
# 中国海洋大学
OCEAN UNIVERSITY OF CHINA

2017 年 秋 季学期

课程名称： 信号与系统

学生姓名： 郑自强 蔡琦琦 陈婉冰 曹羽成 关馨
黄雪桐 刘宁宁 田云芳 王新领 张祥悦

任课教师： 李光亮

# 目录

# 一．Canny 边缘检测

**方法简介：**

Canny 边缘检测算子是一种多级检测算法。1986 年由 John F. Canny 提出，同时提出了边缘检测的三大准则：

1.低错误率的边缘检测：检测算法应该精确地找到图像中的尽可能多的边缘，尽可能的减少漏检和误检。

2.最优定位：检测的边缘点应该精确地定位于边缘的中心。

3.图像中的任意边缘应该只被标记一次，同时图像噪声不应产生伪边缘。

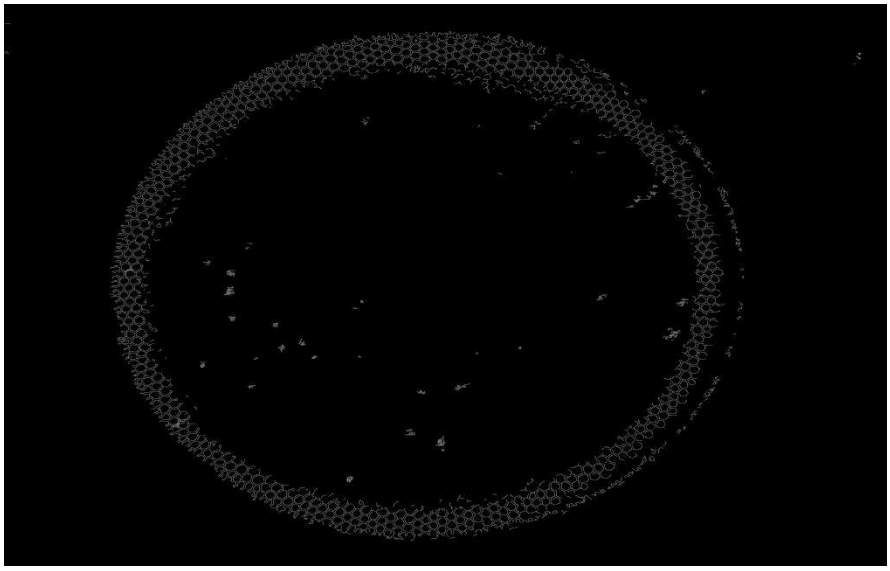
Canny 算法出现以后一直是作为一种标准的边缘检测算法，此后也出现了各种基于 Canny 算法的改进算法。

**代码实现：**

```
#coding=utf-8
# import cv2
# import numpy as np
#
# img = cv2.imread("./demo1.jpg", 0)
#
# img = cv2.GaussianBlur(img, (3, 3), 0)
# canny = cv2.Canny(img, 50, 150)
#
# # cv2.imshow('Canny', canny)
# cv2.imwrite('./demo1_candy.jpg',canny)
# cv2.waitKey(0)
# cv2.destroyAllWindows()

import cv2
import numpy as np

def CannyThreshold(lowThreshold):
    detected_edges = cv2.GaussianBlur(gray,(3,3),0)
    detected_edges                                                    =
cv2.Canny(detected_edges,lowThreshold,lowThreshold*ratio,apertureSize
= kernel_size)
    dst = cv2.bitwise_and(img,img,mask = detected_edges)  # just add
some colours to edges from original image.
    cv2.imshow('canny demo',dst)

lowThreshold = 0
max_lowThreshold = 100
ratio = 3
kernel_size = 3
img = cv2.imread('./demo1.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
```

```
cv2.namedWindow('canny demo')
cv2.createTrackbar('Min threshold','canny demo',lowThreshold,
max_lowThreshold, CannyThreshold)
CannyThreshold(0)  # initialization
if cv2.waitKey(0) == 27:
    cv2.destroyAllWindows()
```

**实验结果：**



# 二.Sobel 边缘检测

**方法简介：**

Sobel 算子是一个主要用作边缘检测的离散微分算子 (discrete differentiation operator)。 它结合了高斯平滑和微分求导，用来计算图像灰度函数的近似梯度。在图像的任何一点使用此算子，将会产生对应的梯度矢量或是其法矢量。

OpenCV 中的 Sobel 函数：

会使用到 9 个参数，分别是：

第一个参数，InputArray 类型的 src，为输入图像，填 Mat 类型即可。

第二个参数，OutputArray 类型的 dst，即目标图像，函数的输出参数，需要和源图片有一样的尺寸和类型。

第三个参数，int 类型的 ddepth，输出图像的深度，支持如下 src.depth()和 ddepth 的组合：

若 src.depth() = CV_8U, 取 ddepth =-1/CV_16S/CV_32F/CV_64F；

若 src.depth() = CV_16U/CV_16S, 取 ddepth =-1/CV_32F/CV_64F；

若 src.depth() = CV_32F, 取 ddepth =-1/CV_32F/CV_64F；

若 src.depth() = CV_64F, 取 ddepth = -1/CV_64F；

第四个参数，int 类型 dx，x 方向上的差分阶数。

第五个参数，int 类型 dy，y 方向上的差分阶数。

第六个参数，int 类型 ksize，有默认值 3，表示 Sobel 核的大小;必须取 1，3，5 或 7。

第七个参数，double 类型的 scale，计算导数值时可选的缩放因子，默认值是 1，表示默认情况下是没有应用缩放的。我们可以在文档中查阅 getDerivKernels 的相关介绍，来得到这个参数的更多信息。

第八个参数，double 类型的 delta，表示在结果存入目标图（第二个参数 dst）之前可选的 delta 值，有默认值 0。

第九个参数， int 类型的 borderType，我们的老朋友了（万年是最后一个参数），边界模式，默认值为 BORDER_DEFAULT。

**代码实现：**#include<iostream>

#include<opencv2/imgproc/imgproc.hpp>

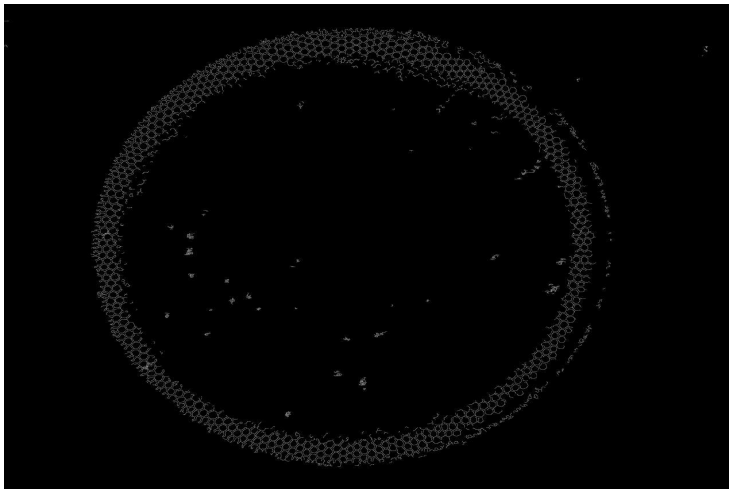
#include<opencv2/highgui/highgui.hpp>

#include<opencv2/core/core.hpp>

```
#include<math.h>
#include<time.h>
using namespace cv;
using namespace std;
using namespace cv;
int main( int argc, char** argv )
{ Mat src, src_gray;
  Mat grad;
  string window_name = "Sobel Demo - Simple Edge Detector";
  int scale = 1;
  int delta = 0;
  int ddepth = CV_16S;
  int c;
  src = imread( argv[1] );
  if( !src.data )
  { return -1; }
  GaussianBlur( src, src, Size(3,3), 0, 0, BORDER_DEFAULT );
  cvtColor( src, src_gray, CV_RGB2GRAY );
  namedWindow( window_name, CV_WINDOW_AUTOSIZE );
  Mat grad_x, grad_y;
  Mat abs_grad_x, abs_grad_y;
  //Scharr( src_gray, grad_x, ddepth, 1, 0, scale, delta, BORDER_DEFAULT );
  Sobel( src_gray, grad_x, ddepth, 1, 0, 3, scale, delta, BORDER_DEFAULT );
  convertScaleAbs( grad_x, abs_grad_x );
  Scharr(src_gray, grad_y, ddepth, 0, 1, scale, delta, BORDER_DEFAULT );
  Sobel( src_gray, grad_y, ddepth, 0, 1, 3, scale, delta, BORDER_DEFAULT );
  convertScaleAbs( grad_y, abs_grad_y );
  addWeighted( abs_grad_x, 0.5, abs_grad_y, 0.5, 0, grad );
```

```
imshow( window_name, grad );
waitKey(0);
return 0;
}
```

**实验结果：**



# 三、Gabor 变换

**方法简介：**Gabor 变换是短时傅里叶变换的一个特例。短时傅里叶变换能够完成局部分析的关键是窗口函数，窗口的尺度是局部性程度的表征。当窗函数取为高斯窗时一般称为 Gabor 变换。选高斯窗的原因在于高斯函数的 Fourier 变换仍是高斯函数，这使得 Fourier 逆变换也用窗函数局部化了，同时体现了频率域的局部化，同时根据 Heisenberg 测不准原理，高斯函数窗口面积已达到测不准原理下界，是时域窗口面积达到最小的函数，即 Gabor 变换是最优的短时傅里叶变换。

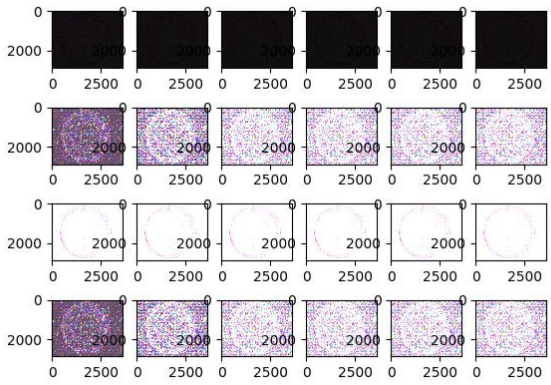
Gabor 变换可以达到时频局部化的目的：它能够在整体上提供信号的全部信息而又能提供在任一局部时间内信号变化剧烈程度的信息。简言之，可以同时提供时域和频域局部化的信息。

**代码实现：**

```
# coding:utf-8
import cv2
import numpy as np
import pylab as pl
from PIL import Image
def build_filters():
filters = []
ksize = [7,9,11,13,15,17]
lamda = np.pi/2.0
for theta in np.arange(0,np.pi,np.pi/4):
for k in range(6):
kern = cv2.getGaborKernel((ksize[k],ksize[k]),1.0,theta,lamda,0.5,0,ktype=cv2.CV_32F)
kern /= 1.5*kern.sum()
filters.append(kern)
return filters
def process(img,filters):
accum = np.zeros_like(img)
for kern in filters:
fimg = cv2.filter2D(img,cv2.CV_8UC3,kern)
np.maximum(accum,fimg,accum)
```

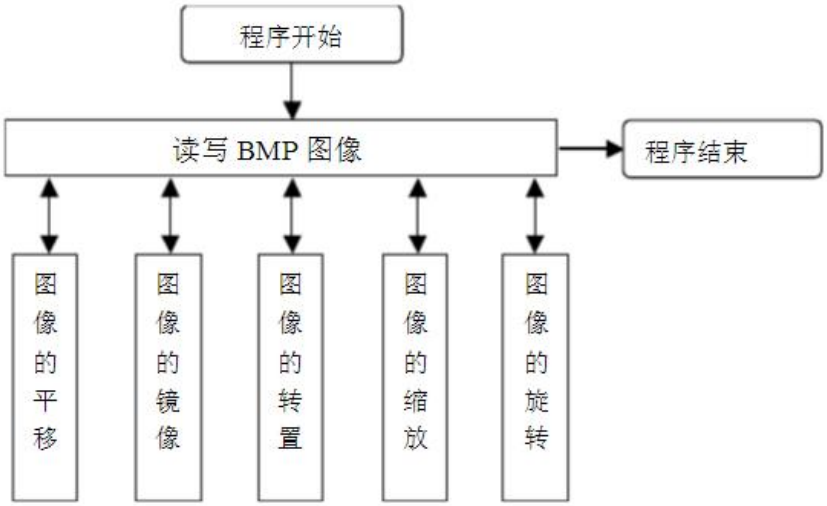
```
 return accum;
def getGabor(img,filters):
image = Image.open(img)
img_ndarray = np.asarray(image)
res = []
for i in range(len(filters)):
res1 = process(img_ndarray,filters[i])
res.append(np.asarray(res1))
pl.figure(2)
for temp in range(len(res)):
pl.subplot(4,6,temp+1)
pl.imshow(res[temp],cmap='gray')
pl.show()
return res
if __name__ == '__main__':
filters = build_filters()
getGabor('./demo1.jpg',filters)
```

**实验结果：**



# 四．几何变换

**方法简介：**几何变换是最常见的图像处理手段，通过对变形的图像进行几何校正，可以得出准确的图像。常用的几何变换功能包括图像的平移、图像的镜像变换、图像的转置、图像的缩放、图像的旋转等等。

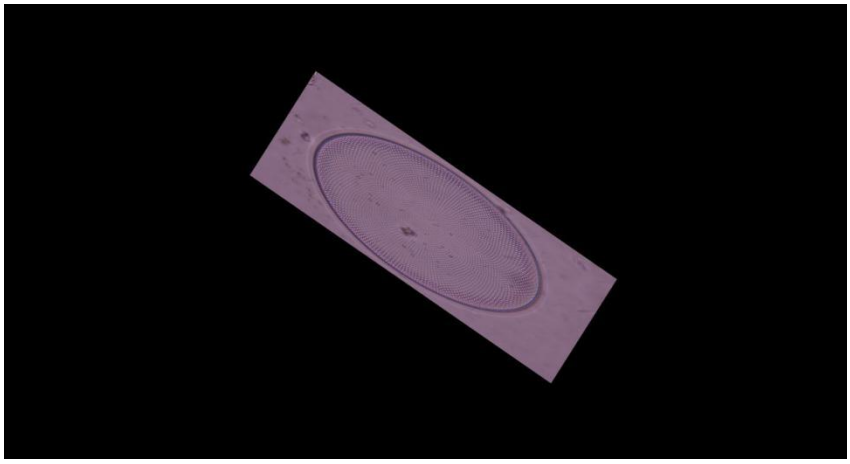



**代码实现：**import cv2

```
import numpy as np

img = cv2.imread('./demo1.jpg')

rows,cols = img.shape[:2]

Source points

srcTri = np.array([(0,0),(cols-1,0),(0,rows-1)], np.float32)

 Corresponding Destination Points. Remember, both sets are of float32 type

dstTri = np.array([(cols*0.0,rows*0.33),(cols*0.85,rows*0.25),(cols*0.15,rows*0.7)],np.float32)

Affine Transformation

warp_mat = cv2.getAffineTransform(srcTri,dstTri)

Generating affine transform matrix of size 2x3

dst = cv2.warpAffine(img,warp_mat,(cols,rows))

  # Now transform the image, notice dst_size=(cols,rows), not (rows,cols)

Image Rotation
```

```
center
(cols/2,rows/2)
 Center point about which image is transformed
angle  -50.0
 Angle, remember negative angle denotes clockwise rotation
scale 0.6
Isotropic scale factor
rot_mat = cv2.getRotationMatrix2D(center,angle,scale)
 Rotation matrix generated
dst_rot = cv2.warpAffine(dst,rot_mat,(cols,rows))
 Now transform the image wrt rotation matrix
cv2.imshow('dst_rt',dst_rot)
cv2.imwrite("geometric_transform.jpg",dst_rot)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

**实验结果：**



# 五．分水岭算法

**方法简介：**分水岭算法是以数学形态学作为基础的一种区域分割方法。其基本思想是将梯度图像看成是假想的地形表面，每个像素的梯度值表示该点的海拔高度。原图中的平坦区域梯度较小，构成盆地，边界处梯度较大构成分割盆地的山脊。分水岭算法模拟水的渗入过程，假设水从最低洼的地方渗入，随着水位上升，较小的山脊被淹没，而在较高的山脊上筑起水坝，防止两区域合并。当水位达到最高山脊时，算法结束，每一个孤立的积水盆地构成一个分割区域。由于受到图像噪声和目标区域内部的细节信息等因素影响，使用分水岭算法通常会产生过分割现象，分水岭算法一般是作为一种预分割方法，与其它分割方法结合使用，以提高算法的效率或精度。

在该算法中，分水岭计算分两个步骤，一个是排序过程，一个是淹没过程。首先对每个像素的灰度级进行从低到高排序，然后在从低到高实现淹没过程中，对每一个局部极小值在 h 阶高度的影响域采用先进先出(FIFO)结构进行判断及标注。

分水岭变换得到的是输入图像的集水盆图像，集水盆之间的边界点，即为分水岭。显然，分水岭表示的是输入图像极大值点。因此，为得到图像的边缘信息，通常把梯度图像作为输入图像，即

g(x,y)=grad(f(x,y))={[f(x,y)-f(x-1,y)]2[f(x,y)-f(x,y-1)]2}0.5

式中，f(x,y)表示原始图像，grad{.}表示梯度运算。

为消除分水岭算法产生的过度分割，通常可以采用两种处理方法，一是利用先验知识去除无关边缘信息。二是修改梯度函数使得集水盆只响应想要探测的目标。为降低分水岭算法产生的过度分割，通常要对梯度函数进行修改，一个简单的方法是对梯度图像进行阈值处理，以消除灰度的微小变化产生的过度分割。即

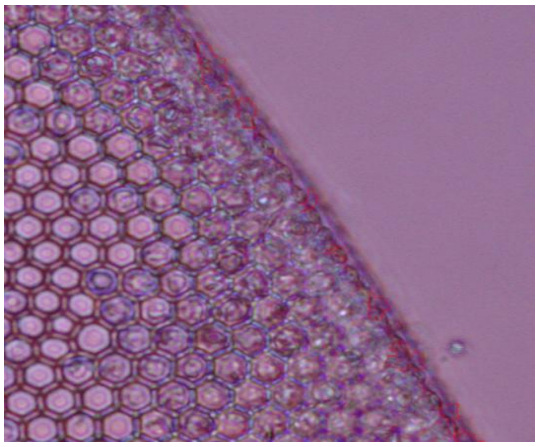
g(x,y)=max(grad(f(x,y)),gθ)式中，gθ表示阈值。

**代码实现：**#coding=utf-8

```
from PIL import Image

from pylab import *

import cv2

import numpy as np

img = cv2.imread("/home/scs4450/demo/大作业/115 拖网 1 蜂窝三角藻壳面观 2－40.jpg")

gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

ret,thresh=cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)

# thresh = cv2.imread(files)
```

```
print(img.shape)

thresh = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

kernel = np.ones((3, 3), np.uint8)

opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel, iterations=2)

sure_bg = cv2.dilate(opening, kernel, iterations=3)

Finding sure foreground area

dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)

ret, sure_fg = cv2.threshold(dist_transform, 0.7*dist_transform.max(), 255, 0)

 Finding unknown region

sure_fg = np.uint8(sure_fg)

unknown = cv2.subtract(sure_bg, sure_fg)

ret, markers = cv2.connectedComponents(sure_fg)

Add one to all labels so that sure background is not 0, but 1

markers = markers + 1

Now, mark the region of unknown with zero

markers[unknown == 255] = 0

markers = cv2.watershed(img, markers)

img[markers == -1] = [255, 0, 0]

img_save = Image.fromarray(img, 'RGB')

img_save.save('demo_0.7.png')
```

**实验结果：**



# 六. 卷积操作图像

**方法简介：**在图像处理中，卷积操作指的是使用一个卷积核对图像中的每个像素进行一系列操作。 卷积核（算子）是用来做图像处理时的矩阵,图像处理时也称为掩膜，是与原图像做运算的参数。卷积核通常是一个四方形的网格结构（例如3*3 的矩阵或像素区域），该区域上每个方格都有一个权重值。使用卷积进行计算时，需要将卷积核的中心放置在要计算的像素上，一次计算核中每个元素和其覆盖的图像像素值的乘积并求和，得到的结构就是该位置的新像素值。
卷积模板

$$\begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

卷积公式表达

$$g = f * h.$$

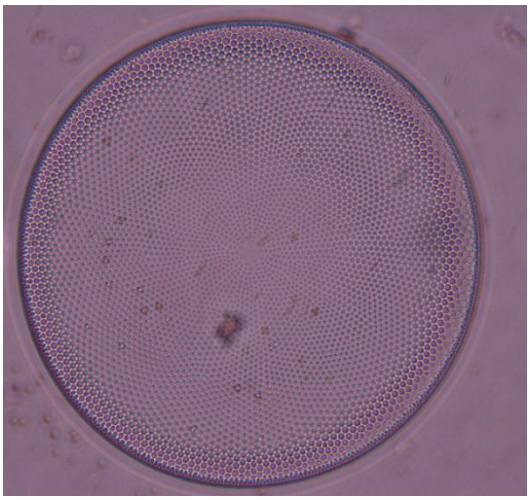
$$g(i,j) = \sum_{k,l} f(i-k, j-l)h(k,l) = \sum_{k,l} f(k,l)h(i-k, j-l)$$

我们读取一幅图像，将该图像与指定的卷积核进行卷积操作。得到不同的卷积结果，通过对比结果来获取不同的卷积核对图像卷积的影响。
**代码实现：**

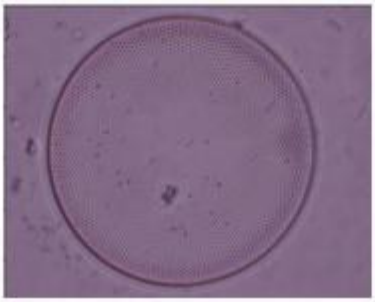
```
#coding=utf-8
import matplotlib.pyplot as plt
import pylab
import cv2
import numpy as np

img = cv2.imread("demo1.jpg")                          #在这里读取图片

# plt.imshow(img)                                      #显示读取的图片
# pylab.show()

fil = np.array([[ 1.,1, 1],                            #这个是设置的滤波，也就是卷积核
                [ 1, -2, 1],
                [  -1, -1, -1]])

res = cv2.filter2D(img,-1,fil)                         #使用opencv的卷积函数

plt.imshow(res)                                        #显示卷积后的图片
plt.imsave("res.jpg",res)
pylab.show()
```

**实验结果：**

Input

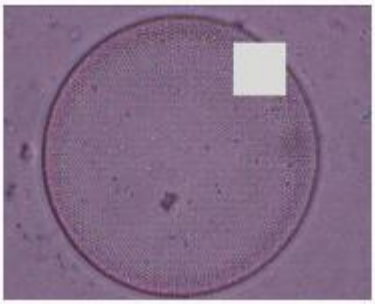


$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \times \frac{1}{9}$$



低通滤波器

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$



高通滤波器

$$\begin{bmatrix} -1 & 0 & -1 \\ 0 & 4 & 0 \\ -1 & 0 & -1 \end{bmatrix}$$

边缘检测器


**实验结果分析：**

可以看到使用不同的卷积核的实验结果是不同的，且我们可以针对不同的需要效果来调整我们的卷积核来实现一个滤波的效果，比如图 3，我们可以滤去颜色信息而保留边缘信息。

# 七．拉普拉斯变换

**方法简介：**对图像进行处理，使图像中的各灰度值得到保留，使灰度突变处的对比度得到增强，在保留图像背景的前提下，突现出图像中小的细节信息。

图像锐化处理的作用是使灰度反差增强，从而使模糊图像变得更加清晰。图像模糊的实质就是图像受到平均运算或积分运算，因此可以对图像进行逆运算，如微分运算能够突出图像细节，使图像变得更为清晰。由于拉普拉斯是一种微分算子，它的应用可增强图像中灰度突变的区域，减弱灰度的缓慢变化区域。因此，锐化处理可选择拉普拉斯算子对原图像进行处理，产生描述灰度突变的图像，再将拉普拉斯图像与原始图像叠加而产生锐化图像。

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

拉普拉斯算子的定义：

Laplacian 算子一般不以其原始形式用于边缘检测，因为其作为一个二阶导数，Laplacian 算子对噪声具有无法接受的敏感性；同时其幅值产生算边缘，这是复杂的分割不希望有的结果；最后 Laplacian 算子不能检测边缘的方向；所以 Laplacian 在分割中所起的作用包括：（1）利用它的零交叉性质进行边缘定位；（2）确定一个像素是在一条边缘暗的一面还是亮的一面；一般使用的是高斯型拉普拉斯算子（Laplacian of a Gaussian,LoG），由于二阶导数是线性运算，利用 LoG 卷积一幅图像与首先使用高斯型平滑函数卷积改图像，然后计算所得结果的拉普拉斯是一样的。所以在 LoG 公式中使用高斯函数的目的就是对图像进行平滑处理，使用 Laplacian 算子的目的是提供一幅用零交叉确定边缘位置的图像；图像的平滑处理减少了噪声的影响并且它的主要作用还是抵消由 Laplacian 算子的二阶导数引起的逐渐增加的噪声影响。

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | | | 1 | 1 | 1 |
| 1 | -4 | 1 | | | 1 | -8 | 1 |
| 0 | 1 | 0 | | | 1 | 1 | 1 |

（a）拉普拉斯运算模板　　（b）拉普拉斯运算扩展模板

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | -1 | 0 | | | -1 | 1 | -1 |
| -1 | 4 | -1 | | | 1 | 8 | -1 |
| 0 | -1 | 0 | | | -1 | 1 | -1 |

（c）拉普拉斯其他两种模板



另外，拉普拉斯算子还可以表示成模板的形式，如图 5-9 所示。

图 5-9（a）表示离散拉普拉斯算子的模板，图 5-9（b）表示其扩展模板，图 5-9（c）则分别表示其他两种拉普拉斯的实现模板。从模板形式容易看出，如果在图像中一个较暗的区域中出现了一个亮点，那么用拉普拉斯运算就会使这个亮点变得更亮。因为图像中的边缘就是那些灰度发生跳变的区域，所以拉普拉斯锐化模板在边缘检测中很有用。一般增强技术对于陡峭的边缘和缓慢变化的边缘很难确定其边缘线的位置。但此算子却可用二次微分正峰和负峰之间的过零点来确定，对孤立点或端点更为敏感，因此特别适用于以突出图像中的孤立点、孤立线或线端点为目的的场合。同梯度算子一样，拉普拉斯算子也会增强图像中的噪声，有时用拉普拉斯算子进行边缘检测时，可将图像先进行平滑处理。

将原始图像和拉普拉斯变换图像相叠加，既能保护拉普拉斯锐化处理的效

$$g(x,y)=\begin{cases} f(x,y)-\nabla^2 f(x,y) \\ f(x,y)+\nabla^2 f(x,y) \end{cases}$$

果，同时又能复原背景信息。

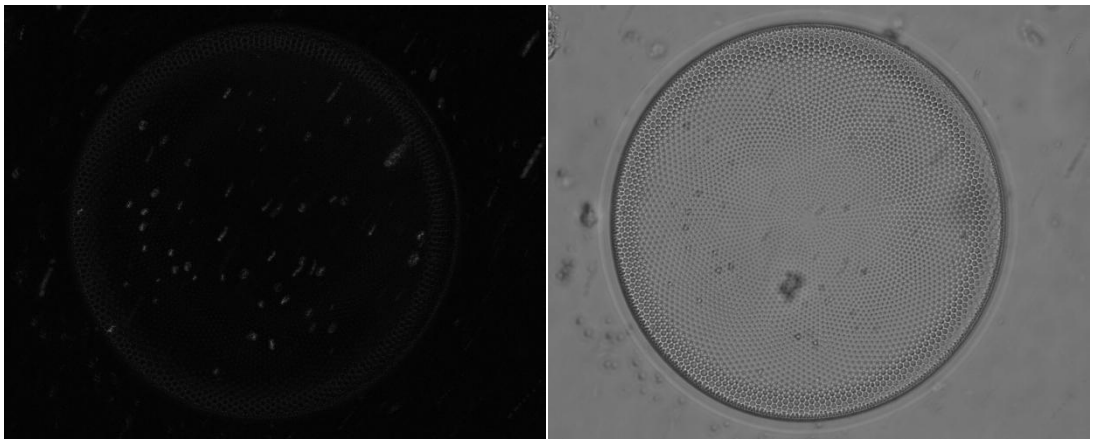
图像相对平坦的区域拉普拉斯变换后的值约等于 0。像素（原）-0=像素（原），即该区域图像几乎没有变化。

在图像中强度值变换剧烈的地方，通常是物体的边缘处，拉普拉斯变换后的该区域的强度值（绝对值）较大。那么 像素（原）-像素（变换后），肯定会发生变化。例如：150（原）-50（变换后的 ）=100，该点处的像素变黑。

**代码实现：**#include<iostream>

```
#include<opencv2/imgproc/imgproc.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/core/core.hpp>
#include<math.h>
#include<time.h>
using namespace cv;
using namespace std;
include<opencv2/highgui/highgui.hpp>
nclude</usr/local/include/opencv/cv.h>
usr/local/include/opencv/cv.h
using namespace cv;
IplImage *Laplacian_cx(IplImage *src){
    IplImage *dst = cvCreateImage(cvGetSize(src),IPL_DEPTH_8U,0);
    cvLaplace(src,dst);
    return dst;
}
int main(){
IplImage * srcImage = cvLoadImage("./pic1.jpg",0);
cvNamedWindow("源灰度图像",CV_WINDOW_AUTOSIZE);
显示源图像
cvShowImage("源灰度图像",srcImage);
cvNamedWindow("Laplacian 变换后的灰度图像",CV_WINDOW_AUTOSIZE);
cvShowImage("Laplacian 变换后的灰度图像",Laplacian_cx(srcImage));
waitKey(60000);
}
```

## 实验结果：



# 八．Contour 检测

**方法简介：**代码中主要用到 threshold,showimage 等函数方法
threshold 是通过遍历灰度图中点，将图像信息二值化，处理过后的图片只有二种色值。

其函数原型如下：

```
double threshold(InputArray src, OutputArray dst, double thresh, double maxval, int type)
```

参数信息：
第一个参数，InputArray 类型的 src，输入数组，填单通道，8 或 32 位浮点类型的 Mat 即可。
第二个参数，OutputArray 类型的 dst，函数调用后的运算结果存在这里，即这个参数用于存放输出结果，且和第一个参数中的 Mat 变量有一样的尺寸和类型。
第三个参数，double 类型的 thresh，阈值的具体值。
第四个参数，double 类型的 maxval，当第五个参数阈值类型 type。
取 THRESH_BINARY 或 THRESH_BINARY_INV 阈值类型时的最大值。
第五个参数，int 类型的 type，阈值类型。

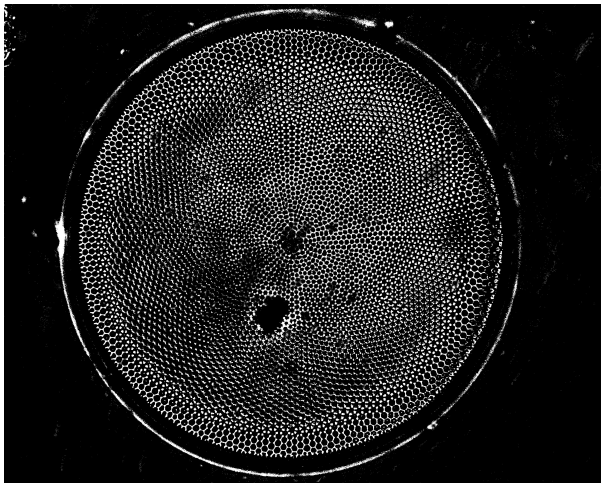
showimage 函数

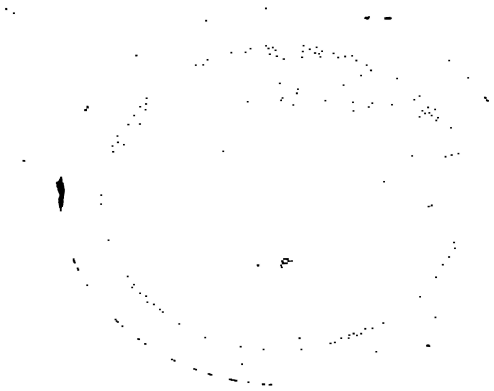
```
1. void CmymfcDlg::ShowImage( IplImage* img, UINT ID ) // ID 是 Picture Control 控件的 ID 号
2. {
3.      // 获得显示控件的 DC
4.      CDC* pDC = GetDlgItem( ID ) ->GetDC();
5.     // 获取 HDC(设备句柄) 来进行绘图操作
6.     HDC hDC = pDC ->GetSafeHdc();
7.      CRect rect; GetDlgItem(ID) ->GetClientRect( &rect );
8.     // 求出图片控件的宽和高
9.     int rw = rect.right - rect.left; int rh = rect.bottom - rect.top;
10.         // 读取图片的宽和高
11.         int iw = img->width; int ih = img->height;
12.        // 使图片的显示位置正好在控件的正中
13.        int tx = (int)(rw - iw)/2;
14.        int ty = (int)(rh - ih)/2; SetRect( rect, tx, ty, tx+iw, ty+ih );
15.        // 复制图片 CvvImage cimg; cimg.CopyOf( img );
16.        // 将图片绘制到显示控件的指定区域内
17.        cimg.DrawToHDC( hDC, &rect );
18.        ReleaseDC( pDC );
19.    }
```

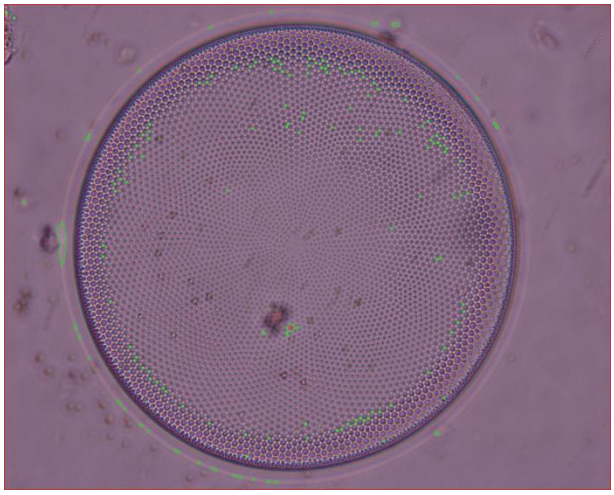
**代码实现：**

```
import  cv2.cv as cv
import cv
orig = cv.LoadImage('./demo1.jpg', cv.CV_LOAD_IMAGE_COLOR)
im = cv.CreateImage(cv.GetSize(orig), 8, 1)
cv.CvtColor(orig, im, cv.CV_BGR2GRAY)
Keep the original in colour to draw contours in the end
cv.Threshold(im, im, 128, 255, cv.CV_THRESH_BINARY)
cvShowImage.("Threshold 1", im)
cv.SaveImage("threshold1.jpg",im)
element = cv.CreateStructuringElementEx(5*2+1, 5*2+1, 5, 5,
cv.CV_SHAPE_RECT)
cv.MorphologyEx(im, im, None, element, cv.CV_MOP_OPEN) #Open and close
to make appear contours
cv.MorphologyEx(im, im, None, element, cv.CV_MOP_CLOSE)
cv.Threshold(im, im, 128, 255, cv.CV_THRESH_BINARY_INV)
cv.ShowImage("After MorphologyEx", im)
cv.SaveImage("after.jpg",im)
vals = cv.CloneImage(im) #Make a clone because FindContours can modify
the image
contours=cv.FindContours(vals, cv.CreateMemStorage(0), cv.CV_RETR_LIST,
cv.CV_CHAIN_APPROX_SIMPLE, (0,0))
_red = (0, 0, 255); #Red for external contours
_green = (0, 255, 0);# Gren internal contours
levels=2 #1 contours drawn, 2 internal contours as well, 3 ...
co=cv.DrawContours (orig, contours, _red, _green, levels, 2,
cv.CV_FILLED) #Draw contours on the colour image
cv.SaveImage("save.jpg",orig)
cv.SaveImage("co.jpg",co)
cv.ShowImage("Image", orig)
cv.WaitKey(0)
```

**实验结果：**threshold







# 九．Histogram 函数

**方法简介：**

1.Histogram()是 MATLAB 中绘制数据直方图的函数，使用自动分箱算法，返回具有统一宽度的分箱，选择该分箱以覆盖 X 中元素的范围，并显示分布的基本形状。 直方图显示为矩形，使每个矩形的高度指示箱中元素的数量。

2.Histogram()函数有以下用法：

histogram(X)；histogram(X,edges)；histogram('BinEdges',edges,'BinCounts',counts)；
histogram(C)；histogram(C,Categories)；
histogram('Categories',Categories,'BinCounts',counts)；
histogram(___,Name,Value).....

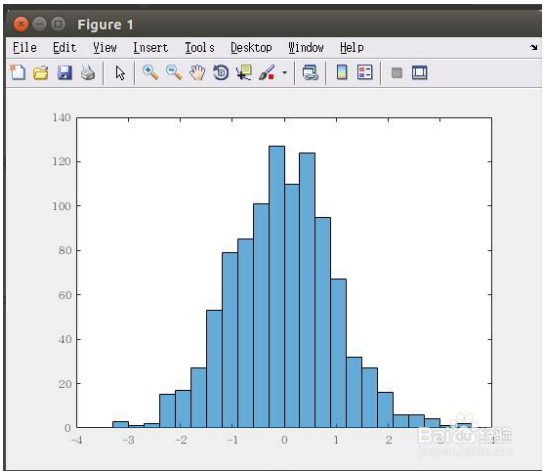
【注　C：分配的数据
Categories：在直方图中包含的类别
edges- Bin：边缘】

3.举例说明

（1）生成一列数据：

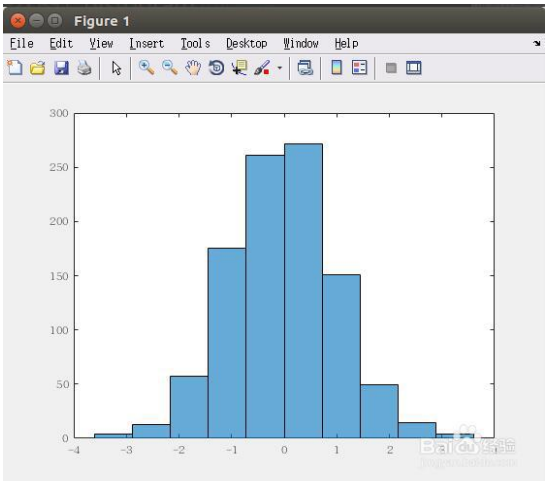
```
a = rand(1000,1);
h = histogram(a);
```

对 h 进行统计，MATLAB 自动对 h 进行分列



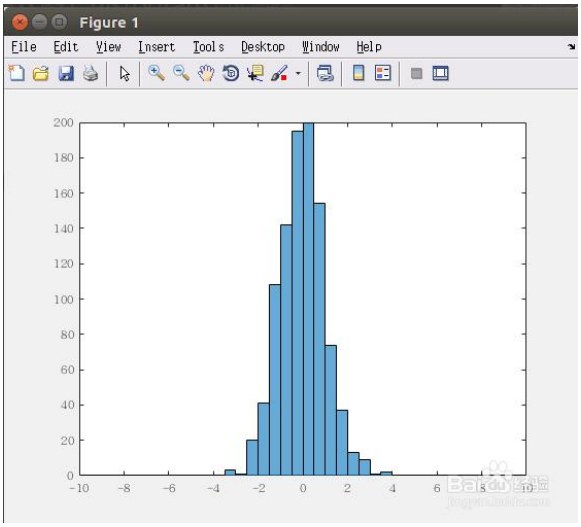

（2）指定柱状的数量：

```
hh = histogram(a,10);
```



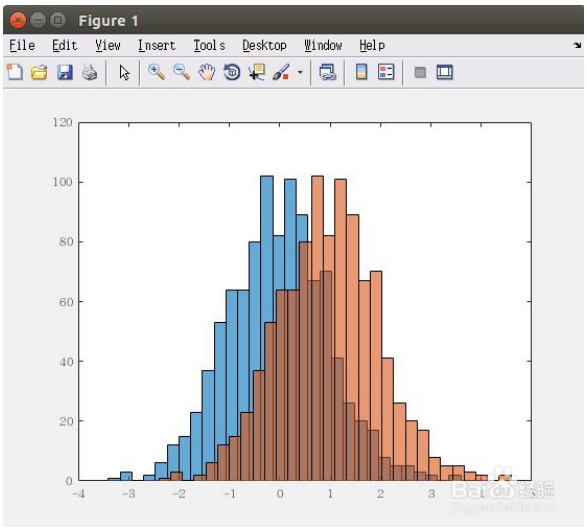

（3）控制横坐标的范围：

```
hhh = histogram(a,[-10:0.5:10]);
```




（4）将几列数据画在一张图上：

```
histogram(a,30);
aa = 1+a;
hold on;
histogram(aa,30);
```



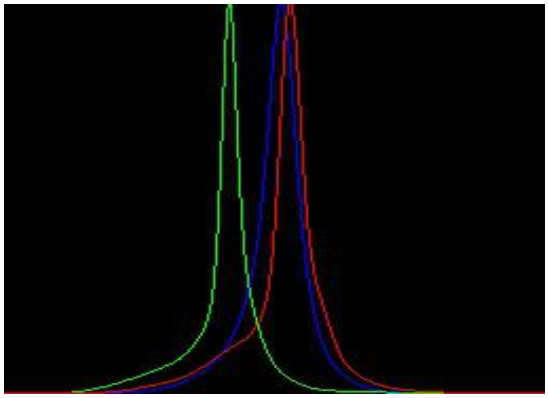

（5）可加入其他要素

**代码实现：**

```
import cv2
import numpy as np
img = cv2.imread('./demo1.jpg')
h = np.zeros((300,256,3))
image to draw histogram
bins = np.arange(256).reshape(256,1)
Number of bins, since 256 colors, we need 256 bins
color = [ (255,0,0),(0,255,0),(0,0,255) ]
for ch,col in enumerate(color):
hist_item = cv2.calcHist([img],[ch],None,[256],[0,256])
Calculates the histogram
cv2.normalize(hist_item,hist_item,0,255,cv2.NORM_MINMAX)
Normalize the value to fall below 255, to fit in image 'h'
 hist=np.int32(np.around(hist_item))
pts = np.column_stack((bins,hist))
stack bins and hist, ie [[0,h0],[1,h1]....,[255,h255]]
cv2.polylines(h,[pts],False,col)
h=np.flipud(h)
You will need to flip the image vertically
cv2.imshow('colorhist',h)
cv2.imwrite("histogram.jpg",h)
cv2.waitKey(0)
 cv2.destroyAllWindows()
import cv2
import numpy as np
  img = cv2.imread('./demo1.jpg')
```

```
    anchor = (-1, -1)
    delta = 0
    ddepth = -1
    ind = 0
    while (True):
        cv2.imshow('image', img)
        k = cv2.waitKey(500)

        if k == 27:
            break
        kernel_size = 3 + 2 * (ind % 5)  # trying for kernel sizes [3,5,7,9,11]
        kernel = np.ones((kernel_size, kernel_size), np.float32) / (kernel_size * kernel_size)
        cv2.filter2D(img, ddepth, kernel, img, anchor, delta, cv2.BORDER_DEFAULT)
        ind = ind + 1
    cv2.destroyAllWindows()
```

**实验结果：**



**总结体会：**

通过此次实验，我们了解了图像处理的基本知识，初步接触了一些方法，并且更好地体会到各学科之间的相通之处。而且在撰写报告的过程中，遇到问题各成员能够相互帮助，共同进步，每个人都收获不少。