

作业 2: 边缘检测和边缘链接

姓名 王明 学号 161220124 邮箱 1090616897@qq.com 联系方式 13851085654

(南京大学 计算机科学与技术系, 南京 210093)

1 实现细节

1.1 边缘检测

1.1.1 一阶边缘检测算子

1.1.1.1 综述

边缘检测强调的是图像对比度。检测对比度（亮度）上的差别，可以增强图像中的边界特征，这些边界出现正是图像亮度上的差别。亮度变化可以通过对相邻点进行差分处理来增强，而一阶边缘检测算子通过计算某个方向上的微分使边缘的变化增强，从而体现出亮度的梯度变化，进而检测出边缘。

对于某个点 (x,y) 计算其梯度，事实上一个点的周围 360 度都可以有梯度，但是考虑到图像的二维性以及计算的方便度，我们一般计算水平方向和垂直方向上的梯度，然后取两者的平方和再开根，这个值就是该点的梯度。示意如下：

$$\nabla f \equiv \text{grad}(f) \equiv \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \quad M(x, y) = \text{mag}(\nabla f) = \sqrt{g_x^2 + g_y^2}$$

计算好梯度之后，设定一个阈值，梯度高于此阈值的灰度为 1，低于此阈值的灰度为 0，这样就得到了边缘检测的二值图。

一阶边缘检测算子主要包含 Robert、Priwitt、Sobel、Canny。

1.1.1.2 Robert 算子

原理介绍：

3*3 的模板

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

Robert 水平方向的模板

-1	0
0	1

Robert 垂直方向的模板

0	-1
1	0

Robert 算子采用 2*2 的模板，水平方向的梯度计算、垂直方向的梯度计算如下：

$$g_x = (z_9 - z_5)$$
$$g_y = (z_8 - z_6)$$

代码实现：

```

%robert
%设置阈值
output_robert = zeros(row, col);
%遍历图像的每个像素，用Robert算子计算梯度
robert_threshold = 0.15;
for r = 1 : row - 1
    for c = 1 : col - 1
        robert_x = 1 * input_image(r, c) - 1 * input_image(r + 1, c + 1);
        robert_y = 1 * input_image(r, c + 1) - 1 * input_image(r + 1, c);
        robert_num = sqrt(robert_x^2 + robert_y^2);
        %将梯度与阈值进行比较，决定该点是0还是1
        if(robert_num < robert_threshold)
            output_robert(r,c) = 0;
        else
            output_robert(r,c) = 1;
        end
    end
end
end

```

1.1.1.3 Priwitt 算子

原理介绍：

3*3 的模板

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

Priwitt 水平方向的模板

-1	-1	-1
0	0	0
1	1	1

Priwitt 垂直方向的模板

-1	0	1
-1	0	1
-1	0	1

Priwitt 算子采用 3*3 的模板，水平方向的梯度计算、垂直方向的梯度计算如下：

$$g_x = \frac{\partial f}{\partial x} = (z_7 + z_8 + z_9) - (z_1 + z_2 + z_3)$$

$$g_y = \frac{\partial f}{\partial y} = (z_3 + z_6 + z_9) - (z_1 + z_4 + z_7)$$

代码实现：

```

%priwitt
%设置阈值
output_priwitt = zeros(row, col);
%遍历图像的每个像素，用Priwitt算子计算梯度
priwitt_threshold = 0.55;
for r = 2 : row - 1
    for c = 2 : col - 1
        priwitt_x = - 1 * input_image(r - 1, c - 1) + 1 * input_image(r - 1, c + 1)
        priwitt_y = - 1 * input_image(r - 1, c - 1) - 1 * input_image(r - 1, c) - 1
        priwitt_num = sqrt(priwitt_x^2 + priwitt_y^2);
        %将梯度与阈值进行比较，决定该点是0还是1
        if(priwitt_num < priwitt_threshold)
            output_priwitt(r,c) = 0;
        else
            output_priwitt(r,c) = 1;
        end
    end
end
end

```

1.1.1.4 Sobel 算子

原理介绍：

3*3 的模板

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

Sobel 水平方向的模板

-1	-2	-1
0	0	0
1	2	1

Sobel 垂直方向的模板

-1	0	1
-2	0	2
-1	0	1

Sobel 算子采用 3*3 的模板，水平方向的梯度计算、垂直方向的梯度计算如下：

$$g_x = \frac{\partial f}{\partial x} = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)$$

$$g_y = \frac{\partial f}{\partial y} = (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)$$

代码实现：

```
%sobel
output_sobel = zeros(row, col);
%设置阈值
sobel_threshold = 0.8;
%遍历图像的每个像素，用Sobel算子计算梯度
for r = 2 : row - 1
    for c = 2 : col - 1
        sobel_x = - 1 * input_image(r - 1, c - 1) + 1 * input_image(r - 1, c + 1)
        sobel_y = 1 * input_image(r - 1, c - 1) + 2 * input_image(r - 1, c) + 1 *
        sobel_num = sqrt(sobel_x^2 + sobel_y^2);
        %将梯度与阈值进行比较，决定该点是0还是1
        if(sobel_num < sobel_threshold)
            output_sobel(r,c) = 0;
        else
            output_sobel(r,c) = 1;
        end
    end
end
```

1.1.1.5 Canny 算子

原理介绍：

canny 边缘检测算子是一种多级检测算法，具有以下优点：

- 1、低错误率，应该是边缘的能判别为边缘，不是边缘的能判别不是边缘。
- 2、最优定位，检测到的边缘点应该位于边缘的中心。
- 3、图像中的边缘只被标记一次，且在噪声图像中不产生为边缘。

canny 边缘检测分为以下步骤：

- 1、高斯平滑：用高斯过滤器进行图像降噪。
- 2、梯度和方向计算：理论上应当用 4 个梯度算子来计算水平、垂直、2 个对角线 4 个方向的梯度，但是为了计算方便，一般只需用 Robert、Priwitt 或者 Sobel 来计算梯度即可。
- 3、非最大值抑制：该步来剔除那些一定不是边缘的点。对于某个点，根据他的方向，用他的梯度和他方向上的 2 个邻居点的梯度进行比较，如果他不是最大的，那一定不是边缘点，直接将其灰度值设

为 0。

- 4、双阈值检测：设定一个高阈值和一个低阈值，大于高阈值被认为是边缘，设置灰度为 1，低于低阈值的被认为不是边缘，设置灰度为 0，剩下的待定。
- 5、滞后边界跟踪：对于未确定的像素，查看他的 8 邻域是否有强边缘，若有则认为他是边缘，否则认为不是边缘。

代码实现：

```
%高斯平滑处理
function [output_image] = gaussian_blur(input_image)
[ row,col] = size(input_image);
gaussian_operator = [1 4 7 4 1;
                    4 16 26 16 4;
                    7 26 41 26 7;
                    4 16 26 16 4;
                    1 4 7 4 1];

%用高斯算子进行卷积
output_image = conv2(input_image, gaussian_operator, 'same') / 273;
for r = 1 : row
    for c = 1 : col
        if(output_image > 1)
            output_image = 1;
        end
    end
end
end
```

```
%用Sobel算子计算梯度
sobel_tmpx = refined_image;
sobel_tmpy = refined_image;
sobel_tmpnum = zeros(row, col);
for r = 2 : row - 1
    for c = 2 : col - 1
        sobel_tmpx(r, c) = - 1 * refined_image(r - 1, c - 1) + 1 * refined_image(r - 1, c + 1) + 1 * refined_image(r + 1, c - 1) + 1 * refined_image(r + 1, c + 1);
        sobel_tmpy(r, c) = 1 * refined_image(r - 1, c - 1) + 2 * refined_image(r - 1, c) + 1 * refined_image(r - 1, c + 1) - 1 * refined_image(r + 1, c - 1) - 2 * refined_image(r + 1, c) - 1 * refined_image(r + 1, c + 1);
        sobel_tmpnum(r, c) = sqrt(sobel_tmpx(r, c)^2 + sobel_tmpy(r, c)^2);
    end
end
```

```
%利用梯度和方向进行非极大值抑制，去掉非极大值的不是边缘的点
for r = 2 : row - 1
    for c = 2 : col - 1
        sobel_x = sobel_tmpx(r, c);
        sobel_y = sobel_tmpy(r, c);
        %计算方向
        if (sobel_y ~= 0)
            angle = atan(sobel_y / sobel_x);
        elseif (sobel_y == 0 && sobel_x > 0)
            angle = pi / 2;
        else
            angle = - pi / 2;
        end
        %根据方向得到邻居的坐标
        if ((mod(angle,pi * 2) >= 15 * pi / 8 && mod(angle,pi * 2) <= 2 * pi) || (mod(angle,pi * 2) >= 3 * pi / 8 && mod(angle,pi * 2) <= 5 * pi / 8))
            r1 = r; c1 = c - 1; r2 = r; c2 = c + 1;
        elseif ((mod(angle,pi * 2) >= pi / 8 && mod(angle,pi * 2) <= 3 * pi / 8) || (mod(angle,pi * 2) >= 5 * pi / 8 && mod(angle,pi * 2) <= 7 * pi / 8))
            r1 = r - 1; c1 = c + 1; r2 = r + 1; c2 = c - 1;
        elseif ((mod(angle,pi * 2) >= 3 * pi / 8 && mod(angle,pi * 2) <= 5 * pi / 8) || (mod(angle,pi * 2) >= 7 * pi / 8 && mod(angle,pi * 2) <= 9 * pi / 8))
            r1 = r - 1; c1 = c; r2 = r + 1; c2 = c;
        else
            r1 = r - 1; c1 = c - 1; r2 = r + 1; c2 = c + 1;
        end
        %比较其与邻居的梯度，若不是极大值则抑制
        if (sobel_tmpnum(r, c) > sobel_tmpnum(r1, c1) && sobel_tmpnum(r, c) > sobel_tmpnum(r2, c2))
            output_canny(r, c) = refined_image(r, c);
        else
            output_canny(r, c) = 0;
        end
    end
end
end
```

```

%设定高低两个阈值，进行双阈值检测
tmpimage = output_canny;
sort1s = sort(tmpimage(:));
number = floor(0.995 * row * col);
canny_upthreshold = sort1s(number);
canny_dowthreshold = 0.5 * canny_upthreshold;
%canny_upthreshold = 0.77;
%canny_dowthreshold = 0.4 * canny_upthreshold;
for r = 1 : row
    for c = 1 : col
        if(output_canny(r,c) ~= 0)
            %根据阈值情况判定是边缘、不是边缘、待定边缘
            if(sobel_tmpnum(r,c) > canny_upthreshold)
                output_canny(r,c) = 1;
            elseif(sobel_tmpnum(r,c) < canny_dowthreshold)
                output_canny(r,c) = 0;
            end
        end
    end
end
end

```

```

%滞后边界跟踪，对待定的点进行判定
tmp_image = output_canny;
near = [-1 -1; -1 0; -1 1; 0 -1; 0 1; 1 -1; 1 0; 1 1];
for r = 1 : row
    for c = 1: col
        if(tmp_image(r,c) ~= 0 && tmp_image(r,c) ~= 1)
            %8邻域内有边缘点则认定该点也为边缘
            for pos = 1 : 8
                x1 = r + near(pos, 1);
                y1 = c + near(pos, 2);
                if(tmp_image(x1,y1) == 1)
                    output_canny(r, c) = 1;
                    break;
                end
            end
        end
    end
end
end
end

```

1.1.2 二阶边缘检测算子

1.1.2.1 综述

对应于一阶微分取极值点（即局部梯度最大的点），该点的二阶微分应该为 0，因此二阶边缘检测算子是通过计算二阶导数为 0 的方法来确定局部梯度最大的点。而采用二阶微分为零的优点是：确定零点的位置比确定极值点要容易得多，也精确得多。

比较常见的二阶微分算子是 Laplacian 算子，在此基础上优化出了 LoG 算子（Laplacian of Gaussian），它将拉普拉斯锐化滤波器和高斯平滑滤波器结合起来，先平滑掉噪声，再进行边缘检测，效果相对更好。

1.1.2.2 LoG 算子（Laplacian of Gaussian）

原理介绍：

通过拉普拉斯变换求二阶导， 计算如下：

$$\begin{aligned}
 \nabla^2 f(x,y) &= \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \\
 &= \{f(x+1,y) + f(x-1,y) - 2f(x,y)\} + \{f(x,y+1) + f(x,y-1) - 2f(x,y)\} \\
 &= f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y)
 \end{aligned}$$

此时得到的 Laplacian 算子的模板如右边所示：

1	1	1
1	-8	1
1	1	1

单纯使用 Laplacian 算子对于噪声的敏感度很高，因此而为了得到更好的效果，我使用了 9*9 的 LoG 算子。LoG 算子将拉普拉斯锐化滤波器和高斯平滑滤波器结合起来，对于噪声的抵抗更好。

代码实现：

```
%log
output_log = zeros(row, col);
%设置阈值
log_threshold = 9;
%生成9*9的LoG算子
log_operator = [0 1 1 2 2 2 1 1 0;
                1 2 4 5 5 5 4 2 1;
                1 4 5 3 0 3 5 4 1;
                2 5 3 -12 -24 -12 3 5 2;
                2 5 0 -24 -40 -24 0 5 2;
                2 5 3 -12 -24 -12 3 5 2;
                1 4 5 3 0 3 5 4 1;
                1 2 4 5 5 5 4 2 1;
                0 1 1 2 2 2 1 1 0];
%遍历图像用LoG算子求卷积，并根据阈值判定边缘
get_model = conv2(input_image, log_operator, 'same');
for r = 1 : row
    for c = 1 : col
        if(get_model(r, c) < log_threshold)
            output_log(r,c) = 0;
        else
            output_log(r,c) = 1;
        end
    end
end
end
```

1.2 边缘链接

1.2.1 综述

由于边缘检测之后生成的二值图存在噪声、间断的边界、不确切的位置和方向等各种问题，因此需要边缘链接算法来对边缘进行处理，从而形成清晰的边缘。

我针对边缘间断、多边缘重复两种情况设计了两种不同的边缘链接算法。

1.2.2 针对边缘间断的边缘链接

原理介绍：

往往一些效果不好的边缘检测算法得到很多间断的边缘，这时就需要将间断的边缘链接起来。算法的思路很简单也很常规：从某个起始点开始，对于周围半径为 1 的一圈按照一定的顺序进行访问，如果有灰度值为 1 的点，则标记，作为下一个开始点；如果没有则访问周围半径为 2 的一圈，如此不断加大半径直到找到下一个点。就这样一个接一个直到回到起始点则算法结束，被标记的所有点便是找到的边缘的路径。

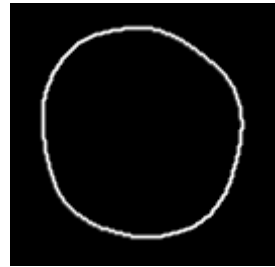
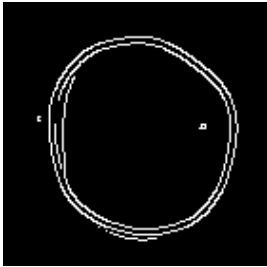
代码实现（由于代码较长，这里就截取算法核心部分）：

```
while(ending == 0 && find == 0)
    %半径每次加1
    dis = dis + 1;
    getpath = [];
    %得到周围一圈的路径，放入getpath中
    for j = col - dis : col + dis...
        for i = row - dis + 1 : row + dis - 1...
            for j = col + dis : -1 : col - dis...
                for i = row + dis - 1 : -1 : row - dis + 1...
                    if size(getpath) == [0,0]
                        ending = 1;
                        break;
                    end
                    [len, wid] = size(getpath);
                    %按次序访问周围一圈路径上的点，若有灰度为1的点则标记为下一个开始点，若没有则扩大搜索半径
                    for i = 1 : len...
                        end
```

1.2.3 针对多边缘重叠的边缘链接

原理介绍：

有时候边缘检测算法检测出来的边缘是完整的，但是可能会出现重复的情况，比如实验要求中的示例，左上角的橡皮圈很容易检测出来同心圆（如左图），而此算法的目标是把它变成单边缘的（如右图）。



具体的方法就是引入一个方向变量，每次都从这个方向开始规定一个顺序，这样对于在同一圈像素中得到 2 个满足的点就能做出正确的选择，而不会走入其他错误的边缘中去。每次找到点之后会更新方向，是一种带反馈的算法。算法的结束条件同样是回到起始点。关于方向的确定使用一下的规则：

对 8 邻域而言，初始值 $\text{dir} = 7$

按照逆时针顺序搜索当前像素的 3*3 邻域，，计算 dir ：

$(\text{dir}-k+7) \bmod 8$ (dir 为偶数)

$(\text{dir}-k+6) \bmod 8$ (dir 为奇数)

代码实现（由于代码较长，这里就截取算法核心部分）：

%按照方向dir得到周围一圈的访问顺序

```
path = [0 1; -1 1; -1 0; -1 -1; 0 -1; 1 -1; 1 0; 1 1];
```

```
getpath = [];
```

```
for i = dir + 1 : 8...
```

```
if dir > 0
```

```
    for j = 1 : dir...
```

```
end
```

```
if size(getpath) == [0,0]
```

```
    ending = 1;
```

```
end
```

```
[len, wid] = size(getpath);
```

%取第一个访问到的点为下一个点，并且更新方向

```
for i = 1 : len
```

```
    x = row + getpath(i, 1);
```

```
    y = col + getpath(i, 2);
```

```
    if(binary_image(x, y) == 1)
```

```
        row_tmp = x;
```

```
        col_tmp = y;
```

```
        dir = mod(dir + i - 1, 8);
```

```
        if mod(dir, 2) == 0
```

```
            dir_tmp = mod(dir + 7, 8);
```

```
        else
```

```
            dir_tmp = mod(dir + 6, 8);
```

```
        end
```

```
        find = 1;
```

2 结果

2.1 实验设置

输入：png 彩色图像（带有噪声的以及没有噪声的）。

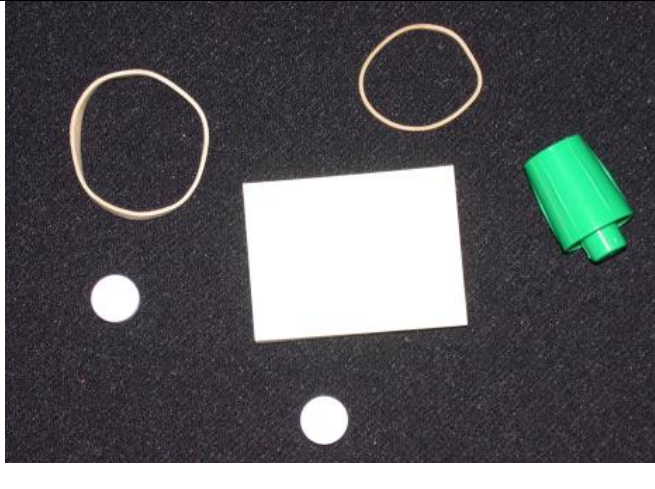
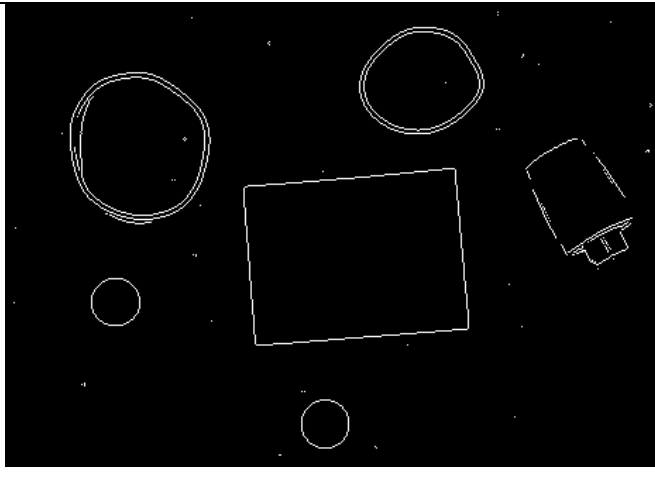
输出：png 二值图像（边缘检测后的二值图像（对于 rubberband_cap 这幅图输出还包含边缘链接图））。

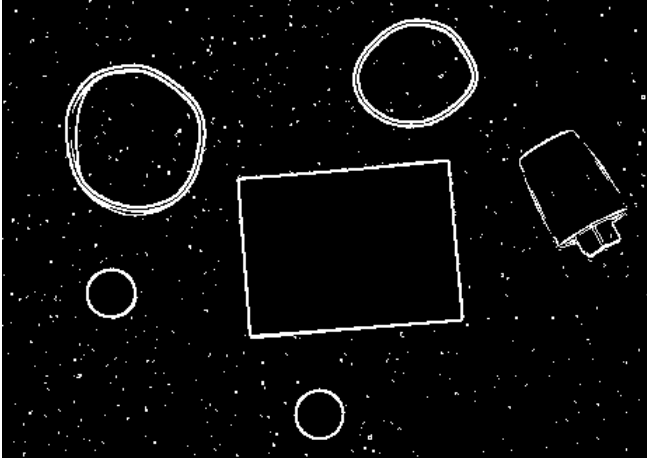
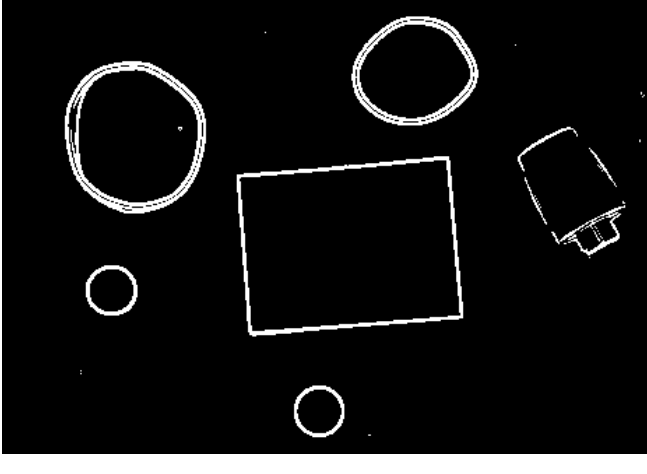
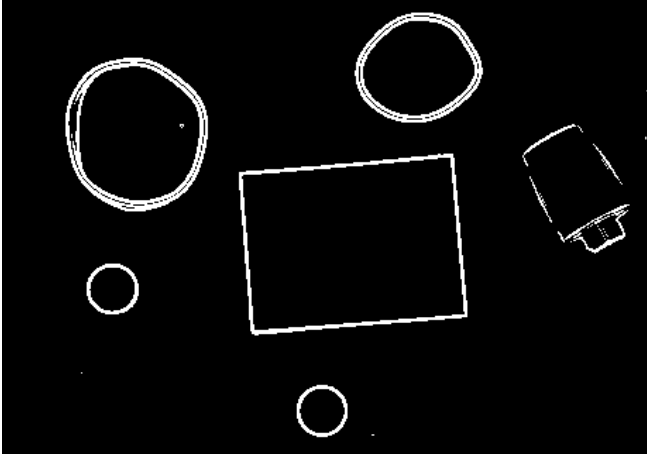
额外说明：对于每幅图像，都采用 matlab 自带算法、robert 算法、priwitt 算法、sobel 算法、log 算法、canny 算法等 6 种算法进行边缘检测（有部分效果不好的图片被去掉了）。rubberband_cap 采用了 2 种不同的方法进行了边缘链接。

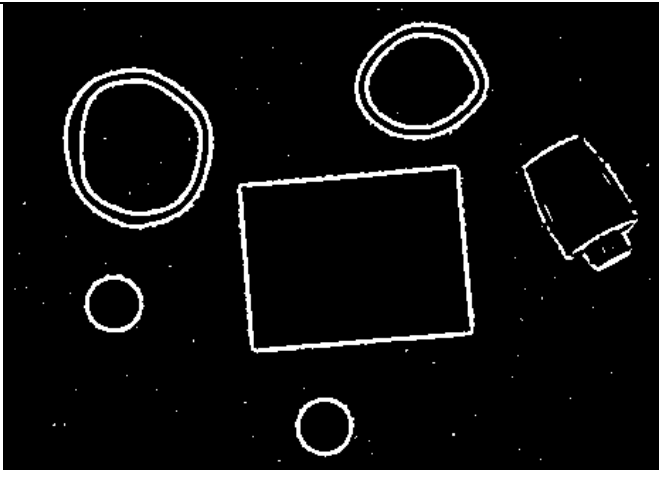
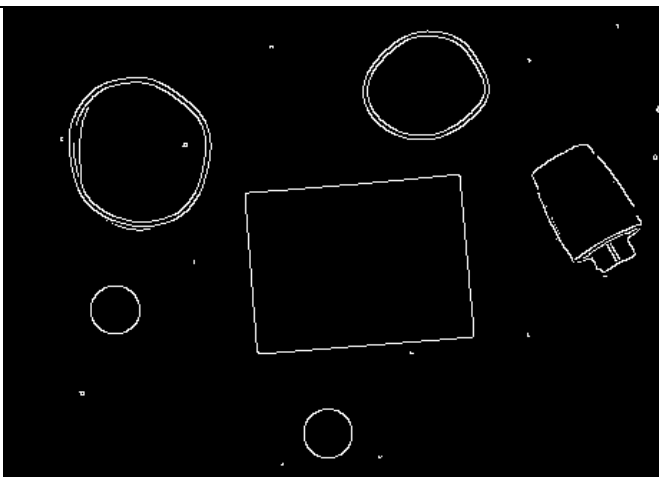
2.2 实验结果

2.2.1 边缘检测


2.2.1.1 rubberband_cap.png




输入		<p>原图</p> <p>主体部分比较简单，形状较为规整，但是背景带来了很多的噪声。</p>
输出 1		<p>matlab 自带算法</p> <p>对于噪声有很好的处理，检测出来的边缘也是相当的清晰，对于大圆有多边缘的现象，以及对于杯子出现了边缘的断裂。</p>




输出 2		<p>Robert 算法</p> <p>对于边缘的检测还是可以的，轮廓比较清晰，但是对于噪声没有很好地去除。</p>
输出 3		<p>Priwitt 算法</p> <p>去除噪声的能力很强，边缘检测效果也很好，就是得到的边缘较粗。</p>
输出 4		<p>Sobel 算法</p> <p>去除噪声的能力很强，边缘检测效果也很好，但也出现粗边缘和多边缘的问题。</p>

输出 5		<p>LoG 算法</p> <p>噪声点还是挺多的，检测出的边缘虽然很清晰，杯子那边也没有发生太多的间断，但是边缘同样是粗边缘。</p>
输出 6		<p>Canny 算法</p> <p>效果是相当好，不仅对于噪声有很强的抵抗力，检测出的边缘也是单像素的细边缘，对于杯子的轮廓检测效果也是几个图中最好的。其效果甚至比 matlab 自带的边缘检测算法要好。</p>


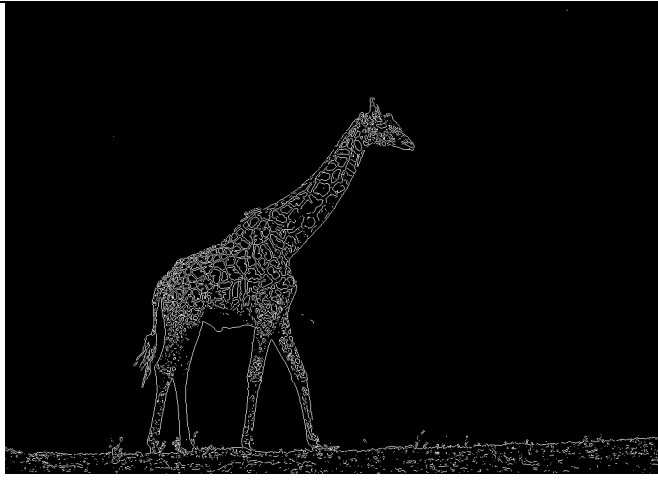
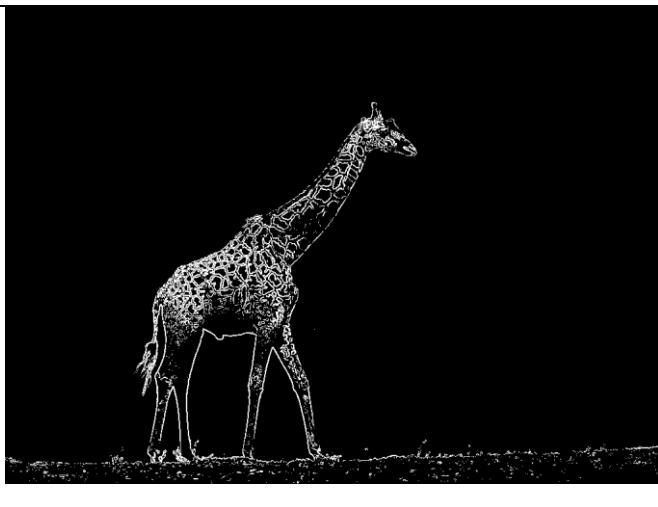
2.2.1.2 bird.png

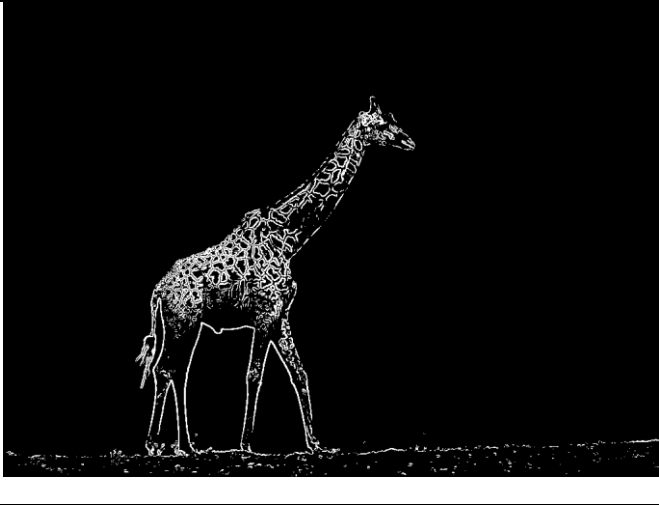
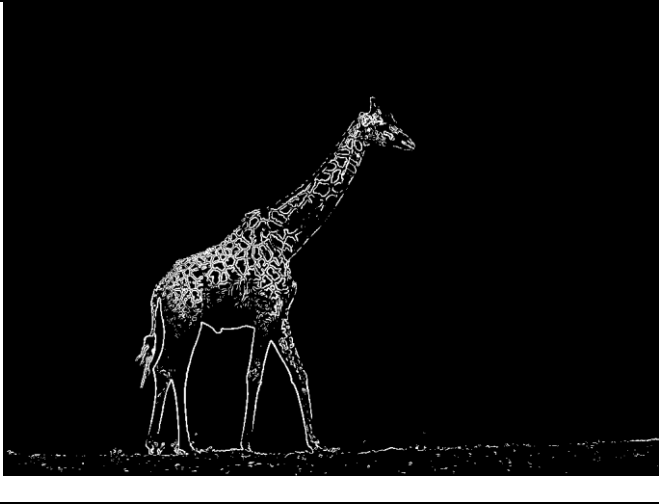
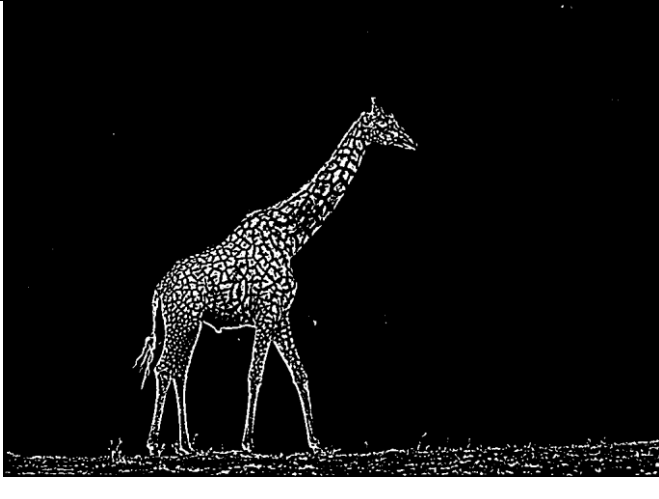
输入		<p>原图</p> <p>主体部分鸟的结构相当复杂，轮廓也不是很清晰，背景的白色噪声点也是挺多的。</p>
----	---	---

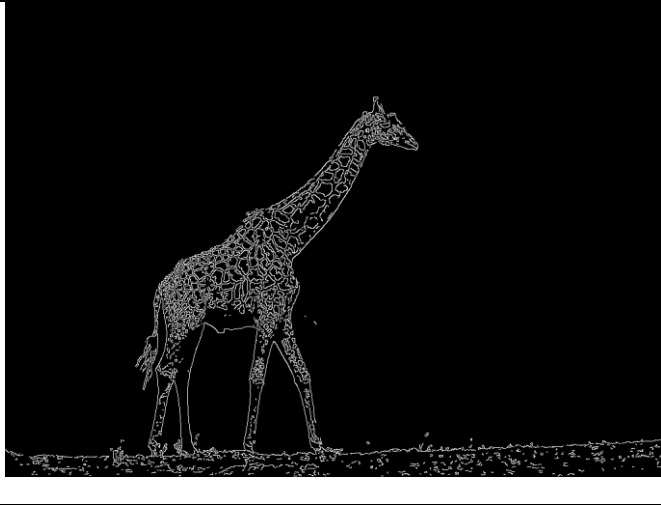

输出 1		<p>matlab 自带算法</p> <p>对于噪声有很好的处理，但是检测出来的边缘不尽如人意，鸟翅膀的外侧边缘检测得还是很不错的，但是翅膀内部直接就没有了。</p>
输出 2		<p>Robert 算法</p> <p>去噪效果也还行，对于边缘的检测还是可以的，轮廓比较清晰，鸟左半部分的检测还是挺好的，但是右翅膀的内部还是有部分未检测出来</p>
输出 3		<p>Priwitt 算法</p> <p>去噪效果还行，边缘检测效果一般，边缘间断的地方较多。</p>

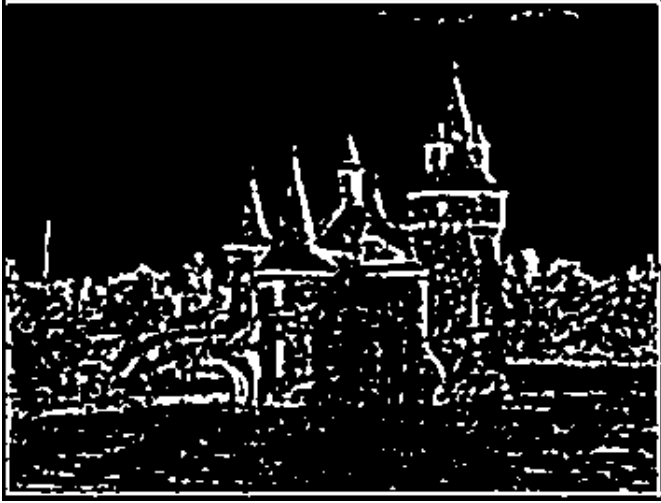

输出 4		<p>Sobel 算法</p> <p>去除噪声的能力还可以，边缘检测对于鸟翅膀的效果也不是很好</p>
输出 5		<p>LoG 算法</p> <p>噪声还是不少，边缘的检测细节做得挺好，尤其左翅膀的细节很清晰，就是边缘太粗了。</p>
输出 6		<p>Canny 算法</p> <p>抗噪声能力很强，检测出的边缘较细，对于左翅膀的细节还是检测得挺好的，总体来讲比 matlab 自带算法的效果好一些。</p>

2.2.1.3 giraffe.png


输入	 A photograph of a giraffe standing on a grassy plain under a blue sky with scattered white clouds. The giraffe is facing right, and its distinctive brown and white patterned coat is clearly visible.	<p>原图</p> <p>背景的噪声看上去是不多的，但是长颈鹿的花纹的边缘是相当复杂的。</p>
输出 1	 The image shows the edge detection result of the original giraffe image using MATLAB's built-in algorithm. The background is black, and the edges of the giraffe and the ground are highlighted in white. The edges are very sharp and clear, capturing the complex patterns of the giraffe's coat.	<p>matlab 自带算法</p> <p>对于噪声有很好的处理，检测出来的边缘也是相当的清晰，整体上的效果是很好的，毕竟系统自带函数，就是厉害。</p>
输出 2	 The image shows the edge detection result of the original giraffe image using the Robert algorithm. The background is black, and the edges of the giraffe and the ground are highlighted in white. Compared to the MATLAB result, the edges are less sharp and clear, especially in the neck and leg areas.	<p>Robert 算法</p> <p>对于边缘的检测效果一般，可以看到长颈鹿的脖子下部和后腿的部分花纹检测得不太好。</p>




输出 3		<p>Priwitt 算法</p> <p>对于边缘的检测效果一般，可以看到长颈鹿的脖子下部和后腿的部分花纹检测得不太好。</p>
输出 4		<p>Sobel 算法</p> <p>对于边缘的检测效果一般，可以看到长颈鹿的脖子下部和后腿的部分花纹没有检测出来，同时边缘相对较粗。</p>
输出 5		<p>LoG 算法</p> <p>边缘检测的效果很好，每一处的细节都能体现出来，就是由于边缘线较粗，以至于长颈鹿的部分花纹感觉像是在填充。</p>

输出 6		<p>Canny 算法</p> <p>效果还是很不错的，比较接近于 matlab 自带算法的效果，每一处的细节也处理的不错，是可以接受的效果。</p>
<p>2.2.1.4 noise.png</p> <p>输入</p>		<p>原图</p> <p>主体部分的灰度变化就不明显，边缘很模糊，以及满屏的白色噪声点，简直是灾难级的图片。</p>
输出 1		<p>matlab 自带算法</p> <p>果然就连 matlab 自带算法也吃不消了，除了城堡塔尖部位还能测出连续的边缘，城堡底部几乎就是离散的间断的点。</p>


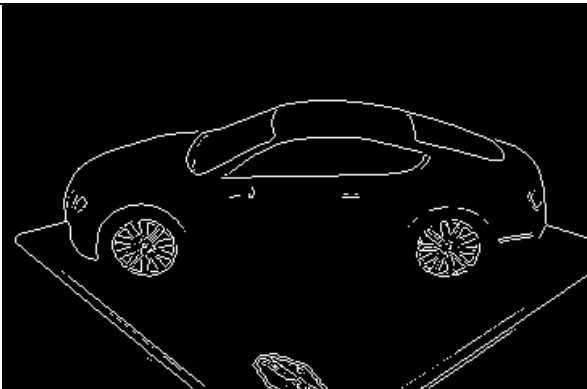
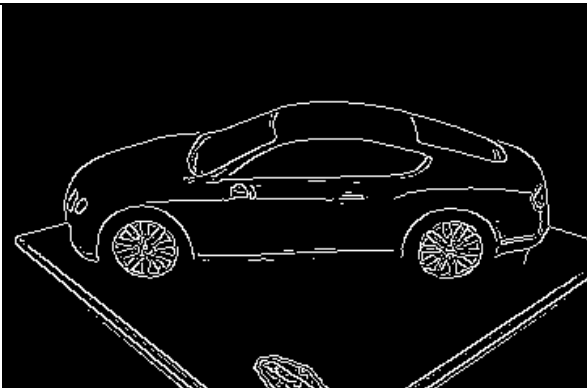
输出 2		<p>LoG 算法</p> <p>整体的城堡的边缘检测得相对来说好太多了，但是就是边缘非常粗，以及间断点还是很多。</p>
输出 3		<p>Canny 算法</p> <p>说实话，对于阈值的设定调过无数次，试图能得到好的效果，毕竟 canny 是公认的效果很好的算法。然而无论怎么调参，还是效果很不好。从效果上看，只能说比 matlab 自带算法好一些，对于城堡的上半部还可以，下半部也变成了离散的点了。</p>

2.2.1.5 noise2.png

输入		<p>原图</p> <p>背景感觉有些模糊并且伴有少量白色噪声点，主体部分还算清晰，至少比城堡清晰。</p>
----	---	--

输出 1		<p>matlab 自带算法</p> <p>周围的噪声还是挺多的，主体部分的边缘检测得还是挺好的。</p>
输出 2		<p>LoG 算法</p> <p>LOG 算法的效果都是不错的，边缘的细节检测得也挺好，主要的毛病是边缘线较粗</p>
输出 3		<p>Canny 算法</p> <p>个人觉得 canny 的效果比 matlab 自带算法好，边缘更加细，细节展现得更加到位，任务身上的花纹也更加明显、有条理，而不是交叉混乱。</p>

2.2.1.6 car.png

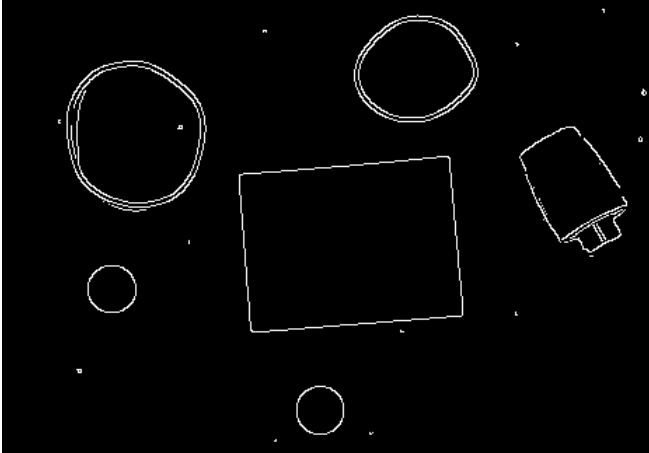
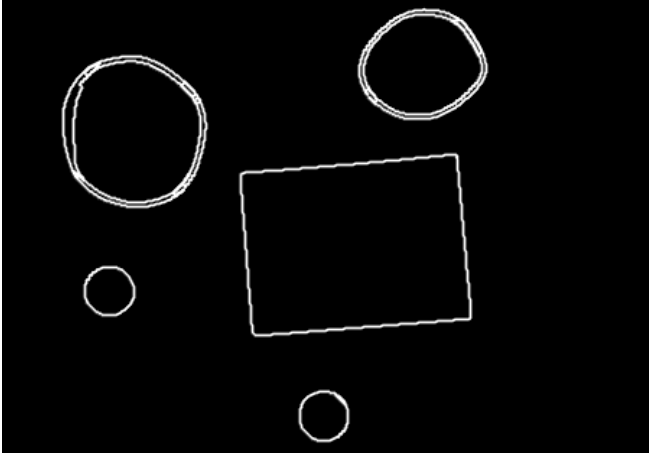
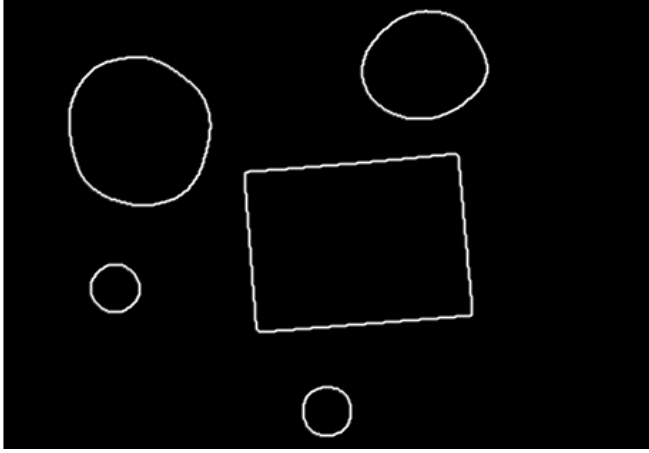
输入		原图 噪声几乎没有，边缘很清晰。
输出 1		matlab 自带算法 对于汽车底盘的检测就没有了，看上去很奇怪，效果不是很好。
输出 2		Canny 算法 canny 的效果比 matlab 自带算法好，整个汽车看得更加清晰，轮廓也是相当完整。

2.2.1.7 cat.png

输入		<p>原图</p> <p>背景感觉有些模糊并且伴有少量噪声点，主体部分的边缘也没有那么清晰，难度还是比较大的。</p>
输出 1		<p>matlab 自带算法</p> <p>对于猫轮廓的检测感觉不是很好，尤其是在后背的地方断断续续，身体的细节也不明显。</p>
输出 2		<p>Canny 算法</p> <p>canny 的效果比 matlab 自带算法好，猫的外轮廓相对清晰，四肢的细节也处理的比 matlab 的效果好。</p>

2.2.2 边缘链接

2.2.2.1 rubberband_cap.png

输入		<p>Canny 边缘检测图</p> <p>输入是 canny 边缘检测后生成的二值图，噪声很少，边缘较为清晰。</p>
输出 1		<p>第一种边缘链接算法</p> <p>这种边缘链接算法可以处理间断的边缘，将不连续的边缘链接起来，但是无法解决多边缘的问题，可以看到图像上面两个大圆还是链接出了内外两圈。</p>
输出 2		<p>第二种边缘链接算法</p> <p>这种边缘链接算法可以处理多边缘的问题，如刚刚两个大圆的内外两圈就被处理成了一圈，效果可以说是非常棒。但是此算法对于边缘检测的要求很高。</p>

2.3 实验对比结论

可以明显得看出，对于绝大部分的图片，Canny 算法的效果相对来说是最好的，这也是由于其算法考虑了各方面的情况，进行了多方面的处理。值得一提的是 LoG 算法的效果甚至有时候是超过 Canny 的，唯一的缺点是边缘线较粗。