
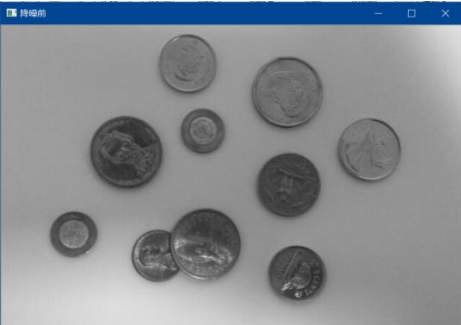


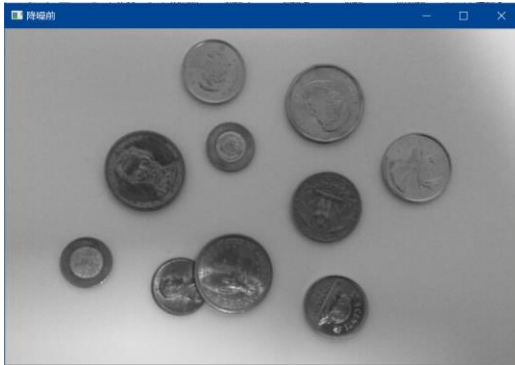
计算机视觉 课程实验报告

学号: 201900161140	姓名: 张文浩	2021.10.30
实验题目: 图像结构二: 霍夫变换		
<p>实验过程中遇到和解决的问题: (记录实验过程中遇到的问题, 以及解决过程和实验结果。可以适当配以关键代码辅助说明, 但不要大段贴代码。)</p> <p>实验要求:</p> <p>实现基于霍夫变换的图像圆检测 (边缘检测可以用 opencv 的 <code>canny</code> 函数), 并尝试对其准确率和效率进行优化实现</p> <p>Hough Transform for Circles</p> <p>For every edge pixel (x,y) :</p> <p>For each possible radius value r:</p> <p>For each possible gradient direction θ:</p> <p><i>// or use estimated gradient</i></p> <p>$a = x - r \cos(\theta)$</p> <p>$b = y + r \sin(\theta)$</p> <p>$H[a,b,r] += 1$</p> <p>end</p> <p>end</p> <p>实验步骤:</p> <p>第一步:</p> <p>读入图像, 转化为灰度图像。</p> <p>原图:  灰度图像: </p> <p>第二步:</p> <p>利用 opencv 自带的 <code>Canny</code> 函数进行边缘检测, 得到的边缘图像</p>		

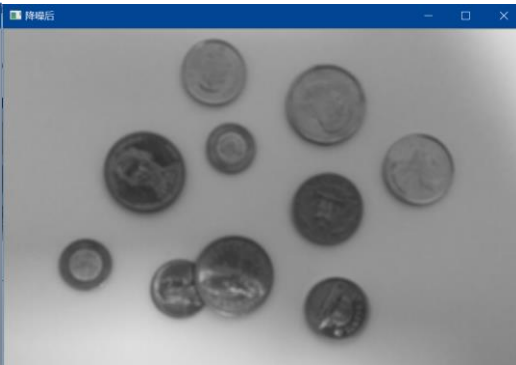
```
Canny(image_gray, image_edge, 70, 100, 3);
```

这里遇到的问题，输出得到的边缘图像发现，噪音非常多，不利于下一步的图形的检测，通过查阅资料，发现在进行边缘提取的时候往往会先对灰度图像进行降噪处理（高斯模糊），我尝试了之后发现先进行降噪处理边缘提取的效果好很多。

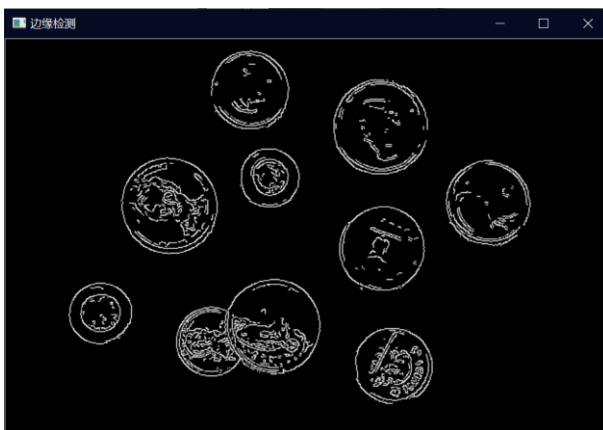
降噪前灰度图像：



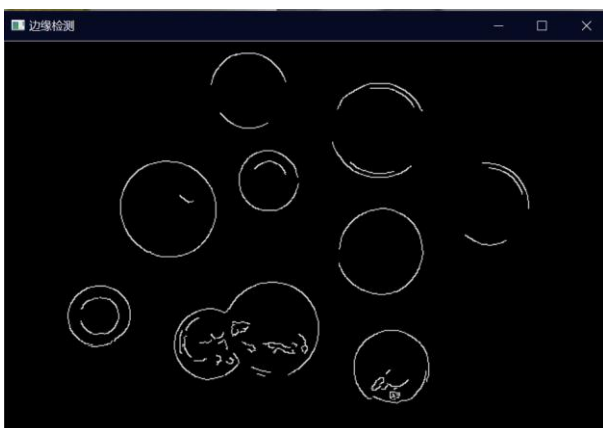
降噪后灰度图像：



不进行降噪处理得到的边缘



进行降噪处理得到的边缘



通过对比很容易发现进行降噪处理更有利于下一步图形的提取。

第三步：

根据 ppt 上的伪代码，构造霍夫变换函数。首先我们需要一个三维的 H 矩阵，

Hough Transform for Circles

For every edge pixel (x,y) :

For each possible radius value r :

For each possible gradient direction θ :

// or use estimated gradient

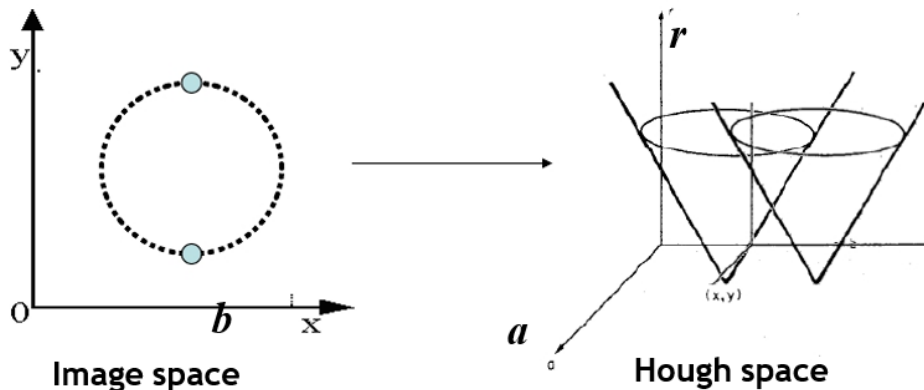
$$a = x - r \cos(\theta)$$

$$b = y + r \sin(\theta)$$

$$H[a,b,r] += 1$$

end

end



先用两个 for loop 遍历每个图像中每个位置，在两个 for 循环里面，还要再用两个 for loop 遍历所有可能的半径和 theta 角度取值，进行投票 vote。

```
a = i + r * cos(theta);  
b = j + r * sin(theta);  
if (a >= 0 && a < image_edge.rows && b >= 0 && b < image_edge.cols)  
    H[a][b][r]++;
```

现在我们有每个位置对应半径的票数，存在 H 三维数组中。用三个 for 循环遍历 H 数组的三个维度，如果对应位置和半径的 vote 投票数大于一个阈值 (threshold1)，其中这个阈值的设定不同，对结果的影响比较大，所以我后面用了一个 trackerbar 可以交互地调整阈值。

如果 vote 投票数大于这个阈值，就设定这个位置有一个圆，把圆心和圆的周边画出来。效果如下：



问题:

这里我发现效果不好的原因是因为出现了圆重叠的情况，导致有的线很粗。分析原因，是因为同一个圆因为存在误差的原因被当成了多个圆，而它们又靠的非常近，所以导致有的线会变得比较粗，效果不好。

解决思路:

如果当前位置投票数大于一个阈值，即被检测成了一个圆形位置，那么如果之后再检测到跟它位置很近且半径差不多的圆时，就忽略了，因为者很有可能是误差引起的将一个圆当作多个圆的情况。

实现:

新构造一个 `vis` 三维数组，维度大小与 `H` 数组相同，`vis[i][j][r]=1` 表示 (i, j) 这个位置半径为 r 的圆，之前已经检测出很相似的圆了，这个圆就不要画出来了。在根据 `H` 数组遍历画圆的三个 `for loop` 里面加一个判断即可。`vis` 数组维护的办法也比较简单，每画一个圆，就将这个圆周围的 `vis` 置为 1，“告诉”它们不要画了。

```
if (H[i][j][r] >= threshold1 && !vis[i][j][r]) {
    circle(dst, Point(j, i), 3, Scalar(0, 255, 0), -1, 8, 0);
    circle(dst, Point(j, i), r, Scalar(155, 50, 255), 2, 8, 0);
    for (int x0 = max(0, i - minDist); x0 < min(image_edge.rows, i + minDist);
x0++) {
        for (int y0 = max(0, j - minDist); y0 < min(image_edge.cols, j + minDist);
y0++) {
            for (int r0 = max(minr, r - 10); r0 < min(maxr, r + 10); r0++) {
                vis[x0][y0][r0] = 1;
            }
        }
    }
}
```

改进后的效果如下:

改进前:



对比发现，效果好了很多，没有再出现多个圆重叠的情况。

第四步：

修改阈值 threshold，观察对结果的影响

Threshold=200

threshold=300



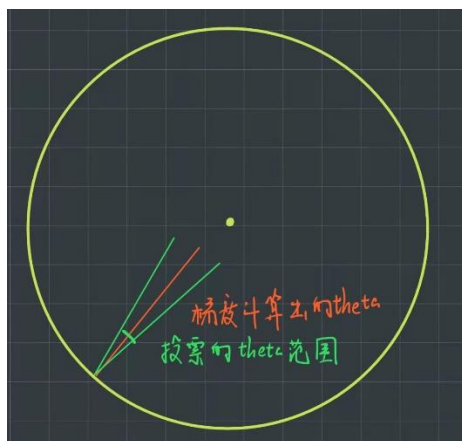
我发现 300 的阈值比 200 的少了一些误检的情况，越小的阈值会检测出更多的圆形，其中有可能是错误的，所以我们可以通过增大阈值的方法减少误检率。

但如果我们将阈值继续调大。发现只有少部分的圆被检测出来，所以阈值也不能太大，阈值过大会导致漏检过多的情况。



第五步：优化性能。

目前自己编写的 `hough` 函数的效果可以了，但是耗时非常长，每次计算耗时都会达到十秒左右，这非常令人无法接受，所以我考虑优化性能的方法。在 `hough` 函数中，可以分析出来比较消耗时间的地方是四个 `for loop`。发现其中对角度 `theta` 的遍历是可以进行优化的。方法是，通过计算图像的当前位置的梯度，直接得到到圆心的方向，就不用遍历 $0-2\pi$ 了。同时，为了减少因为梯度计算不准确和圆形本身不规则而带来的误差，我们把计算出来的 `theta` 角做一个泛化处理，并不只对求出来的 `theta` 方向上的点投票，而是用 `theta` 附近一个很小的范围。



关键代码如下：

```
for (double theta = -0.5; theta <= 0.5; theta = theta + 0.05) {
    a = int(i + r * cos(atan(dy.ptr<float>(i)[j] / dx.ptr<float>(i)[j]) + theta));
    b = int(j + r * sin(atan(dy.ptr<float>(i)[j] / dx.ptr<float>(i)[j]) + theta));
    if (a >= 0 && a < image_edge.rows && b >= 0 && b < image_edge.cols) {
        H[a][b][r] += 1;
    }
}
```

可以看到利用加速方法后的计算耗时减少了很多，起到了性能优化的作用。

myHough_fast算法用时： 395ms

看一下加速后的效果。



虽然加速后的版本在时间复杂度上完胜加速前的，但也可以看到效果也受到了一定的影响，所以最终在使用的时候需要在时间性能和效果上做一个权衡。

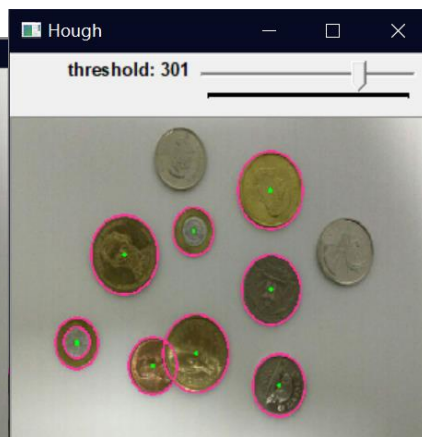
第六步：

与 opencv 的 HoughCircles 算法进行对比

HoughCircles:



myHough:



```
opencv的HoughCircles函数用时: 69ms  
start  
mid  
finish  
myHough算法用时: 10709ms
```

HoughCircles

myHough:



通过对比发现在效果上差不多，但是时间 HoughCircles 比 myHough 快了非常多。

第六步：

分析 opencv 源代码，学习 HoughCircles 使用的算法。

通过分析 HoughCircles 的源码，了解了效率高的原因。它使用的算法也是改进的霍夫变换——2-1 霍夫变换（21HT）。

也就是把霍夫变换分为两个阶段，从而减小了霍夫空间的维数。

①第一阶段用于检测圆心，②第二阶段从圆心推导出圆半径。

①检测圆心的原理是圆心是它所在圆周所有法线的交汇处，因此只要找到这个交点，即可确定圆心，该方法所用的霍夫空间与图像空间的性质相同，因此它仅仅是二维空间。

②检测圆半径的方法是从圆心到圆周上的任意一点的距离（即半径）是相同，只要确定一个阈值，只要相同距离的数量大于该阈值，我们就认为该距离就是该圆心所对应的圆半径，该方法只需要计算半径直方图，不使用霍夫空间。

圆心和圆半径都得到了，那么通过公式 1 一个圆形就得到了。

$$(x-a)^2+(y-b)^2=r^2 \quad (1)$$

从上面的分析可以看出，2-1 霍夫变换把标准霍夫变换的三维霍夫空间缩小为二维霍夫空间，因此无论在内存的使用上还是在运行效率上，2-1 霍夫变换都远远优于标准霍夫变换。但该算法有一个不足之处就是由于圆半径的检测完全取决于圆心的检测，因此如果圆心检测出现偏差，那么圆半径的检测肯定也是错误的。2-1 霍夫变换的具体步骤为：

第一阶段：检测圆心

1.1、对输入图像边缘检测；

1.2、计算图形的梯度，并确定圆周线，其中圆周的梯度就是它的法线；

1.3、在二维霍夫空间内，绘出所有图形的梯度直线，某坐标点上累加和的值越大，说明在该点上直线相交的次数越多，也就是越有可能是圆心；

1.4、在霍夫空间的 4 邻域内进行非最大值抑制；

1.5、设定一个阈值，霍夫空间内累加和大于该阈值的点就对应于圆心。

第二阶段：检测圆半径

2.1、计算某一个圆心到所有圆周线的距离，这些距离中就有该圆心所对应的圆的半径的值，这些半径值当然是相等的，并且这些圆半径的数量要远远大于其他距离值相等的数量；

2.2、设定两个阈值，定义为最大半径和最小半径，保留距离在这两个半径之间的值，这意味着我们检测的圆不能太大，也不能太小；

2.3、对保留下来的距离进行排序；

2.4、找到距离相同的那些值，并计算相同值的数量；

2.5、设定一个阈值，只有相同值的数量大于该阈值，才认为该值是该圆心对应的圆半径；

2.6、对每一个圆心，完成上面的 2.1~2.5 步骤，得到所有的圆半径。

HoughCircles 函数的原型为：

```
void HoughCircles(InputArray image,OutputArray circles,int method,double dp,double minDist, double param1=100,double param2=100,int minRadius=0,int maxRadius=0 )
```

image 为输入图像，要求是灰度图像

circles 为输出圆向量，每个向量包括三个浮点型的元素——圆心横坐标，圆心纵坐标和圆半径

method 为使用霍夫变换圆检测的算法，Opencv2.4.9 只实现了 2-1 霍夫变换，它的参数是 CV_HOUGH_GRADIENT

dp 为第一阶段所使用的霍夫空间的分辨率，dp=1 时表示霍夫空间与输入图像空间的大小一致，dp=2 时霍夫空间是输入图像空间的一半，以此类推

minDist 为圆心之间的最小距离，如果检测到的两个圆心之间距离小于该值，则认为它们是同一个圆心

param1 为边缘检测时使用 Canny 算子的高阈值

param2 为步骤 1.5 和步骤 2.5 中所共有的阈值

minRadius 和 maxRadius 为所检测到的圆半径的最小值和最大值

结果分析与体会：

在本次实验中，我实现了基于霍夫变换的图像圆检测。在实验中遇到了将同一个圆检测为多个非常相似的圆的问题。并利用一个 `vis` 三维数组解决了这个问题，优化了实现的效果。

在性能优化方面，我利用图像梯度直接得到了 `theta` 方向，减少了对 `theta` 角度的遍历，大大提升了性能，降低了时间复杂度。

通过对比与 `opencv` 的 `HoughCircles` 函数的源代码，发现 `HoughCircles` 的计算效率比自己的写的方法高了很多，于是了解了 `HoughCircles` 的实现算法的思想和步骤。