

**计算机视觉实验报告**

**边缘检测**

班级：110

姓名：耿翊中

学号：2021213382

2024年3月

## Canny边缘检测

**1.1使用高斯滤波器，以平滑图像，滤除噪声**

1. **def** Gaussian\_ambiguity(self):
2. self.img = cv2.GaussianBlur(self.img, (self.Gaussian\_kernel, self.Gaussian\_kernel), self.Gaussian\_sd)
3. **return** self.img

**1.2计算图像中每个像素点的梯度强度和方向**

首先将图片转换为灰度图，然后使用sobel算子计算像素点的梯度强度和方向。

1. **def** get\_gradient(self):
2. # 转换图像为灰度
3. **if** len(self.img.shape) > 2:  # 如果不是灰度图
4. gray = cv2.cvtColor(self.img, cv2.COLOR\_BGR2GRAY)
5. **else**:
6. gray = self.img
7. # 使用cv2.Sobel获取X和Y方向的梯度
8. gradient\_x = cv2.Sobel(gray, cv2.CV\_64F, 1, 0, ksize=3)
9. gradient\_y = cv2.Sobel(gray, cv2.CV\_64F, 0, 1, ksize=3)
11. # 计算梯度的幅度和方向
12. self.magnitude = np.sqrt(gradient\_x \*\* 2 + gradient\_y \*\* 2)
13. self.angle = np.arctan2(gradient\_y, gradient\_x)  # 这里保持使用弧度

**1.3非极大值抑制**

非极大值抑制是一种边缘稀疏技术，非极大值抑制的作用在于“瘦”边。对图像进行梯度计算后，仅仅基于梯度值提取的边缘仍然很模糊。对于标准3，对边缘有且应当只有一个准确的响应。而非极大值抑制则可以帮助将局部最大值之外的所有梯度值抑制为0，对梯度图像中每个像素进行非极大值抑制的算法是：

将当前像素的梯度强度与沿正负梯度方向上的两个像素进行比较。

1. # 根据梯度方向，比较当前像素与邻域像素的梯度幅度
2. # 梯度方向为0度
3. **if** (0 <= angle[i, j] < 22.5) **or** (157.5 <= angle[i, j] <= 180):
4. q = self.magnitude[i, j + 1]
5. r = self.magnitude[i, j - 1]

如果当前像素的梯度强度与另外两个像素相比最大，则该像素点保留为边缘点，否则该像素点将被抑制。

1. # 只保留梯度方向上的局部最大值点
2. **if** self.magnitude[i, j] >= q **and** self.magnitude[i, j] >= r:
3. Z[i, j] = self.magnitude[i, j]
4. **else**:
5. Z[i, j] = 0

**1.4双阈值检测**

在施加非极大值抑制之后，剩余的像素可以更准确地表示图像中的实际边缘。然而，仍然存在由于噪声和颜色变化引起的一些边缘像素。为了解决这些杂散响应，必须用弱梯度值过滤边缘像素，并保留具有高梯度值的边缘像素，可以通过选择高低阈值来实现。

首先定义强边缘和若边缘：

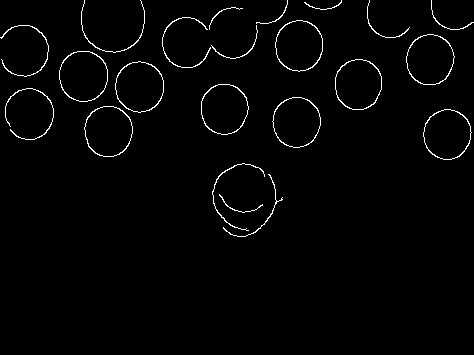
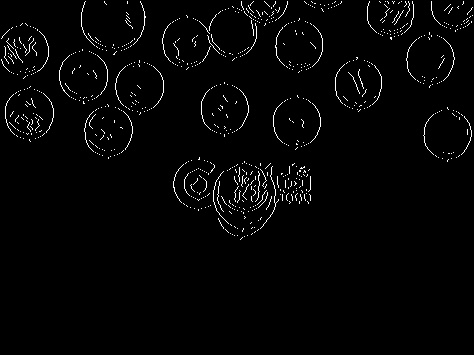
1. strong\_i, strong\_j = np.where(self.nms\_result >= self.high\_threshold)
2. weak\_i, weak\_j = np.where((self.nms\_result <= self.high\_threshold) & (self.nms\_result >= self.low\_threshold))

link\_edges是一个递归函数，用于检查一个给定的弱边缘点是否与强边缘点相连。如果一个弱边缘点周围有强边缘点，那么这个弱边缘点会被升级为强边缘点。递归的过程确保了所有与强边缘相连的弱边缘都会被检查和更新。

link\_edges函数通过检查当前点的周围8个像素（即8连通区域）来实现边缘连接。如果周围的像素中有被标记为强边缘的，当前的弱边缘点就会被转变为强边缘点。如果周围像素是未检查的弱边缘，会递归调用link\_edges进行检查。

1. **def** link\_edges(i, j):
2. **if** res[i, j] == 75 **and** **not** checked[i, j]:
3. # 标记当前弱边缘点为已检查
4. checked[i, j] = True
5. # 检查周围8个邻域像素
6. **for** x **in** range(max(0, i - 1), min(i + 2, M)):
7. **for** y **in** range(max(0, j - 1), min(j + 2, N)):
8. # 如果邻域像素中有强边缘，则当前弱边缘点也变为强边缘
9. **if** res[x, y] == 255:
10. res[i, j] = 255
11. **return**
12. # 否则，递归地应用相同的逻辑到邻域的弱边缘点
13. **elif** res[x, y] == 75:
14. link\_edges(x, y)

**1.5最终效果**



图左为自己实现的Canny检测效果，图右为使用opencv库实现的效果

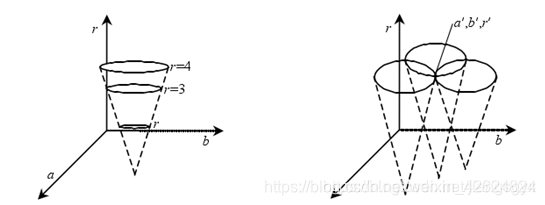
## **Hough变换**

**2.1使用累加器检测圆形**

圆一般方程为：

选择 a,b,r 作为参数，变换到参数空间后，即对应一个圆锥，圆锥方程为





故使用一个三维累加器来计算圆

累加器的定义和目的:

在霍夫变换中，累加器是用来记录图像中所有可能圆的投票结果的三维数组。其维度分别对应于圆心的x坐标、圆心的y坐标以及圆的半径。

在这个案例中，累加器的大小为(height, width, max\_radius - min\_radius)，其中height和width对应于输入图像的尺寸，而第三维max\_radius - min\_radius代表可能的半径范围。

1. accumulator = np.zeros((height, width, self.max\_radius - self.min\_radius))

对于图像中的每一个像素点(x, y)，如果它是一个边缘点（通常由边缘检测算法如Canny边缘检测确定），则对于每一个可能的半径r，以及一系列的角度theta（从0度到360度），使用下列公式来计算一个圆的假设圆心(a, b)：

1. a = int(x - r \* np.cos(np.deg2rad(theta)))
2. b = int(y - r \* np.sin(np.deg2rad(theta)))

每当计算出一个假设的圆心(a, b)时，累加器中对应于(b, a, r-min\_radius)的单元就会增加一个计数。这个过程称为“投票”，其中每个边缘点都为可能通过该点的所有圆“投票”。

**2.2NMS识别圆形**

在完成对所有边缘点的投票之后，累加器会填满了各种可能圆的投票结果。理论上，对于图像中真实存在的圆，其对应的圆心和半径在累加器中的值会显著地高于周围的值，因为许多边缘点会共同为这个圆的存在“投票”。

通过设置一个阈值，可以从累加器中筛选出高票数的元素，这些高票数对应的圆心位置和半径就是检测到的圆。但是，直接这样做可能会检测到很多接近的重复圆，因此需要进一步的处理如非最大值抑制(NMS)。

1. **def** apply\_nms(self, accumulator):
2. circles = []
3. height, width, \_ = accumulator.shape
4. **for** r **in** range(self.min\_radius, self.max\_radius):
5. **for** x **in** range(width):
6. **for** y **in** range(height):
7. **if** accumulator[y, x, r - self.min\_radius] > self.threshold:
8. **if** self.is\_local\_max(accumulator, x, y, r - self.min\_radius):
9. circles.append((x, y, r))
10. # 过滤相近的圆
11. filtered\_circles = []
12. **for** circle **in** circles:
13. x, y, r = circle
14. **if** all((x - cx) \*\* 2 + (y - cy) \*\* 2 > self.nms\_radius \*\* 2 **for** cx, cy, cr **in** filtered\_circles):
15. filtered\_circles.append(circle)
16. **return** filtered\_circles

**2.3最终效果**



图左为自己实现的Canny检测效果，图右为使用opencv库实现的效果