示,OpenCV 的实现速度远快于其他两种方法。但是通过**傅里叶变换、或者现成的库函数实现模板卷积**也是很快的（至少要比手写遍历快的多的多）。从中可以看出，边缘检测算法的主要耗时是在**卷积运算**上，探讨使用什么方法减少计算量提高计算速度对提高性能很有帮助。

2. **巧用傅里叶运算**

傅里叶变换将**信号从时域(或空域)转换到频域**。在图像处理中,我们主要使用二维离散傅里叶变换(2D DFT)。 傅里叶变换的核心思想是:**任何周期函数都可以表示为不同频率的正弦波的加权和**。对于图像来说,可以将图像分解为不同频率的成分。然后用傅里叶变换的重要性质**卷积定理**:时域(空域)的卷积等价于频域的乘积。即:

$$f * g = F^{(-1)}[F(f) \cdot F(g)]$$

利用这一性质,可以实现图像的快速卷积，**对图像和卷积核进行傅里叶变换，之后将两个变换结果进行点乘，最后对点乘结果进行逆傅里叶变换即可**。 这种方法特别适用于大尺寸卷积核,因为传统的空域卷积复杂度为 O(n^2*m²),而使用 FFT 的方法复杂度可以降到 O(n2log(n)),其中 n 是图像的边长,m 是卷积核的边长。 这种方法不仅在理论上优雅,在实际应用中也能显著提高处理速度,特别是对于**大尺寸图像或复杂**的卷积操作。

3. **霍夫空间**的映射关系和投票计算

实现霍夫变换时。最开始我对如何将图像空间中的直线检测问题转换为参数空间中的点检测问题不太明白。特别是怎么高效地进行累加器的投票过程如何实现。最终使用 np.add.at()函数来并行处理投票过程,减少处理时间。同时在算法中用 NumPy 的向量化操作来优化代码,避免显式的循环。

4. 不同方法的**比较与选择**

我实现了多种边缘检测算子(Prewitt、Sobel、LoG、Canny)和霍夫变换(直线检测和圆检测)。通过比较这些方法的效果和效率,我更加理解如何根据具体问题选择合适的算法。比如,Canny 边缘检测虽然计算复杂度较高,但在噪声环境下表现更好;霍夫变换在检测特定形状(如直线和圆)时非常有效。