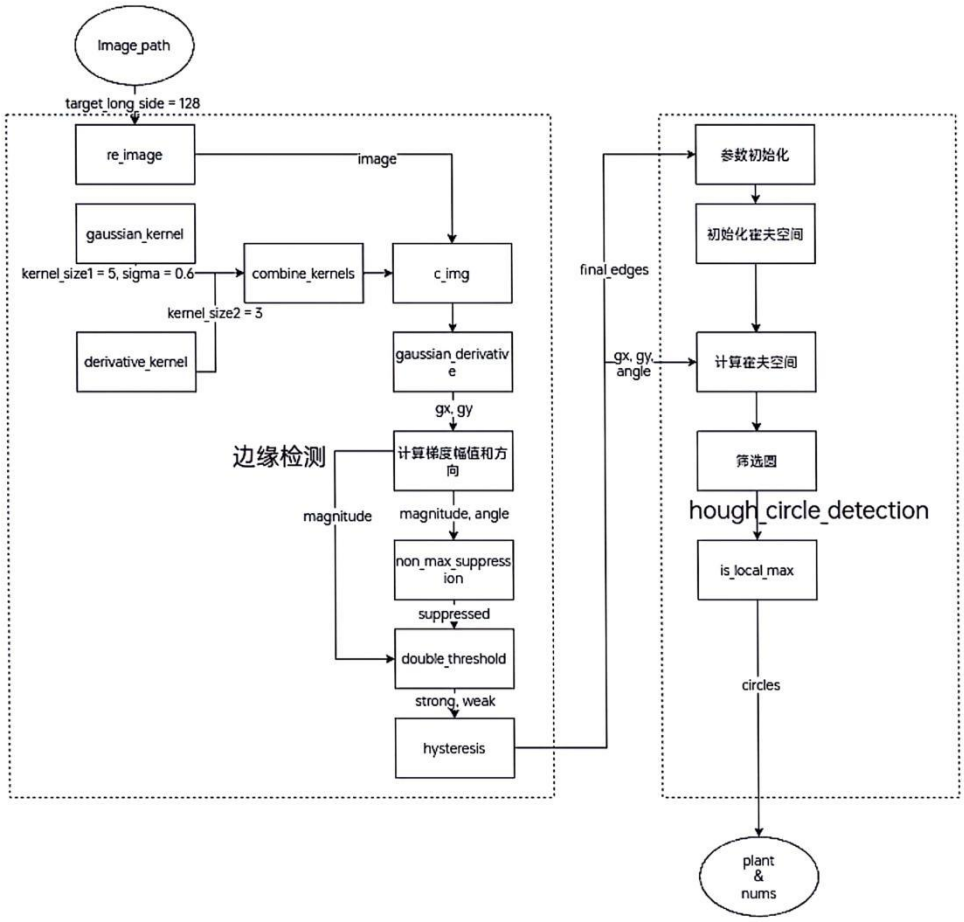
一、算法流程

1、数据流图



2、算法流程文字描述

2.1 预处理与边缘检测

2.1.1 读取图像为灰度

2.1.2 对图像进行自适应压缩

2.1.3 计算高斯偏导数

2.1.4 计算梯度幅度和方向

2.1.5 非最大值抑制

2.1.6 应用双门限（自适应高门限、低门限）

根据图像梯度的统计信息来自适应地确定高门限和低门限。高门限设定为梯度幅值的 80%分位数，低门限设定为高门限的一半。这样可以根据图像的不同特性进行灵活的门限设置，以便更好地捕获边缘。

2.1.7 使用滞后阈值跟踪边缘

从高门限边缘像素出发，沿着边缘方向进行跟踪，并将与高门限相连的低门限边缘像素标记为边缘像素。

2.2 应用霍夫变换圆检测

2.2.1 参数初始化

2.2.2 初始化霍夫空间（默认圆心只能在图片上）

2.2.3 计算霍夫空间

2.2.4 根据阈值和最小距离筛选圆

2.2.5 非极大值抑制过滤和标记检测到的圆

2.3 后处理与输出

2.3.1 使用 cv2 在图上画出圆

2.3.2 输出硬币数量

二、核心函数功能

1、图片自适应压缩 re_image(image_path, target_long_side = 128)

减小 edges 非零点个数，优化后续霍夫圆变换计算时间。

参数	image_path（str）：原始图像的文件路径。 target_long_side（int）：目标长边的大小，默认为 128 像素。
返回值	compressed_image（numpy.ndarray）：自适应缩放后的图像，为灰度图像。
注意事项	函数会保持图像的长宽比例，根据目标长边的大小自适应缩放图像。

2、使用卷积的高斯模糊与近似偏导操作 gaussian_derivative(image, kernel_size1 = 5, sigma = 0)

使用卷积近似替代偏导计算操作（核大小为 3），高斯、偏导卷积核先卷积，再与图片卷积，优化计算。

参数	image（numpy.ndarray）：输入图像。 kernel_size1（int）：高斯核核大小，一般越小越尖锐，越大越平滑。默认为 5。 sigma（float）：高斯核标准差，一般越小越尖锐，越大越平滑。默认为 0，即根据经验函数 sigma = 0.3 * ((size - 1) * 0.5 - 1) + 0.8 自适应计算。
返回值	derivative_x、derivative_y（numpy.ndarray）：图像在 x、y 方向上的高斯偏导数。

3、NMS 非极大值抑制

non_max_suppression(magnitude, angle)

参数	magnitude（numpy.ndarray）：图像的梯度幅值。 angle（numpy.ndarray）：图像的梯度方向。
返回值	suppressed（numpy.ndarray）：执行非最大值抑制后的结果矩阵。

is_local_max(hough_space, y, x, r, min_dist, param2)

参数	hough_space（numpy.ndarray）：霍夫空间，包含了检测到的圆的投票信息。 y、x、r（int）：霍夫空间内圆心的 y、x、r 坐标。 min_dist（int）：最小圆心间距，用于限制邻域范围。 param2（int）：圆半径变化范围，用于限制邻域范围。
返回值	Bool：如果给定点是局部最大值，则返回 True；否则返回 False。

4、基于梯度的圆选择

在为霍夫空间投票时，参考 cv2 源码，基于梯度来减少运算量。

（1）如果当前的像素不是边缘点，或者水平梯度值和垂直梯度值都为 0，则继续循环。因为如果满足上面条件，该点一定不是圆周上的点。

（2）接着便可以在梯度的正反两个方向上进行位移，并对累加器进行投票累计。

（3）如果位移后的点超过了累加器矩阵的范围，则退出。

三、函数参数分析

1、核大小与高斯 sigma

求导核默认为 3；高斯核创建强调 sigma 非零，指定基于核的大小给定合适值，确保生成的高斯核足够平滑，同时又不至于过于模糊。

2、边缘检测的双门限

使用原梯度的 80%位数作为高门限，高门限的一半作为低门限，解决门限设置不好把握大小的问题，实现自适应。

3、霍夫空间格大小

小的 grid_size 的会增加计算量但提高精度，大的会减少计算量但降低精度。这里对 r 也设置了格大小，一定程度上能进一步减少计算量，但太大时会导致拟合出的圆与实际圆偏离的情况严重。

4、最大/最小半径、半径变化范围、最小圆心距离

依赖性高，适应能力和附近圆区分不强。

四、实验结果

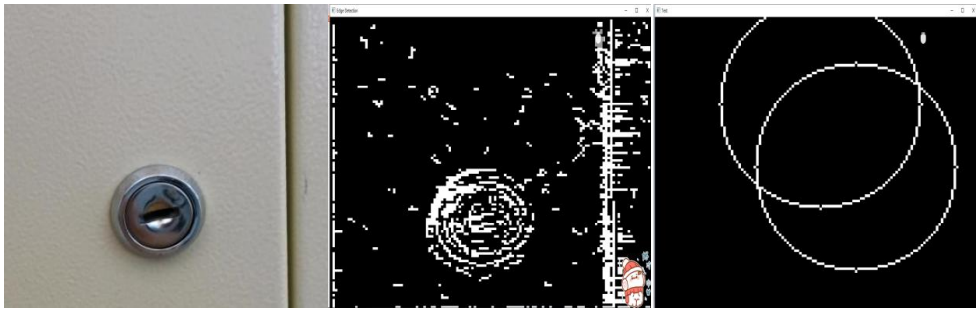
1、原始 Status1

grid_size = [edges.shape[0] // 50, edges.shape[1] // 50, 20]

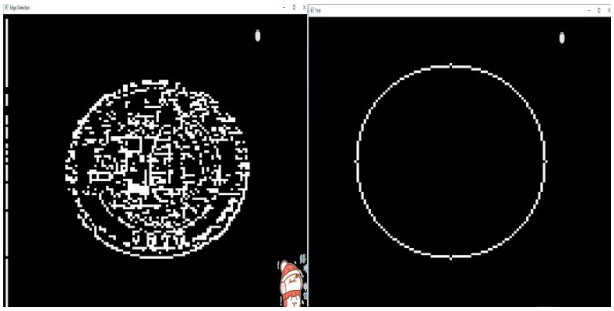
min_radius = 30, max_radius = 50, threshold = 50　# 累加器阈值

min_distance = 100　# 局部圆最小距离　param2 = 20　#圆的半径变化范围

（1）非硬币圆形图，高噪声



（2）单硬币图



（3）多硬币图

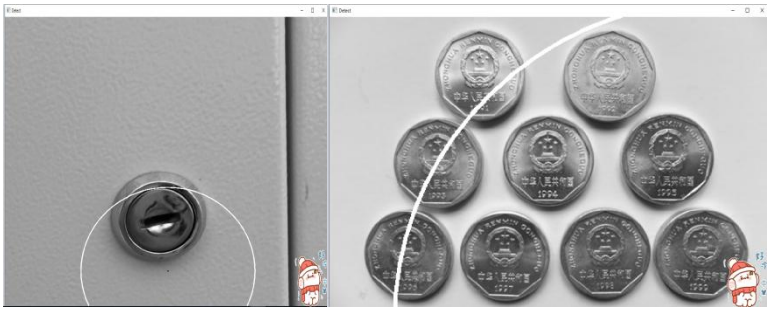
（无圆）

边缘提取效果较好；而圆拟合对最大/最小半径、最小圆心距离依赖性高，多硬币对累加器阈值也有更多要求，设置的 grid_size 会对多硬币检测情况产生影响。

2、对比

（1）原始 status1

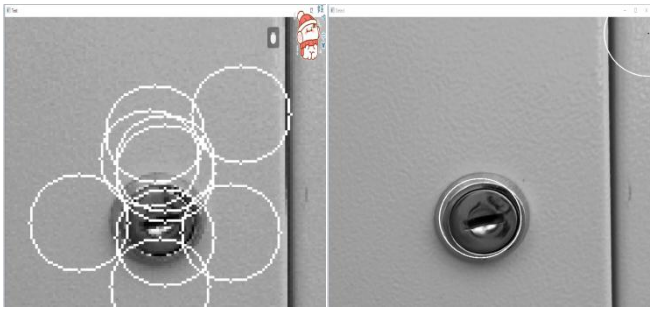
调用 cv2 库函数，在“非硬币圆形图，高噪声”和“多硬币”图像上做圆检测结果：



（2）调整参数对比两种实现

min_radius = 10, max_radius = 30

Mine（左） Cv2（右）



grid_size = [edges.shape[0] // 20, edges.shape[1] // 20, 20]

min_radius = 0, max_radius = 30, threshold = 20 # 累加器阈值

min_distance = 60 # 局部圆最小距离

Mine（左） Cv2（右）

发现自我编写检测圆的位置过图中圆，而调用 cv2 在同样参数条件下不过图中圆。

但 cv2 从时间上来讲具有绝对优势，在分辨率较低时，优势已经很明显：

Mine（上）Cv2（下）

```
[(34, 15, 10), (46, 51, 10), (78, 69, 10), (62, 75, 10)]
image.shape: (365, 530)
time: 10.693244695663452
>>>
================== RESTART: F:\课程\cv\detect_circle_via_cv2
==
330 190 68
90 290 5
time: 5.8527750968933105
>>>
```

其他参数不变改变 dp 为 1 时，cv2 能检测到圆而自我编写检测不到圆，说明应对高分辨率时依然存在问题。