

## 作业完整编程实验部分

1、请阅读所给的实例代码回答下列问题

(1) 该 MFC 的工程是哪种结构？单文档、多文档、还是对话框？

答：单文档，因为没有 CxxxDlg 类，也没有 CChildFrm 类

(2) 文件打开对话框中，哪个表示图像所在的文件名，打开文件后，图像的数据在哪里存放？

答：1、存放在 Onopen 方法下的 Cfile 类型对象 file 中

2、存放在叫做 mybmp 的 CimageprocessView 实例下实例化的一个 CDib 类对象中

(3) 在修改行和列颜色的代码中，x 和 y，两者哪个表示行变量？哪个表示列变量？

答：x 表示列，y 表示行。

(4) 自己研究一下，图像显示在 View 类的哪个成员函数中完成的？

绘制是在 Ondraw(CDC \*pDC) 下 `mybmp.Draw(pDC, CPoint(0,0), sizeDibDisplay);` zhong 完成的，重新绘制(例如绘制打开的图片或者上红绿色)则是用 Invalidate(TRUE) 这个方法激发了 Ondraw 的调用

(5) 程序中 Invalidate(TRUE); 的作用是什么？去掉可不可以？

使整个窗口客户区无效，并进行更新显示的函数。参数 True 代表是否要擦除背景。

2、自己仿照实例建立一个工程，先加入 CDib 类，然后试图修改一下图像中任意像素的颜色。

·建立对应工程，并且识别中心爱心函数对应的像素

```
for (int x = -ImageSize.cx; x < ImageSize.cx; x++)
{
    RGBQUAD p1;
    RGBQUAD p2;
    double transX= ((double)x)/nx/* - (double)cx / 2*/;
    if (transX <= 1 && transX >= -1)
    {

        //1st
        double transY1 = -(sqrt(1 - pow(transX, 2)) + pow(transX*transX, 1.0/3))*ny+(double)cy/2;
        p1 = mybmp.GetPixel(x, transY1+0.5);
        p1.rgbRed = 255;
        p1.rgbGreen = 0;
        p1.rgbBlue = 0;
        mybmp.WritePixel(x+cx/2, transY1+0.5, p1);

        //2nd
        double transY2 = -(-sqrt(1 - pow(transX, 2)) + pow(transX*transX, 1.0 / 3))*ny+(double)cy / 2;
        p2 = mybmp.GetPixel(x, transY2+0.5);
        p2.rgbRed = 255;
        p2.rgbGreen = 0;
        p2.rgbBlue = 0;
        mybmp.WritePixel(x+cx/2, transY2+0.5, p2);
    }
}
```

·将对应像素变为红色，保存得到如下图像



## W02

- 1、建立一个单文档 MFC 工程，将 CDib 类加入，参考我给的代码，能否打开两个图像，并显示在 View 中。要求两个图像并列显示

Ondraw 中改变

```
void CW02View::OnDraw(CDC* pDC)
{
    CW02Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;
    CRect rt;
    GetClientRect(&rt);
    mybmp_1.Draw(pDC, (0, 0), sizeDibDisplay_1);
    mybmp_2.Draw(pDC, (0, /*sizeDibDisplay_1.cx + 1*/(rt.right-rt.left)/2), sizeDibDisplay_2);
    // TODO: add draw code for native data here
}
```

分别注册两个按钮的行为

```
void CW02View::OnOpen()
{
    // TODO: Add your command handler code here
    CFileDialog FileDlg(TRUE, _T("*.bmp"), "", OFN_FILEMUSTEXIST);
    char title[] = { "Open Image" };
    FileDlg.m_ofn.lpstrTitle = title;

    CFile file;
    if (FileDlg.DoModal() == IDOK)
    {
        if (!file.Open(FileDlg.GetPathName(), CFile::modeRead))
        {
            AfxMessageBox("cannot open the file");
            return;
        }
        if (!mybmp_1.Read(&file))
        {
            AfxMessageBox("cannot read the file");
            return;
        }
    }

    if (mybmp_1.m_lpBMH->biCompression != BI_RGB)
    {
        AfxMessageBox("Can not read compressed file.");
        return;
    }
    sizeDibDisplay_1 = mybmp_1.GetDimensions();

    Invalidate(TRUE);
}

void CW02View::OnOpen1()
{
    // TODO: Add your command handler code here
    CFileDialog FileDlg(TRUE, _T("*.bmp"), "", OFN_FILEMUSTEXIST);
    char title[] = { "Open Image" };
    FileDlg.m_ofn.lpstrTitle = title;

    CFile file;
    if (FileDlg.DoModal() == IDOK)
    {
        if (!file.Open(FileDlg.GetPathName(), CFile::modeRead))
        {
            AfxMessageBox("cannot open the file");
            return;
        }
        if (!mybmp_2.Read(&file))
        {
            AfxMessageBox("cannot read the file");
            return;
        }
    }

    if (mybmp_2.m_lpBMH->biCompression != BI_RGB)
    {
        AfxMessageBox("Can not read compressed file.");
        return;
    }
    sizeDibDisplay_2 = mybmp_2.GetDimensions();

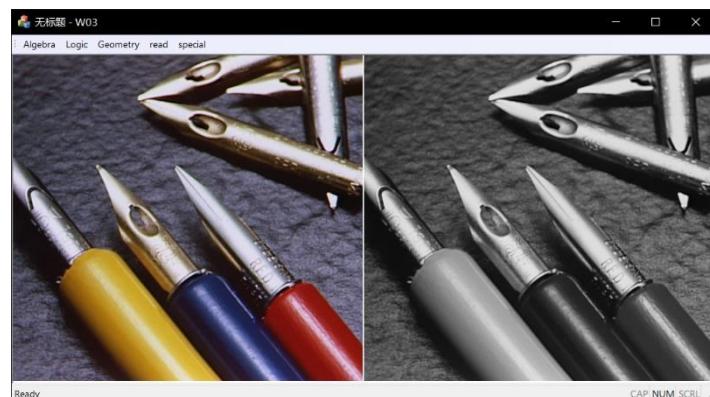
    Invalidate(TRUE);
}
```

得到结果

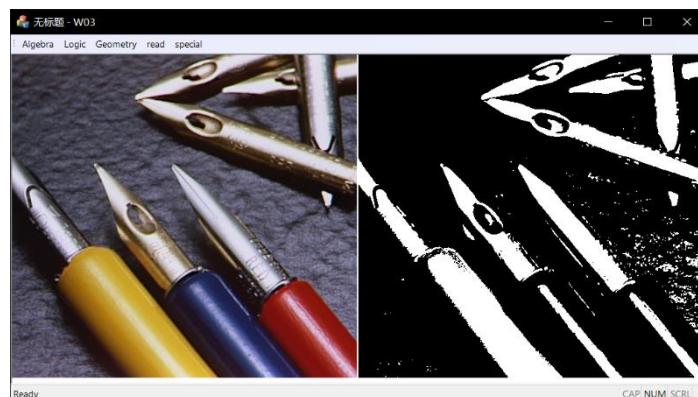


### W03

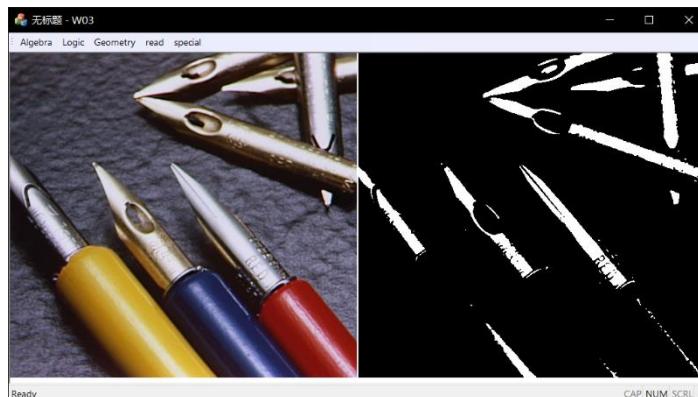
1、编程分别实现对下列图像进行灰度化和二值化处理。分别显示源图像和结果。



Threshold=125



Threshold=200





Threshold=125



Threshold=200



```

if (mybmp_1.IsEmpty())
{
    MessageBox("please read the first picture");
    return;
}
int sx = sizeDibDisplay_1.cx;
int sy = sizeDibDisplay_1.cy;
mybmp_output.CreateCObj(CSize(sx, sy), 24);
//对图像1灰度变换and二值化处理
for (int y = 0; y < sizeDibDisplay_1.cy; y++)
{
    // 每列
    for (int x = 0; x < sizeDibDisplay_1.cx; x++)
    {
        RGBQUAD color;
        color = mybmp_1.GetPixel(x, y);
        //RGB图像转灰度图像 Gray = R*0.299 + G*0.587 + B*0.114
        int gray = onGray(color);
        if (gray >= threshold) ? 255 : 0;
        color.rgbBlue = (unsigned char)gray;
        color.rgbGreen = (unsigned char)gray;
        color.rgbRed = (unsigned char)gray;
        mybmp_output.WritePixel(x, y, color);
    }
}

sizeDibDisplay_output = mybmp_output.GetDimensions();
Invalidate();

```

```

if (mybmp_1.IsEmpty())
{
    MessageBox("Please read the first picture");
    return;
}
int sx = sizeDibDisplay_1.cx;
int sy = sizeDibDisplay_1.cy;
mybmp_output.CreateCObj(CSize(sx, sy), 24);
//对图像1灰度变换and二值化处理
for (int y = 0; y < sizeDibDisplay_1.cy; y++)
{
    // 每列
    for (int x = 0; x < sizeDibDisplay_1.cx; x++)
    {
        RGBQUAD color;
        color = mybmp_1.GetPixel(x, y);
        //RGB图像转灰度图像 Gray = R*0.299 + G*0.587 + B*0.114
        int gray = onGray(color);
        gray = (gray >= threshold) ? 255 : 0;
        color.rgbBlue = (unsigned char)gray;
        color.rgbGreen = (unsigned char)gray;
        color.rgbRed = (unsigned char)gray;
        mybmp_output.WritePixel(x, y, color);
    }
}

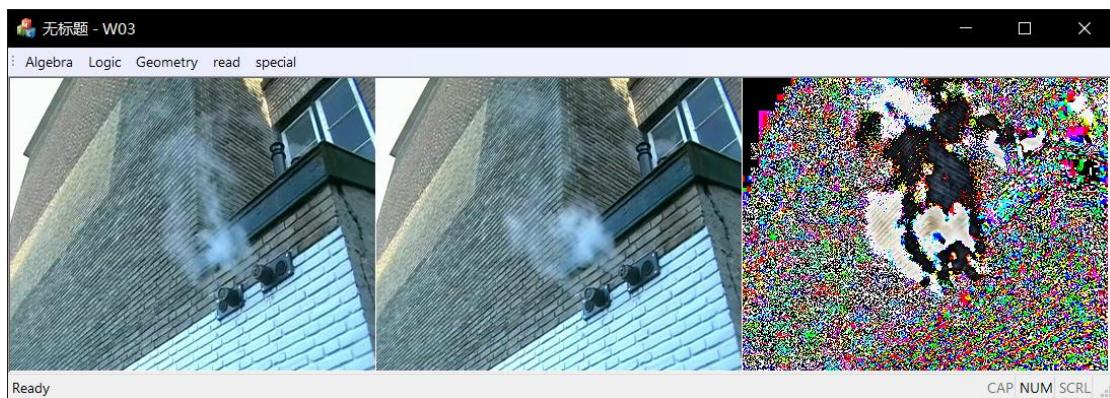
sizeDibDisplay_output = mybmp_output.GetDimensions();
Invalidate();

```

2、编程实现以下图像的相减运算，并分别显示源图像和结果。说明减运算的作用



结果：顺序为图 1-图 2-result



相关代码；作用为识别运动物体（如上图的特殊部分）

```
void CW03View::Onminus()
{
    // TODO: Add your command handler code here
    if (mybmp_1.IsEmpty() || mybmp_2.IsEmpty())
    {
        MessageBox("Please read two pictures");
        return;
    }
    int sx = (sizeDibDisplay_1.cx <= sizeDibDisplay_2.cx) ? sizeDibDisplay_1.cx : sizeDibDisplay_2.cx;
    int sy = (sizeDibDisplay_1.cy <= sizeDibDisplay_2.cy) ? sizeDibDisplay_1.cy : sizeDibDisplay_2.cy;
    mybmp_output.CreateCBitmap(CSize(sx, sy), 24);

    for (int y = 0; y < sy; y++)
    {
        // 每列
        for (int x = 0; x < sx; x++)
        {
            RGBQUAD color1;
            RGBQUAD color2;

            color1 = mybmp_1.GetPixel(x, y);
            color2 = mybmp_2.GetPixel(x, y);

            //合成图像
            RGBQUAD color;
            color.rgbBlue = color1.rgbBlue - color2.rgbBlue ;
            if (color.rgbBlue<0)
                color.rgbBlue = 0;

            color.rgbGreen = color1.rgbGreen - color2.rgbGreen;
            if (color.rgbGreen < 0)
                color.rgbGreen = 0;

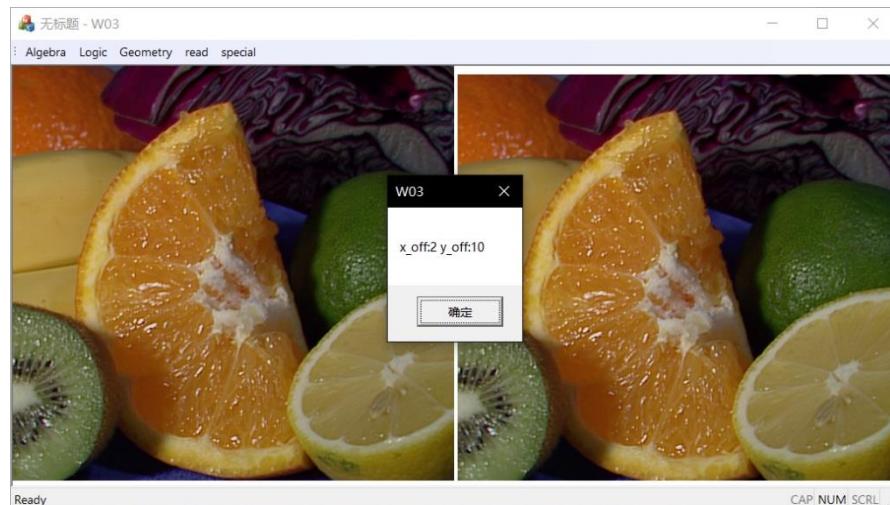
            color.rgbRed = color1.rgbRed - color2.rgbRed;
            if (color.rgbRed<0)
                color.rgbRed = 0;

            mybmp_output.WritePixel(x, y, color);
        }
    }
    sizeDibDisplay_output = mybmp_output.GetDimensions();
    Invalidate();
}
```

3、对以下图像进行平移操作，平移后图像的原点位于 (2,10) 位置，显示原图像和平移后的结果。



结果 (保持原图尺寸不留超出边界)



相关代码：

```
for (int y = 0; y < sizeDibDisplay_1.cy; y++)
{
    // 每列
    for (int x = 0; x < sizeDibDisplay_1.cx; x++)
    {
        RGBQUAD color;
        color = mybmp_1.GetPixel(x, y);

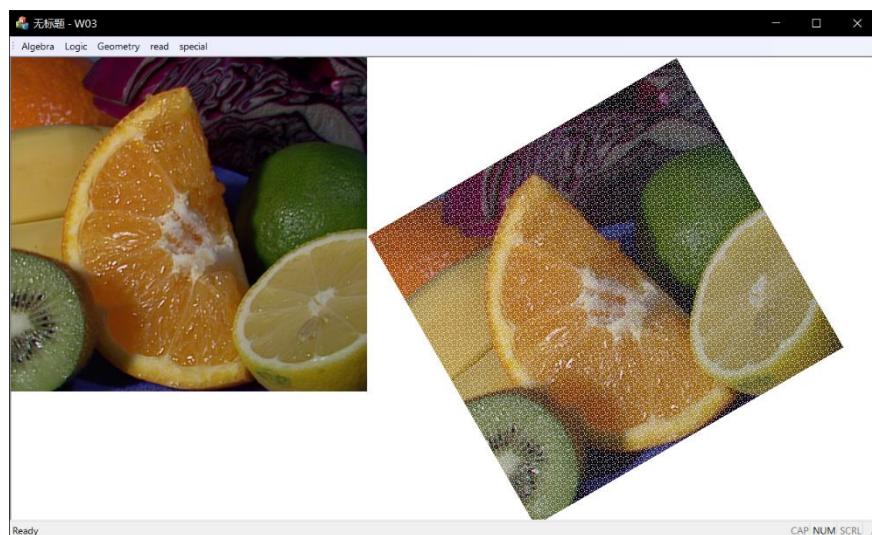
        // 计算该像素在源DIB中的坐标
        int x0 = x + off_x;
        int y0 = y + off_y;

        // 判断是否在源图范围内
        if (x0 >= sizeDibDisplay_1.cx || x0<0 || y0 >= sizeDibDisplay_1.cy || y0<0)
        {
            /*color.rgbGreen = 255;
            color.rgbRed = 255;
            color.rgbBlue = 255;*/
        }
        else {
            RGBQUAD _color;
            _color.rgbBlue = color.rgbBlue;
            _color.rgbGreen = color.rgbGreen;
            _color.rgbRed = color.rgbRed;
            mybmp_output.WritePixel(x0, y0, _color);
        }
    }
}
sizeDibDisplay_output = mybmp_output.GetDimensions();
Invalidate();
```

4、对于以下图像进行旋转 30 度，分别显示原图像和结果。



旋转之后出现噪点



主要代码

```
///////////////
// 源图四个角的坐标 (以图像中心为坐标系原点)
float fSrcX1, fSrcY1, fSrcX2, fSrcY2, fSrcX3, fSrcY3, fSrcX4, fSrcY4;
// 旋转后四个角的坐标 (以图像中心为坐标系原点)
float fDstX1, fDstY1, fDstX2, fDstY2, fDstX3, fDstY3, fDstX4, fDstY4;

long lWidth = sizeDibDisplay_1.cx;           // 获取图像的宽度
long lHeight = sizeDibDisplay_1.cy;          // 获取图像的高度

// 将旋转角度从度转换到弧度
float fRotateAngle = m_rotateangle*3.1415926535 / 180.0;      // 旋转角
float fSina, fCosa;                         // 旋转角度的正弦和余弦
fSina = (float)sin((double)fRotateAngle); // 计算旋转角度的正弦
fCosa = (float)cos((double)fRotateAngle); // 计算旋转角度的余弦

// 计算原图的四个角的坐标 (以图
fSrcX1 = (float)(-lWidth / 2);
fSrcY1 = (float)(lHeight / 2);
fSrcX2 = (float)(lWidth / 2);
fSrcY2 = (float)(lHeight / 2);
fSrcX3 = (float)(-lWidth / 2);
fSrcY3 = (float)(-lHeight / 2);
fSrcX4 = (float)(lWidth / 2);
fSrcY4 = (float)(-lHeight / 2);

// 计算新图四个角的坐标 (以图像中心为坐标系原点)
fDstX1 = fCosa * fSrcX1 + fSina * fSrcY1;
fDstY1 = -fSina * fSrcX1 + fCosa * fSrcY1;
fDstX2 = fCosa * fSrcX2 + fSina * fSrcY2;
fDstY2 = -fSina * fSrcX2 + fCosa * fSrcY2;
fDstX3 = fCosa * fSrcX3 + fSina * fSrcY3;
fDstY3 = -fSina * fSrcX3 + fCosa * fSrcY3;
fDstX4 = fCosa * fSrcX4 + fSina * fSrcY4;
fDstY4 = -fSina * fSrcX4 + fCosa * fSrcY4;

// 图像中心
```

```

// 计算旋转后的图像实际宽度
long lNewwidth = (long)(max(fabs(fDstX4 - fDstX1), fabs(fDstX3 - fDstX2)) + 0.5);
// 计算旋转后的图像高度
long lNewHeight = (long)(max(fabs(fDstY4 - fDstY1), fabs(fDstY3 - fDstY2)) + 0.5);

mybmp_output.Empty();
mybmp_output.CreateCBitmap(CSize(lNewwidth, lNewHeight), mybmp_1.m_lpBMH->biBitCount);

sizeDibDisplay_output = mybmp_output.GetDimensions();

RGBQUAD color;

// 每行
for (int y = 0; y < sizeDibDisplay_1.cy; y++)
{
    // 每列
    for (int x = 0; x < sizeDibDisplay_1.cx; x++)
    {
        color = mybmp_1.GetPixel(x, y);

        // 计算该像素在原DIB中的坐标
        float x0 = fCosa * (x - sizeDibDisplay_1.cx / 2.0) + fSina * (y - sizeDibDisplay_1.cy / 2.0)+0.5;
        float y0 = -fSina * (x - sizeDibDisplay_1.cx / 2.0) + fCosa * (y - sizeDibDisplay_1.cy / 2.0)+0.5;

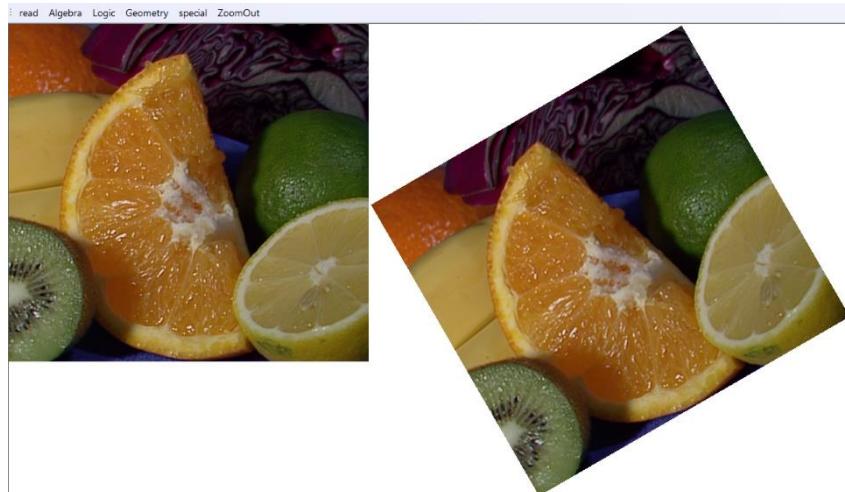
        x0 = x0 + lNewwidth / 2+0.5;
        y0 = y0 + lNewHeight / 2+0.5;

        mybmp_output.WritePixel((int)x0, (int)y0, color);
    }
}

Invalidate();

```

采用双线性插值实现的后向映射之后，可以去噪点得到完整的图像：



核心代码：

```

float x0 = (x - lNewwidth / 2) * fCosa - (y - lNewHeight / 2) * fSina + sizeDibDisplay_1.cx / 2.0;
float y0 = (x - lNewwidth / 2) * fSina + (y - lNewHeight / 2) * fCosa + sizeDibDisplay_1.cy / 2.0;
float cx = x0;
float cy = y0;
if ((x0 >= 1) && (x0 < sizeDibDisplay_1.cx-1) && (y0 >= 1) && (y0 < sizeDibDisplay_1.cy-1))
{
    // 双线性插值
    // f((1+u,j+v) = (1-u)(1-v)f(i,j) + (1-u)vf(i,j+1) + u(1-v)f(i+1,j) + uvf(i+1,j+1)

    float u = cx - (int)cx;
    float v = cy - (int)cy;
    int i = (int)cx;
    int j = (int)cy;

    int red;
    red = (1 - u)*(1 - v)*mybmp_1.GetPixel(i, j).rgbRed + (1 - u)*v*mybmp_1.GetPixel(i, j + 1).rgbRed
        + u*(1 - v)*mybmp_1.GetPixel(i + 1, j).rgbRed + u*v*mybmp_1.GetPixel(i + 1, j + 1).rgbRed;
    int green;
    green = (1 - u)*(1 - v)*mybmp_1.GetPixel(i, j).rgbGreen + (1 - u)*v*mybmp_1.GetPixel(i, j + 1).rgbGreen
        + u*(1 - v)*mybmp_1.GetPixel(i + 1, j).rgbGreen + u*v*mybmp_1.GetPixel(i + 1, j + 1).rgbGreen;
    int blue;
    blue = (1 - u)*(1 - v)*mybmp_1.GetPixel(i, j).rgbBlue + (1 - u)*v*mybmp_1.GetPixel(i, j + 1).rgbBlue
        + u*(1 - v)*mybmp_1.GetPixel(i + 1, j).rgbBlue + u*v*mybmp_1.GetPixel(i + 1, j + 1).rgbBlue;

    color.rgbGreen = green;
    color.rgbRed = red;
    color.rgbBlue = blue;
}
else
{
    color.rgbGreen = 255;
    color.rgbRed = 255;
    color.rgbBlue = 255;
}
mybmp_output.WritePixel(x, y, color);

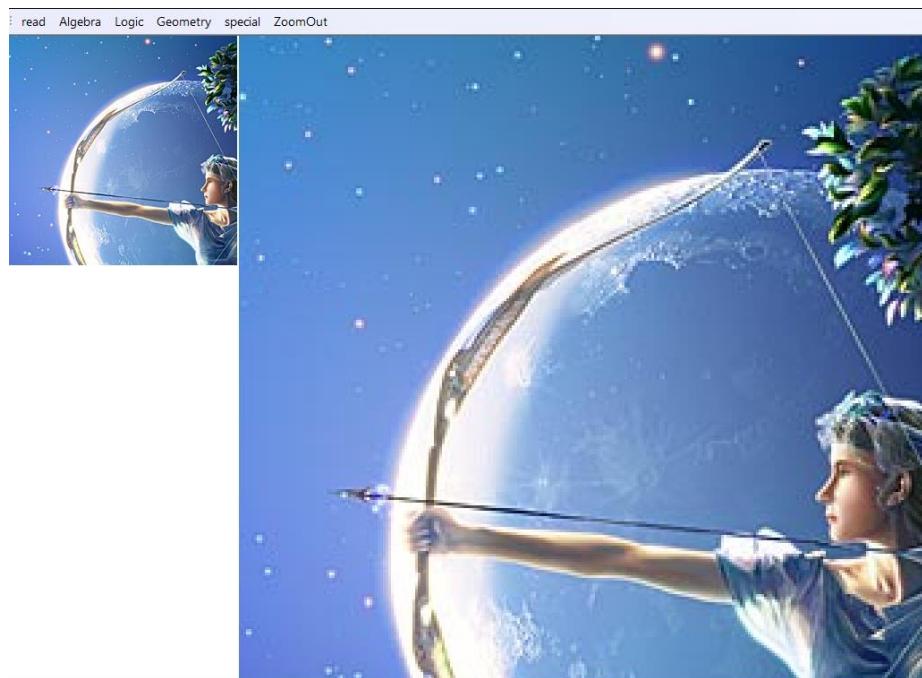
```

## W04

1、对以下图像利用前向映射法放大 3 倍，分别显示原图像和结果。



仅仅向后送颜色，未用行插值或列插值方法处理：





核心代码如下

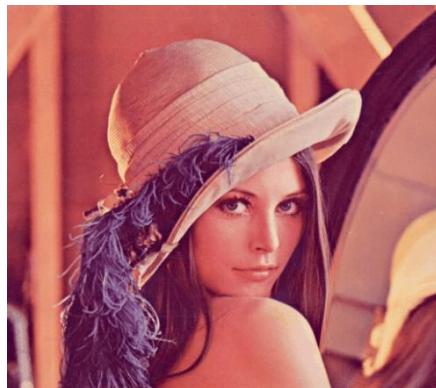
```
long lNewWidth = (long)(sizeDibDisplay_1.cx*fXZoomRatio);
long lNewHeight = (long)(sizeDibDisplay_1.cy*fYZoomRatio);

//mybmp_output.Empty();
mybmp_output.CreateCBitmap(CSize(lNewWidth, lNewHeight), 24);
sizeDibDisplay_output = mybmp_output.GetDimensions();

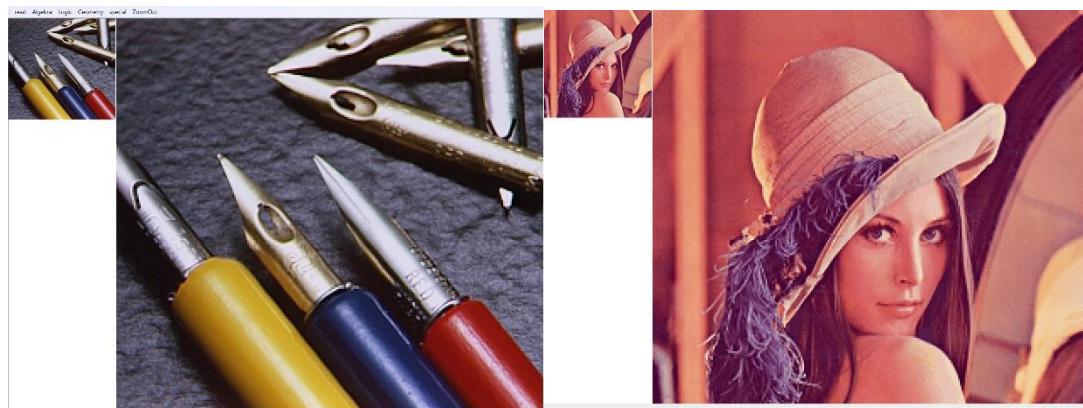
RGBQUAD color;
//向后送颜色
// 每行
for (int y = 0; y < sizeDibDisplay_1.cy; y++)
{
    // 每列
    for (int x = 0; x < sizeDibDisplay_1.cx; x++)
    {
        color = mybmp_1.GetPixel(x, y);
        //计算前向映射中四个点的位置
        int x0, x1, y0, y1;
        x0 = x*fXZoomRatio;
        x1= x*fXZoomRatio+0.5;
        y0 = y*fYZoomRatio;
        y1 = y*fYZoomRatio + 0.5;

        if (x0 < lNewWidth&&y0 < lNewHeight)
        {
            mybmp_output.WritePixel((int)x0, (int)y0, color);
        }
        if (x0 < lNewWidth&&y1 < lNewHeight)
        {
            mybmp_output.WritePixel((int)x0, (int)y1, color);
        }
        if (x1 < lNewWidth&&y0 < lNewHeight)
        {
            mybmp_output.WritePixel((int)x1, (int)y0, color);
        }
        if (x1 < lNewWidth&&y1 < lNewHeight)
        {
            mybmp_output.WritePixel((int)x1, (int)y1, color);
        }
    }
    Invalidate();
}
```

2、对以下图像利用后向映射法放大 4 倍，分别显示原图像和结果。



Ans:利用最邻近值法实现后向映射



代码如下：

```
sizeDibDisplay_output = mybmp_output.GetDimensions();
RGBQUAD color;

//向前取颜色
// 每行
for (int y = 0; y < lNewHeight; y++)
{
    // 每列
    for (int x = 0; x < lNewWidth; x++)
    {
        long x0 = (x / fXZoomRatio+0.5);
        long y0 = (y / fYZoomRatio+0.5);
        if ((x0 >= 0) && (x0 < sizeDibDisplay_1.cx) && (y0 >= 0) && (y0 < sizeDibDisplay_1.cy))
        {
            color = mybmp_1.GetPixel(x0, y0);
        }
        else {
            color.rgbRed = 255;
            color.rgbGreen = 255;
            color.rgbBlue = 255;
        }
        mybmp_output.WritePixel(x, y, color);
    }
}
```

3、比较前两题中，这两种放大方法效果的差异。

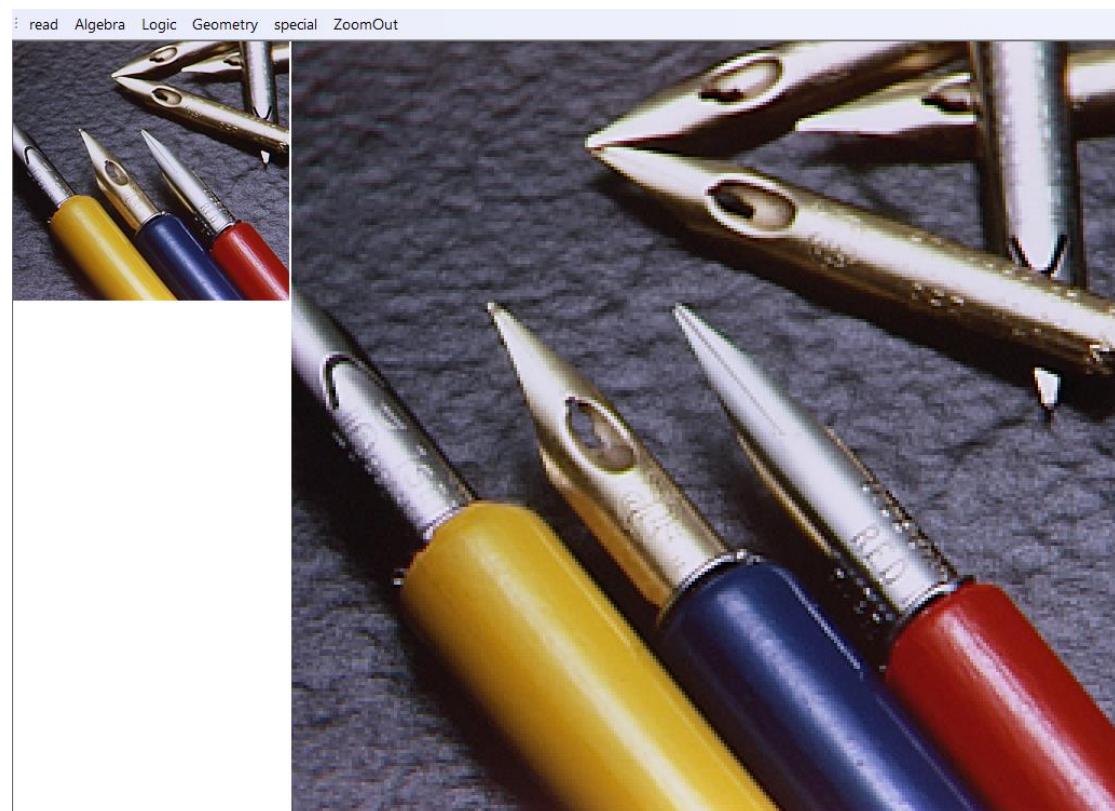
差异首先就是前向映射法是需要行列的多次差值操作来完成图像的填充的（否则会存在白点），但是后向映射法则循环一遍目标图像的点集就可以完成，更加节省时间，效果也更好。前两题中放大结果的差异更多体现在图像色彩的锯齿效果，也意味着前向映射法生成的图片更容易失真。

4、对于下面图像利用后向映射法放大 3 倍时，分别采用最近邻插值和双线性插值完成放大功能，分别显示原图像和结果。

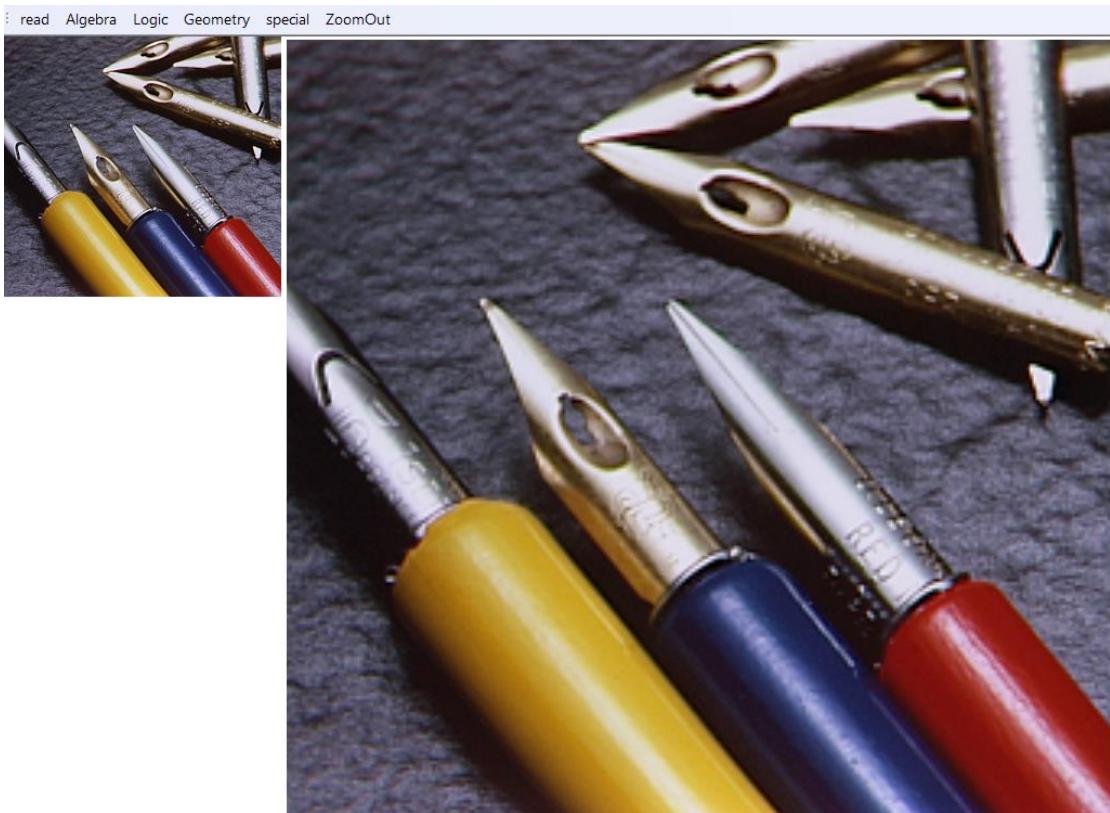


Ans:

最近邻插值：



双线性插值：



上题以附上最近邻插值代码，以下附上

```
//向前取颜色
// 每行
for (int y = 0; y < lNewHeight; y++)
{
    // 每列
    for (int x = 0; x < lNewWidth; x++)
    {
        float cx = x / fXZoomRatio;
        float cy = y / fYZoomRatio;

        if (((int)(cx)-1) >= 0 && ((int)(cx)+1) < sizeDibDisplay_1.cx && ((int)(cy)-1) >= 0 && ((int)(cy)+1) < sizeDibDisplay_1.cy)
        {
            //f(i+u,j+v) = (1-u)(1-v)f(i,j) + (1-u)vf(i,j+1) + u(1-v)f(i+1,j) + uvf(i+1,j+1)
            float u = cx - (int)cx;
            float v = cy - (int)cy;
            int i = (int)cx;
            int j = (int)cy;

            int red;
            red = (1 - u)*(1 - v)*mybmp_1.GetPixel(i, j).rgbRed + (1 - u)*v*mybmp_1.GetPixel(i, j + 1).rgbRed
                + u*(1 - v)*mybmp_1.GetPixel(i + 1, j).rgbRed + u*v*mybmp_1.GetPixel(i + 1, j + 1).rgbRed;
            int green;
            green = (1 - u)*(1 - v)*mybmp_1.GetPixel(i, j).rgbGreen + (1 - u)*v*mybmp_1.GetPixel(i, j + 1).rgbGreen
                + u*(1 - v)*mybmp_1.GetPixel(i + 1, j).rgbGreen + u*v*mybmp_1.GetPixel(i + 1, j + 1).rgbGreen;
            int blue;
            blue = (1 - u)*(1 - v)*mybmp_1.GetPixel(i, j).rgbBlue + (1 - u)*v*mybmp_1.GetPixel(i, j + 1).rgbBlue
                + u*(1 - v)*mybmp_1.GetPixel(i + 1, j).rgbBlue + u*v*mybmp_1.GetPixel(i + 1, j + 1).rgbBlue;

            color.rgbGreen = green;
            color.rgbRed = red;
            color.rgbBlue = blue;
        }
        else
        {
            color.rgbGreen = 255;
            color.rgbRed = 255;
            color.rgbBlue = 255;
        }
    }
}
```

5、根据以上实践结果，请说明图像放大处理应该怎样处理，能够得到较满意的结果。

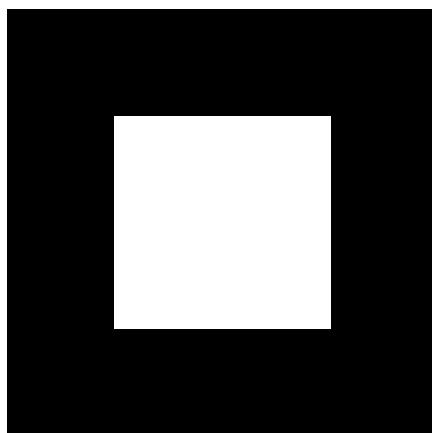
Ans：为了保证图片放大的质量和节省性能，选择双线性插值法实现的后向映射。这样得到的图像比起前向映射完整，不需要多次的行列差值。也比最靠近插值实现的后向映射得到的放大图片更平滑，图片质量更高。

## W05

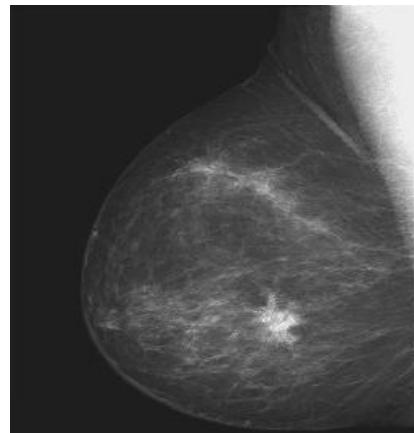
以下作业要求自己新建工程，将老师给的傅里叶变换的代码嵌入你的工程中使用。

1、对下列图像进行傅里叶变换，并分析傅里叶变换频谱的特点。

(1) Retangle



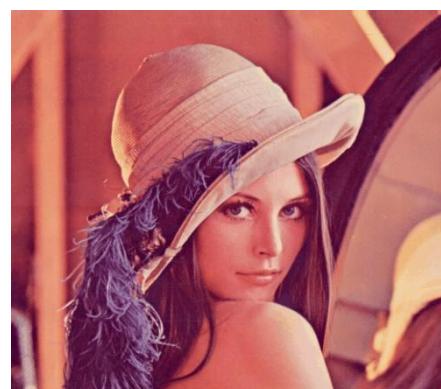
(2) Enhance



(3) Flower

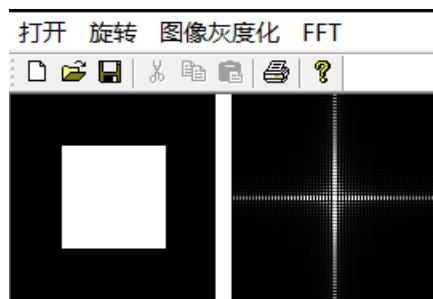


(4)Lenna

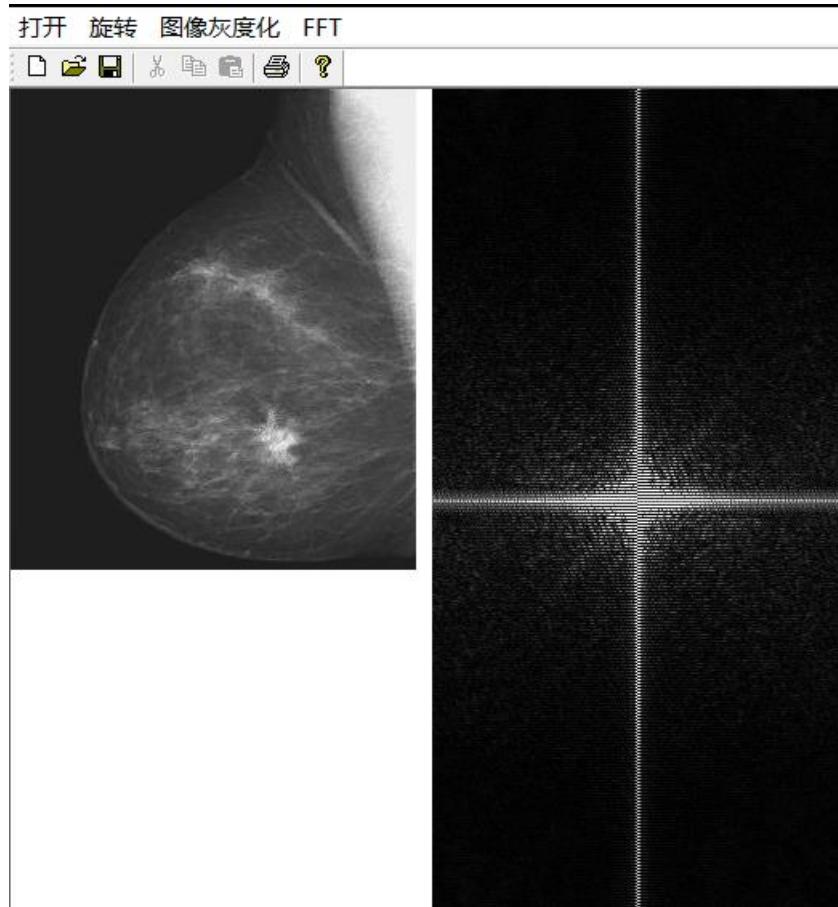


Ans:

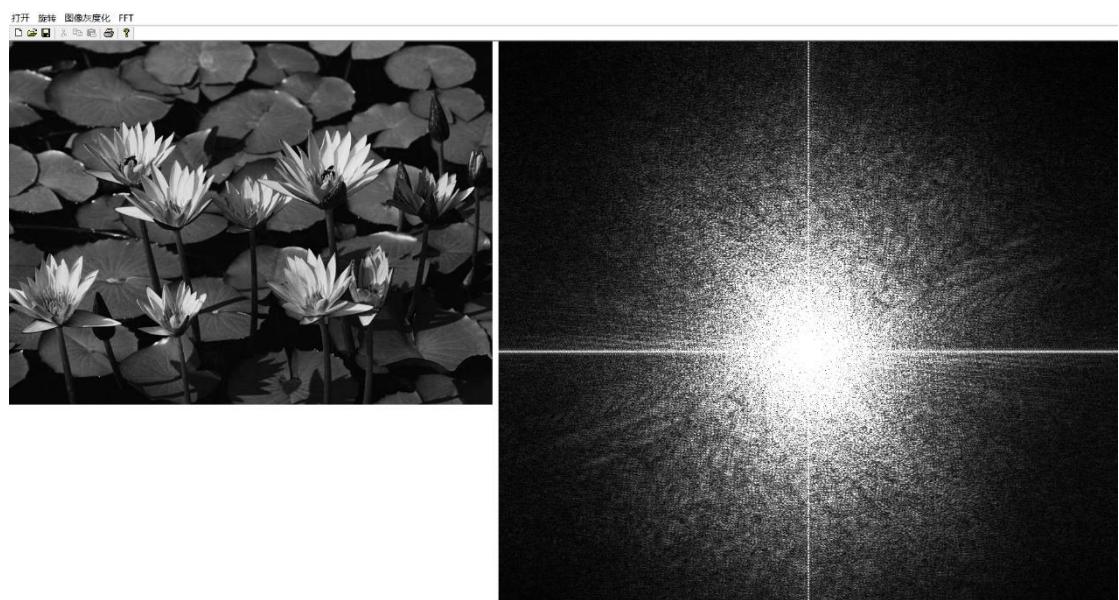
[1] Retangle



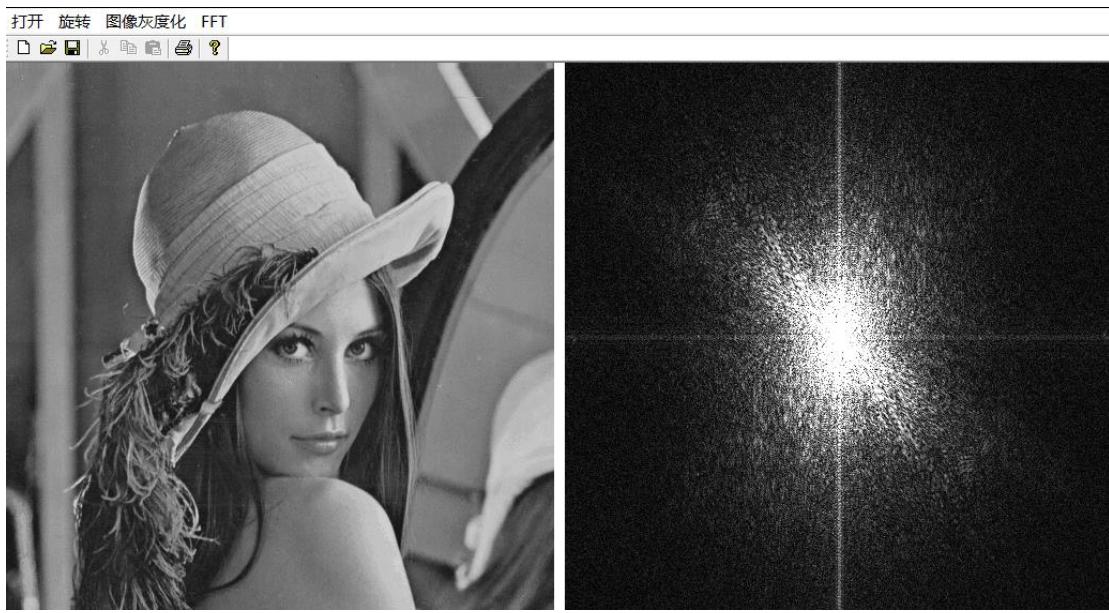
[2] Enhance



[3] Flower



[4] Lenna

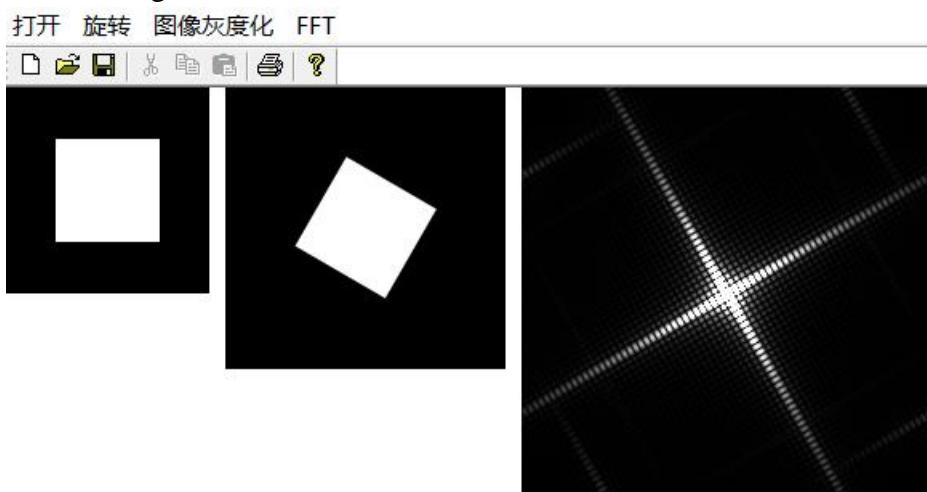


特点是可以将图像转化到傅里叶频域中进行处理，然后再反变换得到图像压缩降噪的效果。在傅里叶频谱中，四个角上的谱表示图像中的低频成分，中心的代表高频成分。在中心化后，中心区域频谱表示图像中低频成分，而四个角上的谱是图像的高频成分。

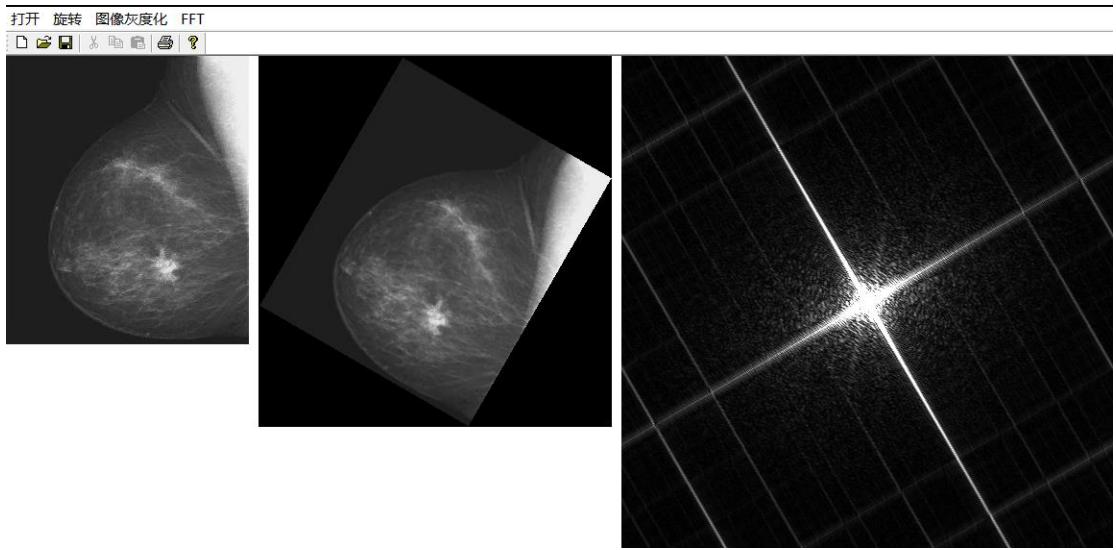
2、对于题目 1 中的所有图像顺时针旋转 30 度；然后再对他们进行傅里叶变换，显示原图像、旋转图像及变换后的结果。

由于旋转 30 度会对边界大小产生影响，所以选择用黑色填充边界部分

#### [5] Rectangle

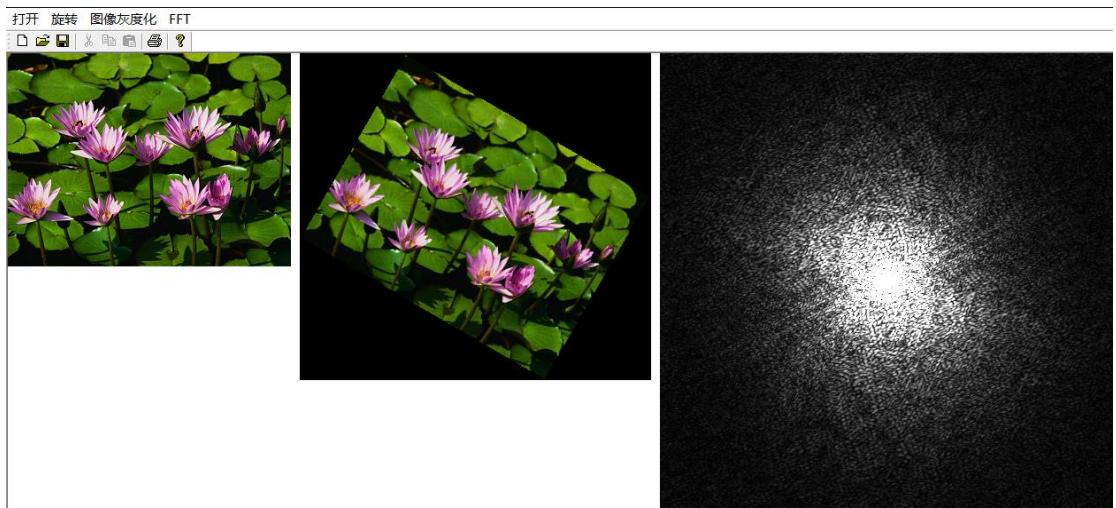


#### [6] Enhance



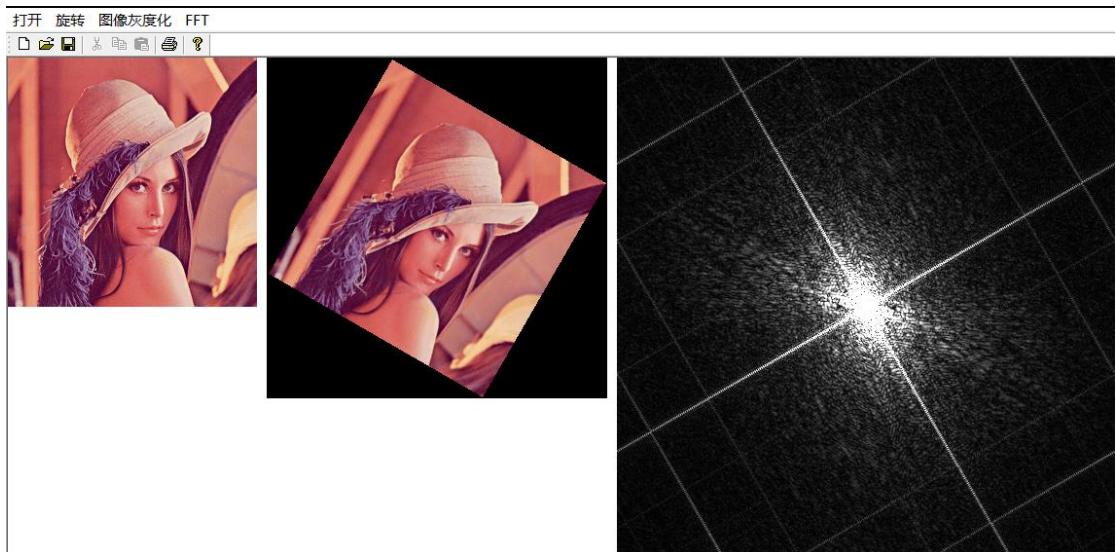
### [7] Flower

由于 Flower 过大显示不下三张图，对其进行缩小处理



### [8] Lenna

由于 Lenna 过大显示不下三张图，对其进行缩小处理



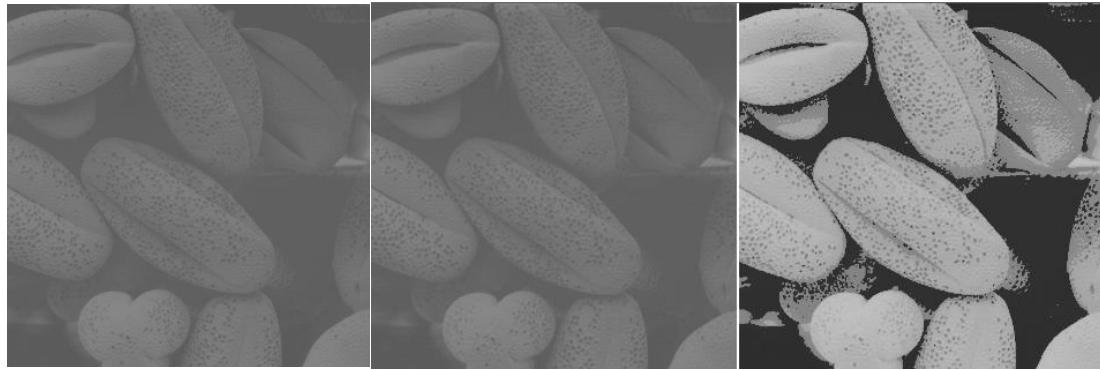
3、对题目 2 的变换结果进行分析，你能得出什么结论。

Ans：傅里叶变换具有旋转不变性，即如果对原图像旋转一个角度，其傅里叶变换也旋转相同的角度

## W06

1、利用图像的线性点运算(可以是分段的)，完成对下列图像的增强处理。  
处理后显示原图像和你处理的结果。

(1) Nut-右边为结果



线性分段

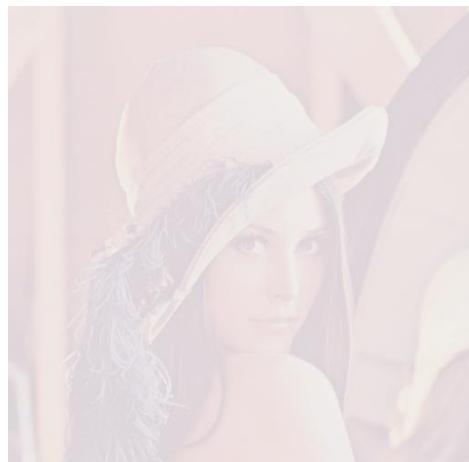
```
if (g >= 110)
{
    a = 1.2;
    b = 30;
    g = a * g + b;

    if (g >= 255)
        g = 255;
}
else if (g<88)
{
    a = 0.2;
    b = 0;
    g = a * g + b;
}

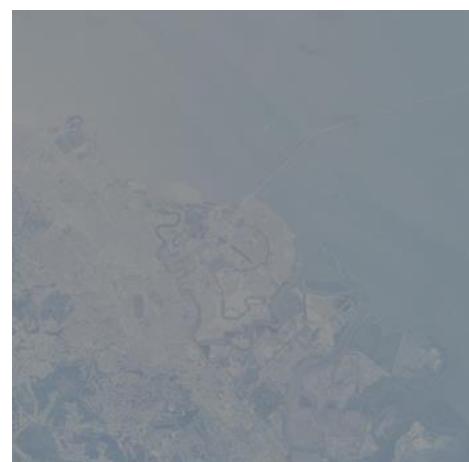
else if (g>88 && g<100)
{
    a = 0.5;
    b = 0;
    g = a * g + b;
}
else if (g>100 && g<110)
{
    a = 1.3;
    b = 0;
    g = a * g + b;
```

2、对下列图像先进行灰度化处理,再分别利用图像的指数及幂运算,完成对下列图像的增强处理。

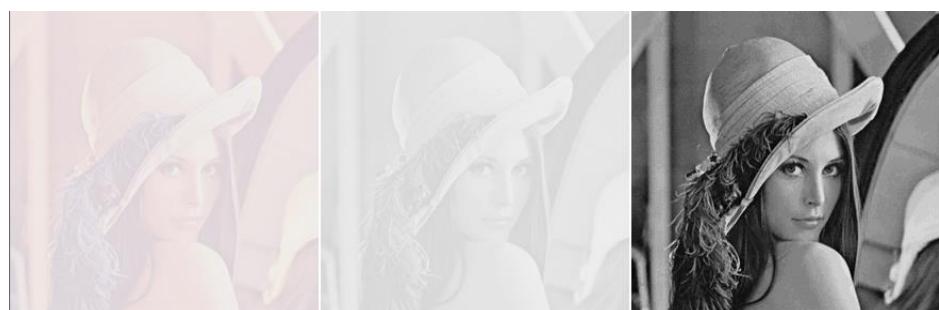
(1) Lenna



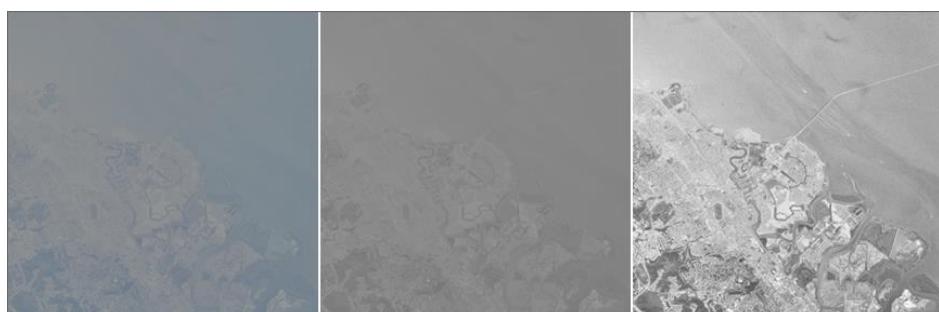
(2) Aerial photography



Result1: 参数 c=4.8 a=1.1



Result2: 参数 c=3.3, a=1.1



参考代码:

```
    ...
    for (int y = 0; y < sizeDibDisplay_1.cy; y++)
    {
        // 每列
        for (int x = 0; x < sizeDibDisplay_1.cx; x++)
        {
            RGBQUAD color;
            color = mybmp_gray.GetPixel(x, y);
            //s = ag + b
            double g = color.rgbRed;
            double c = 3.1;
            double a = 1.1;
            g = c * pow(g, a);

            color.rgbBlue = (unsigned char)g;
            color.rgbGreen = (unsigned char)g;
            color.rgbRed = (unsigned char)g;
            mybmp_output.WritePixel(x, y, color);
        }
    }
```

3、通过上述题目 1 和 2 的增强过程，你总结一下利用空域点运算对图像增强时，参数选择应该注意哪些问题？

参数选择可以选择最暗处的点和最亮处的点需要调整到的颜色值，用二维方程组来解出两个参数的值。防止过亮或者过暗的情况的出现。

4、利用直方图均衡化方法、对以下图像进行增强处理，如果是彩色图像，先进行灰度化预处理，然后再均衡化。对比均衡化前后图像的处理效果。

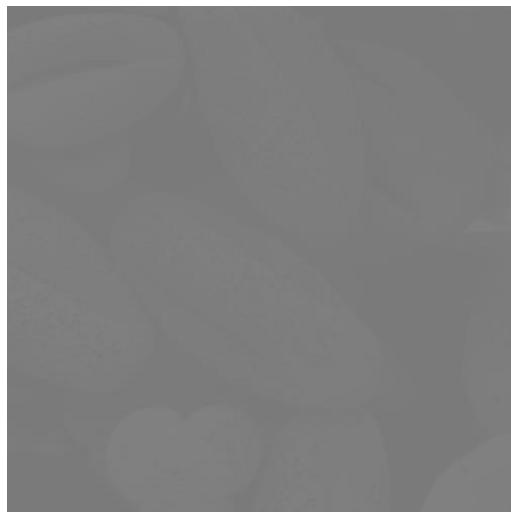
(1) Airplane



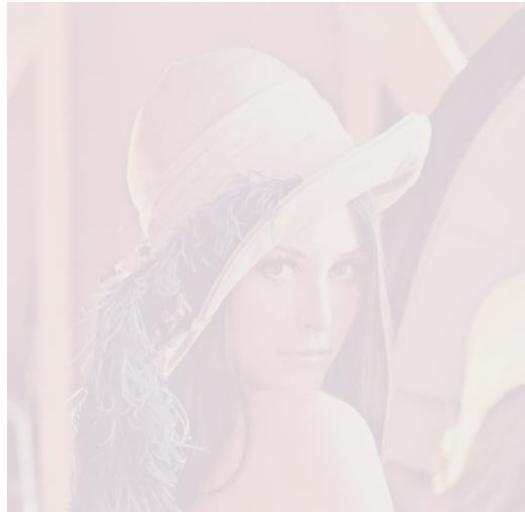
(2) Man



(3) Nut



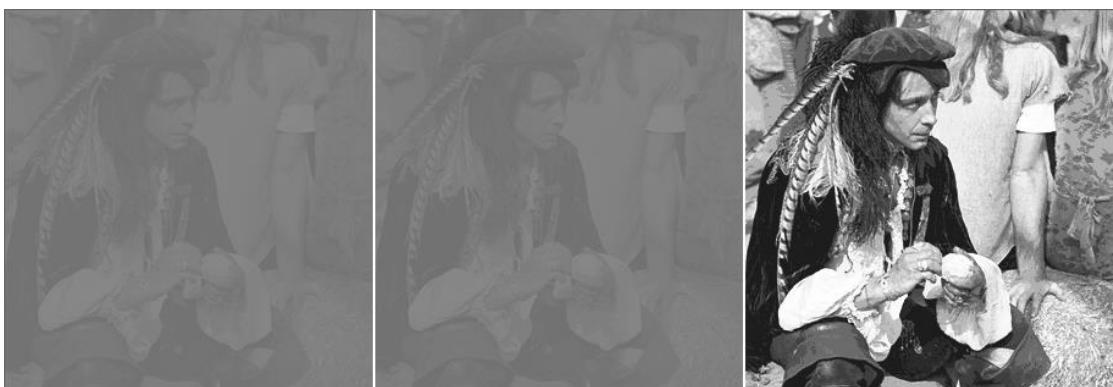
(4) Lenna



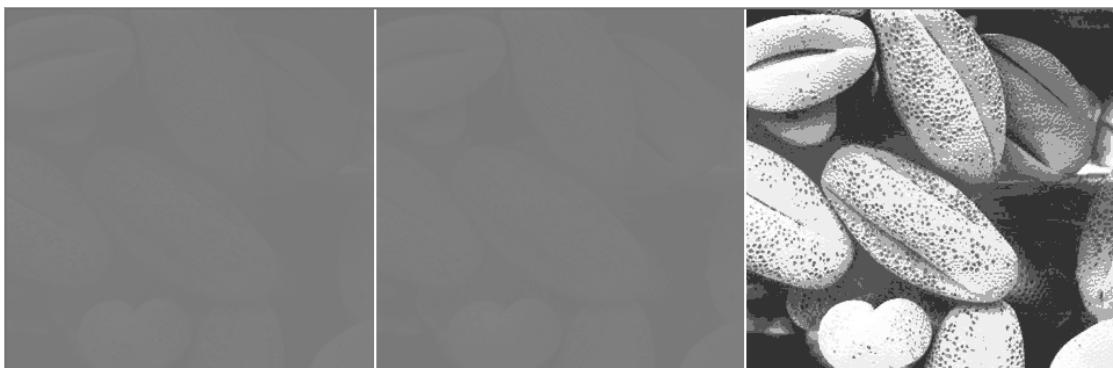
Result1:



Result2:



Result3:



Result4:



实现代码:

```
for (int y = 0; y < sizeDibDisplay_1.cy; y++)
{
    // 每行
    for (int x = 0; x < sizeDibDisplay_1.cx; x++)
    {
        RGBQUAD color;
        color = mybmp_gray.GetPixel(x, y);
        int gray = color.rgbBlue;

        count[gray]++;
    }
}
for (int i = 0; i < 256; i++)
{
    int lTemp = 0;
    for (int j = 0; j <= i; j++)
    {
        lTemp += count[j];
    }

    // 计算对应的新灰度值
    newgray[i] = (lTemp * 255.0 / sizeDibDisplay_1.cy / sizeDibDisplay_1.cx);
}

for (int y = 0; y < sizeDibDisplay_1.cy; y++)
{
    // 每行
    for (int x = 0; x < sizeDibDisplay_1.cx; x++)
    {
        RGBQUAD color;
        color = mybmp_gray.GetPixel(x, y);
        int gray = newgray[color.rgbBlue];
        color.rgbBlue = (unsigned char)gray;
        color.rgbGreen = (unsigned char)gray;
        color.rgbRed = (unsigned char)gray;
        mybmp_output.WritePixel(x, y, color);
    }
}
```

5、通过对上述题目中图像的直方图均衡化增强处理，你总结一下为什么直方图均衡化能够对图像起到增强的作用？

因为直方图可以将色彩分布到 0~255，增加像素之间的灰度差。实质为：

- (1) 减少图像的灰度级以换取对比度的加大；
- (2) 在均衡过程中，原来的直方图上频数较小的灰度级被归入很少几个或一个灰度级内；
- (3) 直方图均衡化使图像中的最大灰度与最小灰度的比值增大，因此图像变得更加清晰。

6、分别利用强度分层法和灰度变换法，对以下图像进行伪彩色化处理。

(1) Airplane



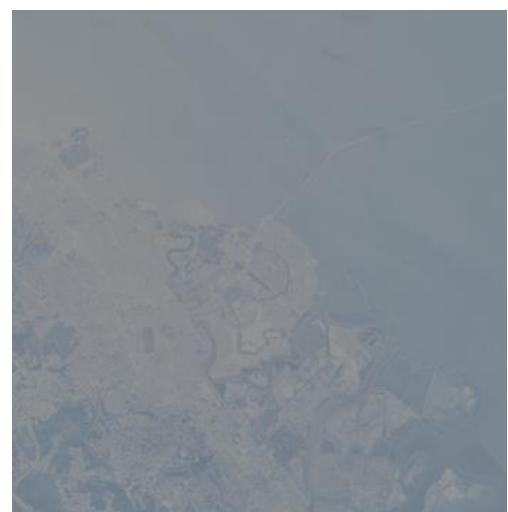
(2) Man



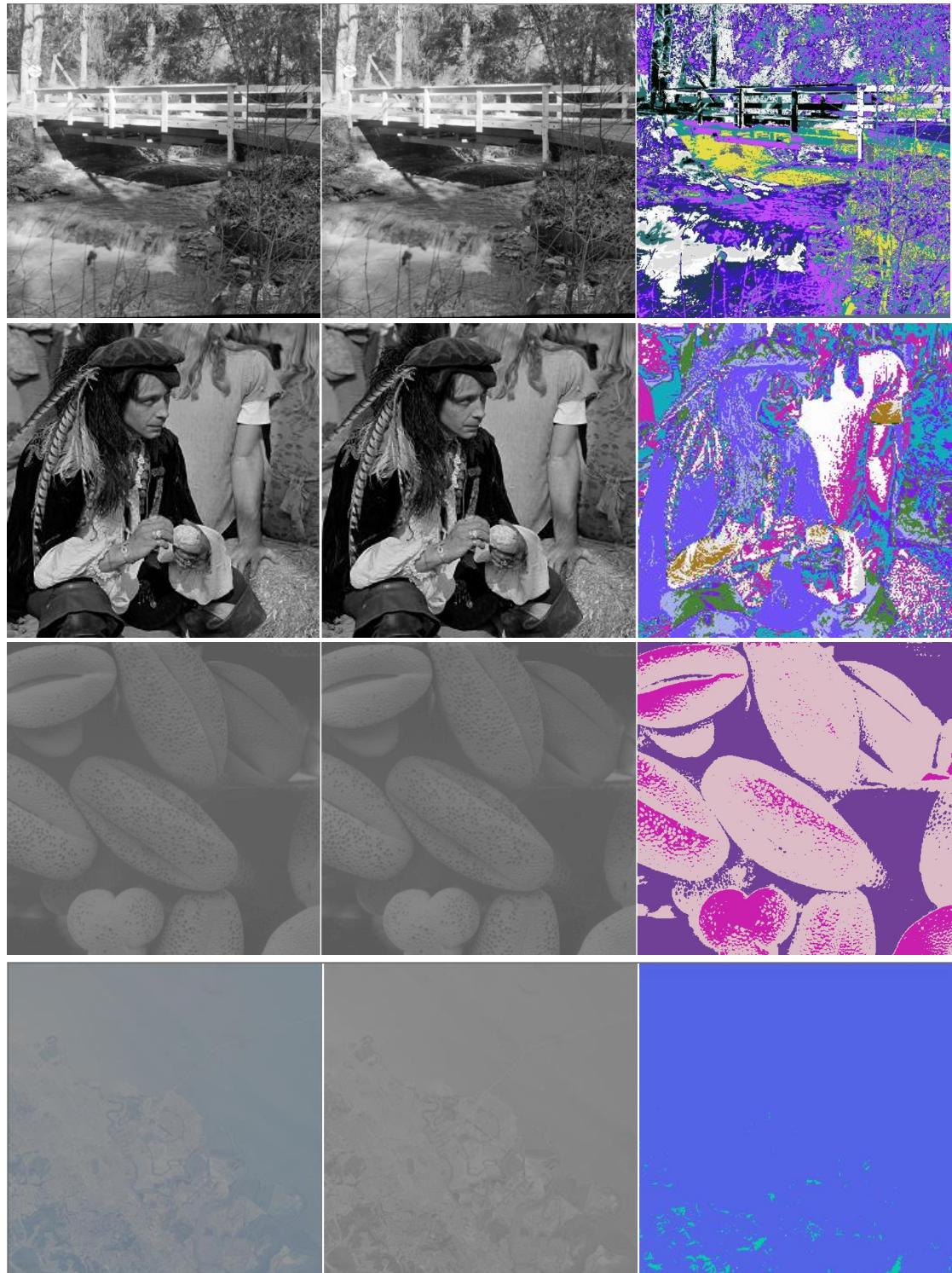
(3) Nut



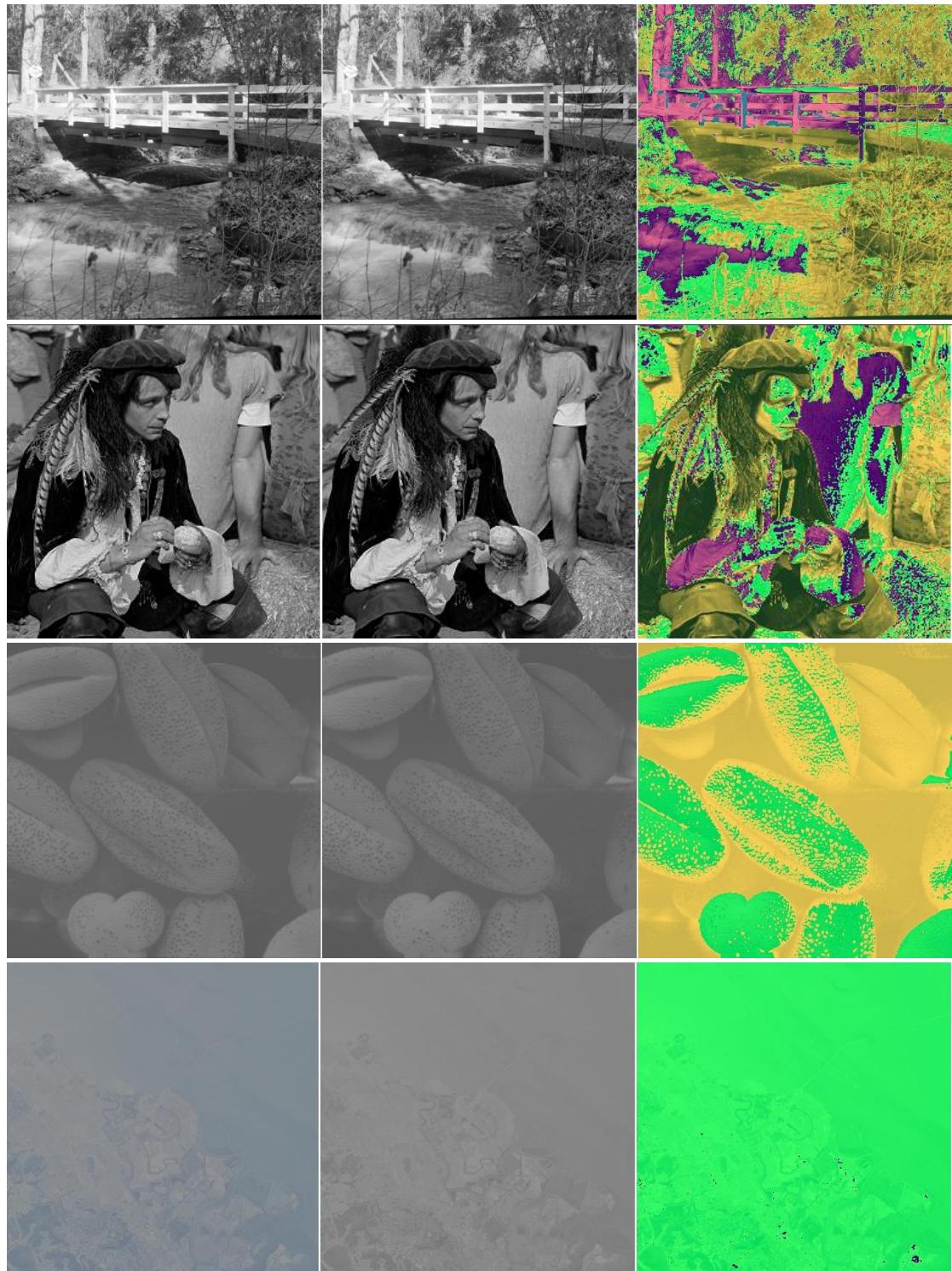
(4) Aerial photography



强度分层法:



灰度变换法：



### W07

1、利用均值滤波方法对下列图像进行平滑处理，处理前后的图像分别显示出来。

(1) lenna



(2) Flower



Result1:



## 核心代码修改为彩色图像与灰色图像通用模板

```
int temp[3][3] = { 1,1,1,1,1,1,1,1,1 }; // 3*3模板
double tempC = 1.0 / 9; //模板系数

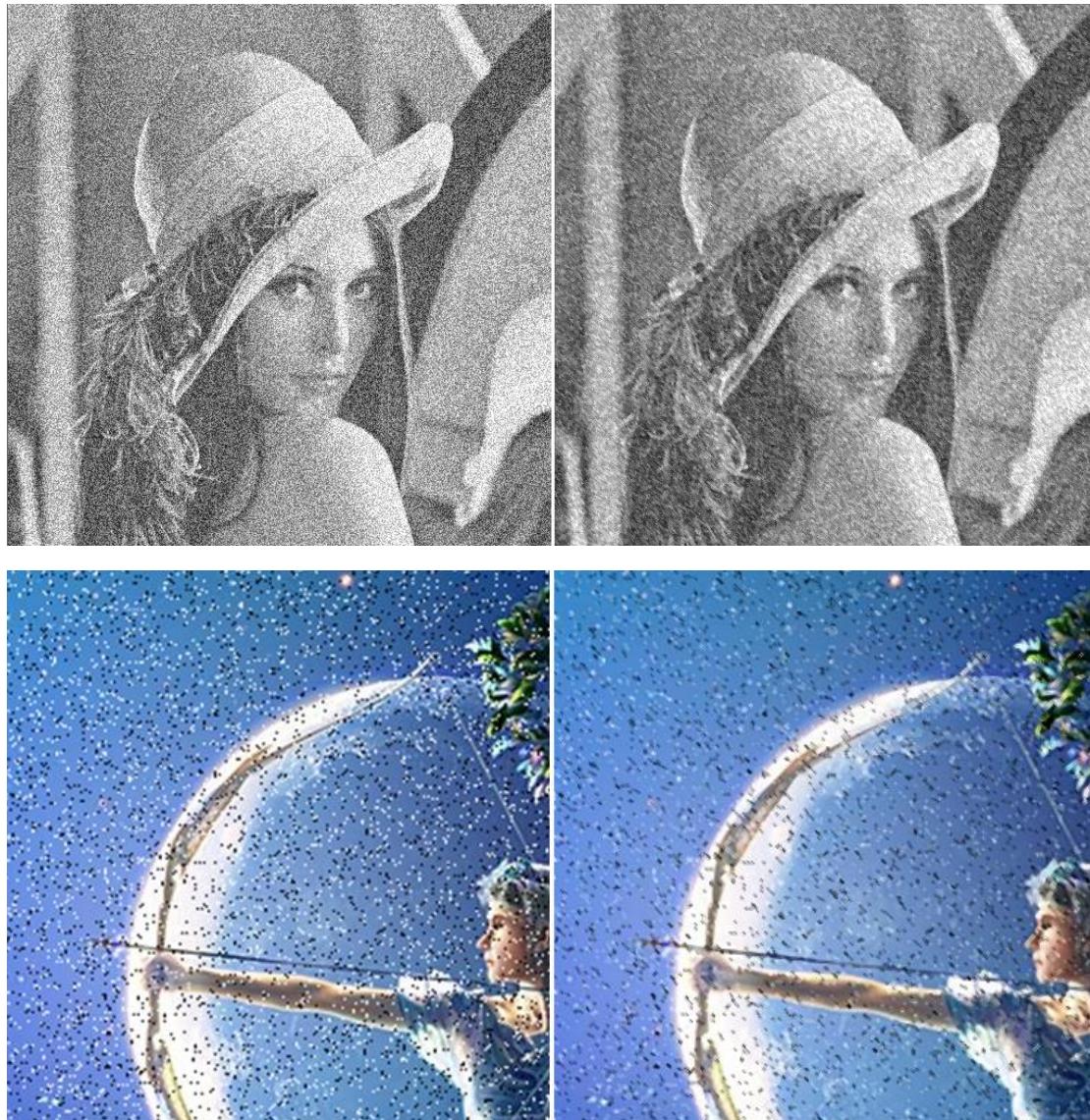
for (int x = 0 + 1; x < sizeDibDisplay_output.cx - 1; x++)
{
    for (int y = 0 + 1; y < sizeDibDisplay_output.cy - 1; y++)
    {
        RGBQUAD color;

        double r = 0;
        double g = 0;
        double b = 0;

        for (int i = -1; i <= 1; i++)
            for (int j = -1; j <= 1; j++)
            {
                color = mybmp_1.GetPixel(x + i, y + j);
                r += color.rgbRed*temp[i + 1][j + 1] * tempC;
                g += color.rgbGreen*temp[i + 1][j + 1] * tempC;
                b += color.rgbBlue*temp[i + 1][j + 1] * tempC;
            }

            color.rgbBlue = (int)b;
            color.rgbGreen = (int)g;
            color.rgbRed = (int)r;
            mybmp_output.WritePixel(x, y, color);
    }
}
```

2、利用中值滤波方法对第 1 题中的图像进行平滑处理，显示处理前后的图像。



核心代码为

```
for (int x = 0 + 1; x < sizeDibDisplay_output.cx - 1; x++)
{
    for (int y = 0 + 1; y < sizeDibDisplay_output.cy - 1; y++)
    {
        RGBQUAD color;

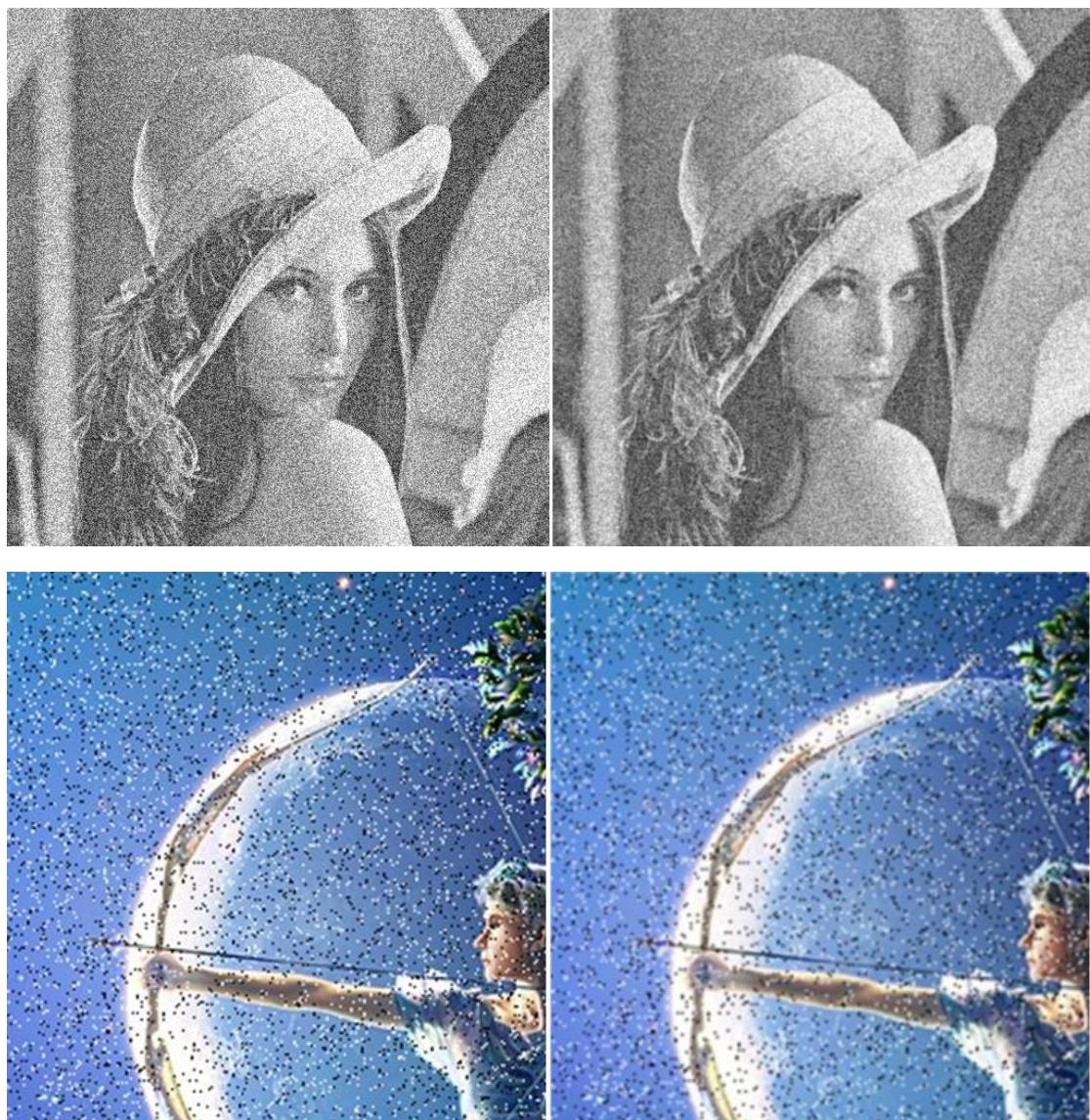
        int r[9];
        int g[9] = { 0 };
        int b[9] = { 0 };
        int k = 0;
        for (int i = -1; i <= 1; i++)
            for (int j = -1; j <= 1; j++)
            {
                color = mybmp_1.GetPixel(x + i, y + j);
                r[k] = color.rgbRed;
                g[k] = color.rgbGreen;
                b[k] = color.rgbBlue;
                k++;
            }
        int _r = GetMedianNum(r, 9);
        int _g = GetMedianNum(g, 9);
        int _b = GetMedianNum(b, 9);

        color.rgbBlue = (int)_b;
        color.rgbGreen = (int)_g;
        color.rgbRed = (int)_r;
        mybmp_output.WritePixel(x, y, color);
    }
}
```

```
int cmp(const void * a,const void *b)
{
    int* _a = (int*)a;
    int* _b = (int*)b;
    return *_a < *_b;
}

int Cw07View::GetMedianNum(int * num, int i)
{
    qsort(num, i, sizeof(int), cmp);
    return num[(i + 1) / 2-1];
}
```

3、利用加权平滑滤波方法对第 1 题中的图像进行平滑处理，显示处理前后的图像。



核心代码为（拓展为彩色坐标系中）：

```
for (int x = 0 + 1; x < sizeDibDisplay_output.cx - 1; x++)
{
    for (int y = 0 + 1; y < sizeDibDisplay_output.cy - 1; y++)
    {
        RGBQUAD color;

        double r = 0;
        double g = 0;
        double b = 0;

        for (int i = -1; i <= 1; i++)
            for (int j = -1; j <= 1; j++)
            {
                color = mybmp_1.GetPixel(x + i, y + j);
                r += color.rgbRed * temp[i + 1][j + 1];
                g += color.rgbGreen * temp[i + 1][j + 1];
                b += color.rgbBlue * temp[i + 1][j + 1];
            }

        color.rgbBlue = (int)b;
        color.rgbGreen = (int)g;
        color.rgbRed = (int)r;
        mybmp_output.WritePixel(x, y, color);
    }
}
```

4、比较利用以上三种空域平滑滤波器的实验结果，说明它们的有效性。

均值滤波在领域选择小的时候去噪效果不明显，选择的邻域大的时候会造成图像细节丢失，呈现模糊不清的现象。

中值滤波在抑制图像噪声方面十分有效，算法速度快，便于硬件实现。它对于去除椒盐噪声很有效，不过从图中可以看出去噪的效果与领域的尺度关系很大。

加权平均滤波在模糊程度上比相同邻域下模糊度小很多，但是牺牲为去噪效果稍差。其基本性质与均值滤波相同。

## W08

1、分别用梯度法锐化方法、边缘增强算子和高斯拉普拉斯方法对下列图像进行锐化处理，处理前后的图像分别显示出来。

(1) Airplane



(2) Man



(3) Nut

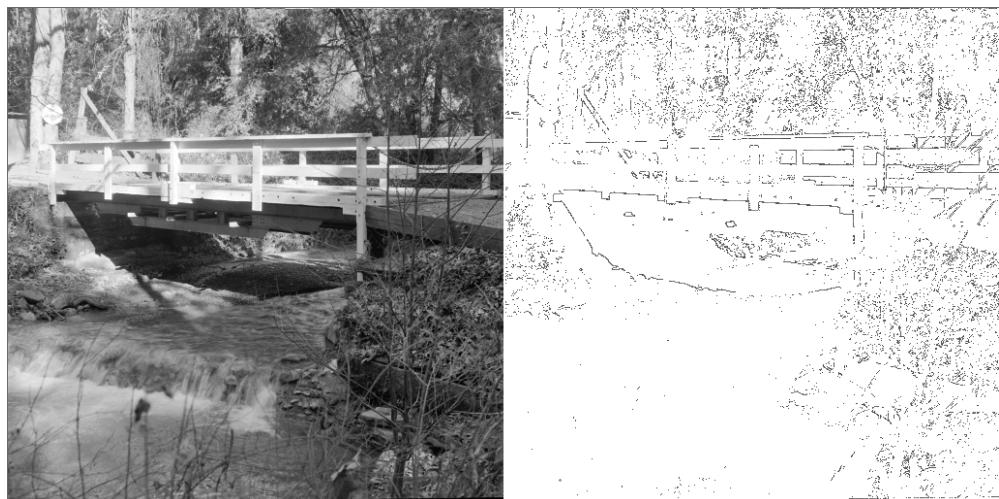


(4) Aerial photography



## 1. AirPlane

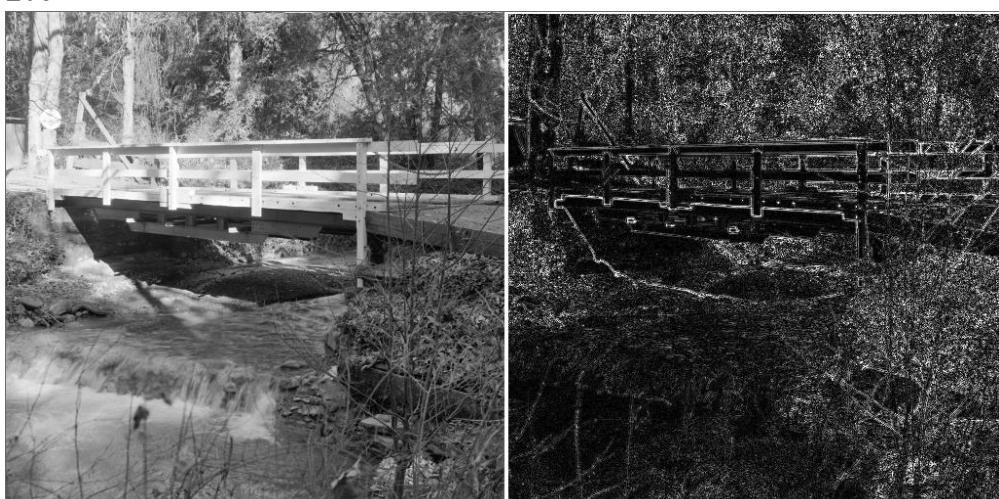
梯度



边缘增强算子



LOG



## 2. Man

梯度



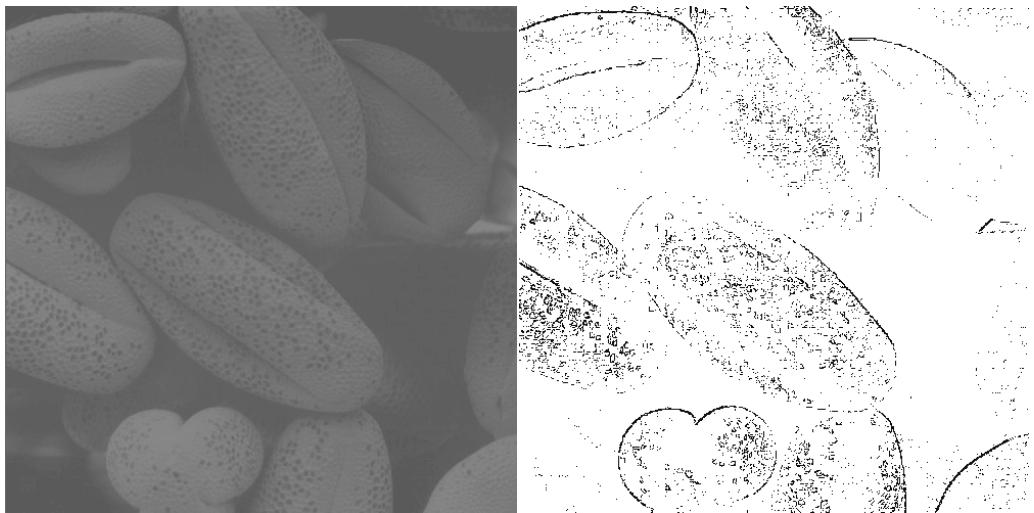
边缘增强算子



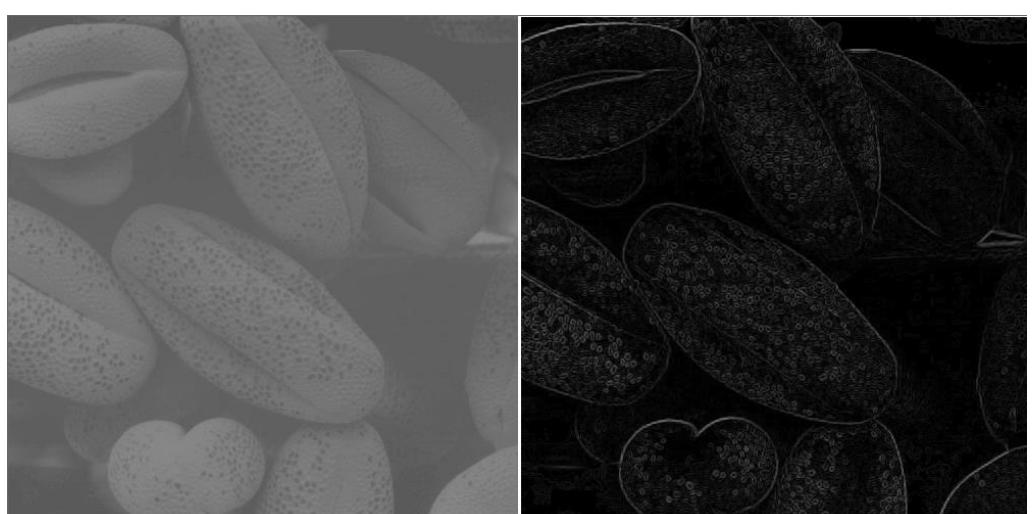
LOG



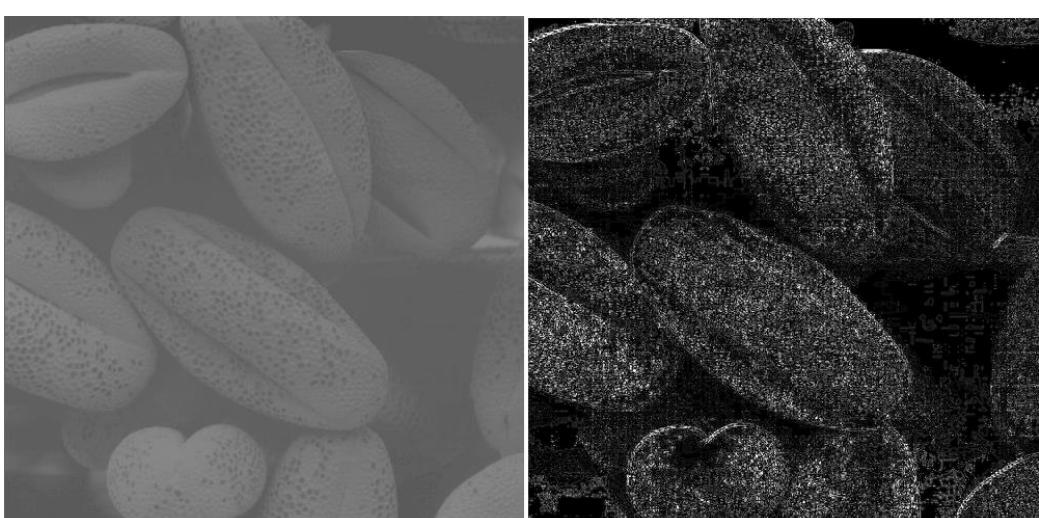
3. Nut  
梯度



边缘增强算子



LOG

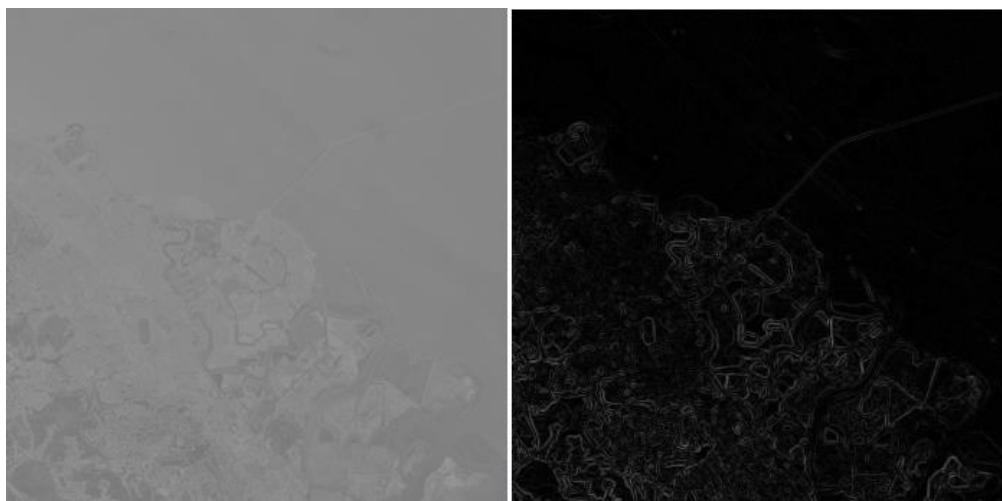


#### 4. Airial photography

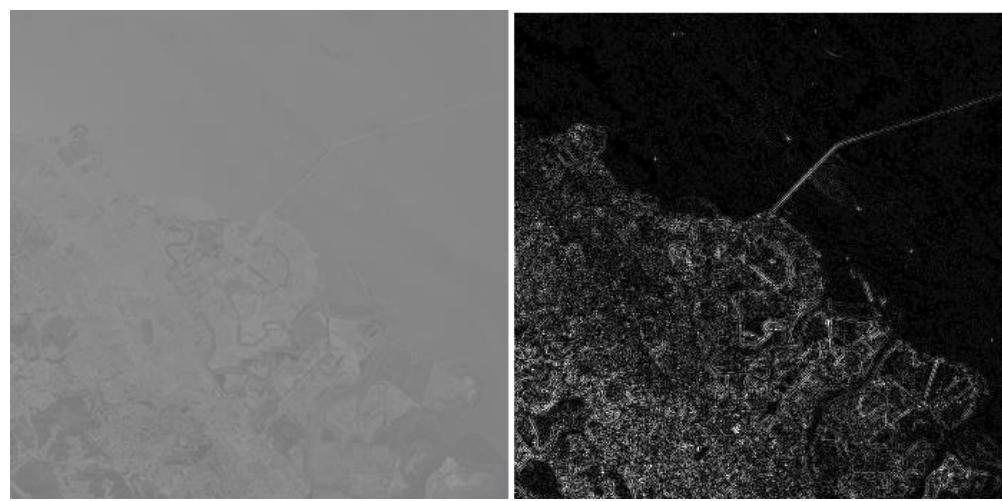
梯度



边缘增强算子



LOG



2、比较利用以上三种空域锐化方法得到的实验结果，说明它们的有效性。

梯度方法：因为梯度得到的对边缘和噪声的区别不敏感，所以保留了很多嘈杂的信息，而且无法找到精确的边缘点。可以从上述结果中看到不仅边缘被保留，还有很多小点被保留。

边缘增强算子：这里用的是 Laplace 算子，它是不依赖边缘方向的二阶微分算子，是一个标量。由于一阶导数的局部最大值对应着二阶导数的零交叉点，通过求图像的二阶导数的零交叉点可以找到精确的边缘点。图中因为没有噪声，所以无法看出特性。

LOG 算子是在 Laplace 算子之前先应用高斯滤波去除噪声，再用 Laplace 边缘增强。可以从图中看出 LOG 是边缘增强效果最佳的。

3、说明分别利用以上三种方法进行空域锐化时，参数选择应该注意哪些问题？

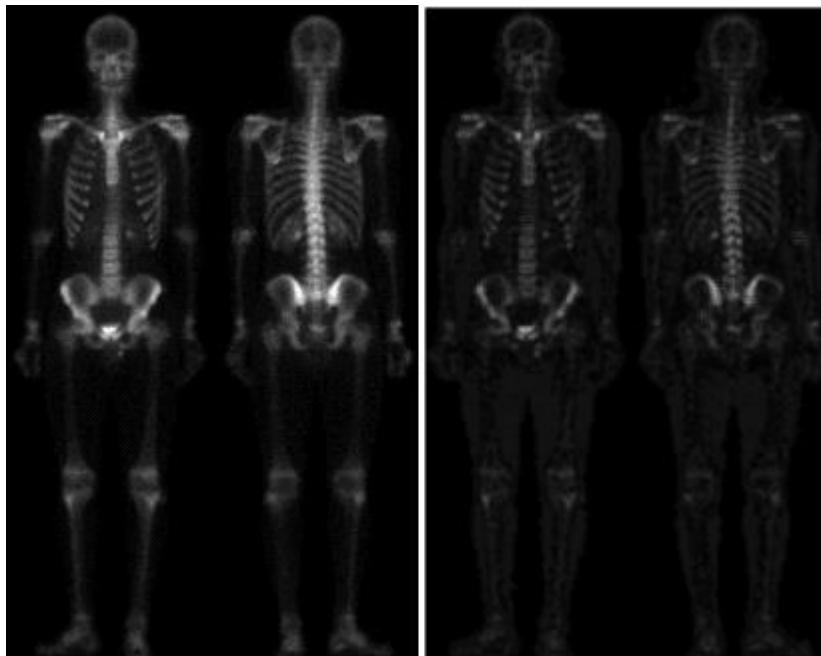
梯度方法对不同图的阈值选择差别很大。在对比度不大的情况下（比如题 1 的图 3 和 4）应该选择很小的阈值。但是在图像对比度很高的情况下需要选择比较大的阈值

边缘增强算子（Prewitt、Laplace、LOG 等）因为有的算子是依赖于边缘方向的，所以要对边缘方向处理。其次需要根据边缘的明显程度和干扰的大小来选择系数大小，可以先对结果 \* 1.0，然后根据结果，边缘不明显则增大系数，干扰太多则降低系数。

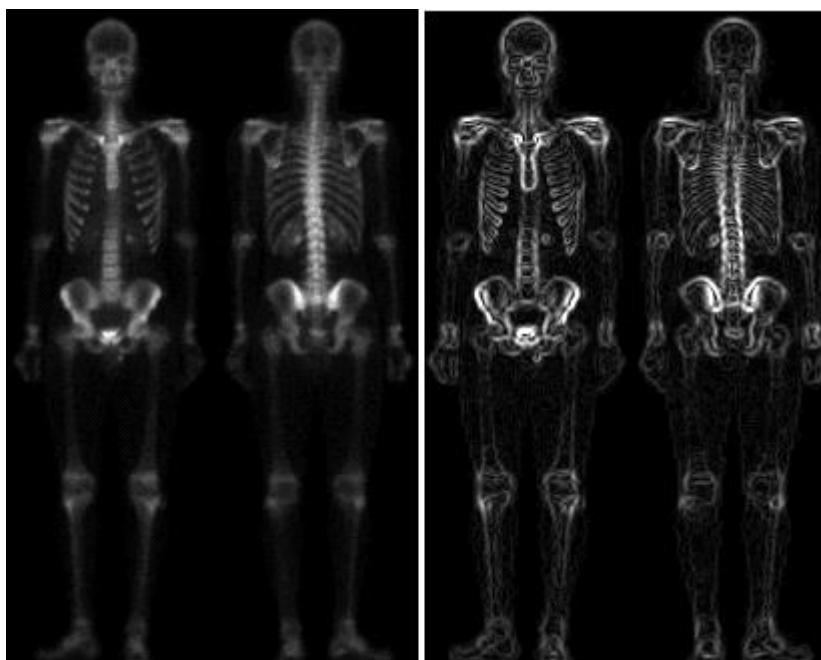
4、利用你所学过的方法，对以下图像进行增强处理。



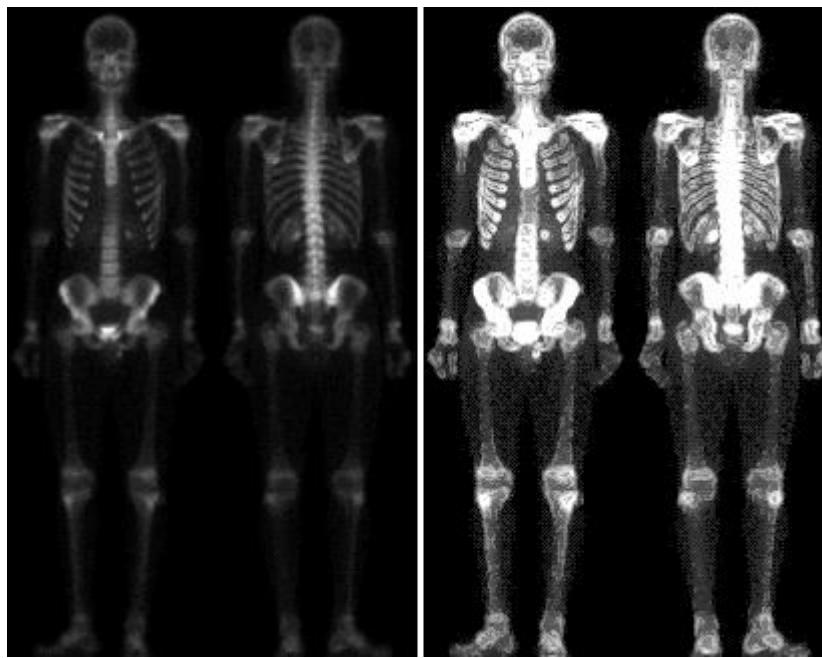
- 首先对原图像进行基于 Laplace 算子的边缘增强和模糊处理



- 其次进行基于 Sober 算子的边缘增强



- 最后原图像进行幂变提高亮度，然后与 Sober 提取出的经过阈值处理的边缘相加得到最终的结果：



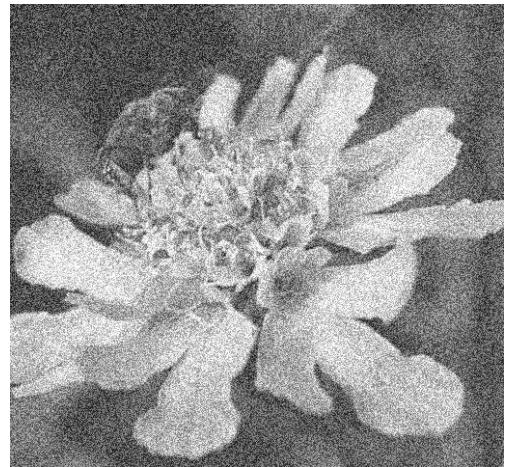
**W09**

1、利用理想低通滤波器，对下列图像进行平滑处理，并显示处理前后的图像。

(1) lenna



(2) Flower



(3) ColorPen

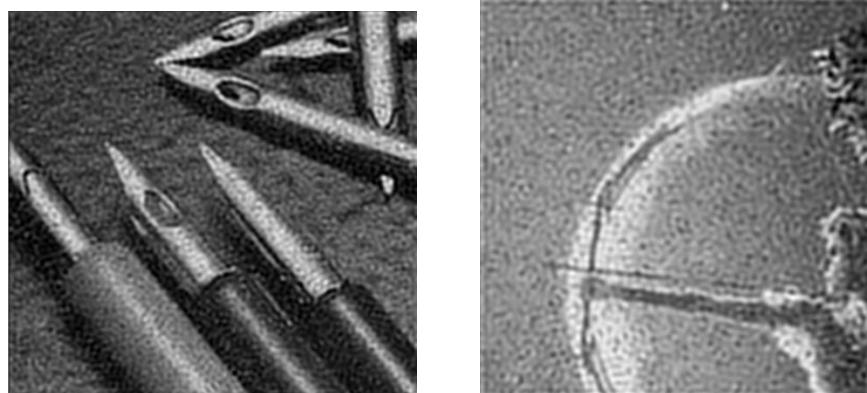


(4) ColorBoy



Ans:



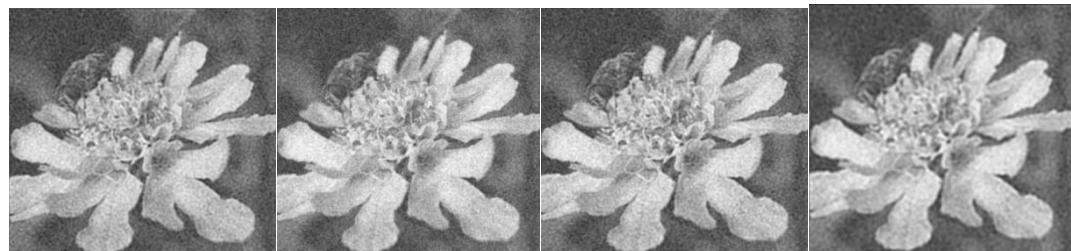


2、利用巴特沃思低通滤波器、高斯低通滤波器、指数形低通滤波器、梯形低通滤波器分别对第1题中的图像进行平滑处理，并比较各种频域低通滤波器的作用的差异。(按题目顺序给出各个滤波器的效果)

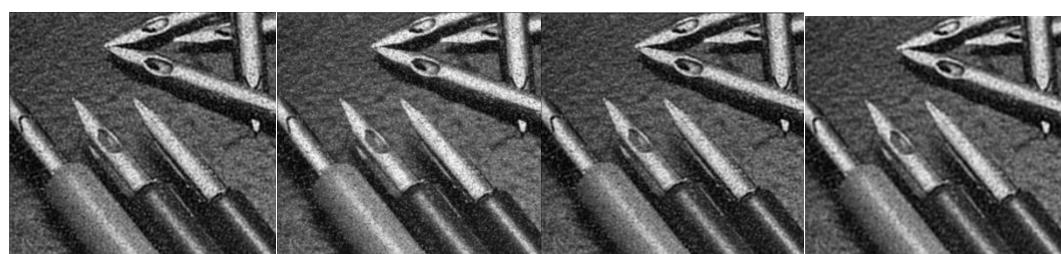
第一张图： $d=50$ ，阶数 5



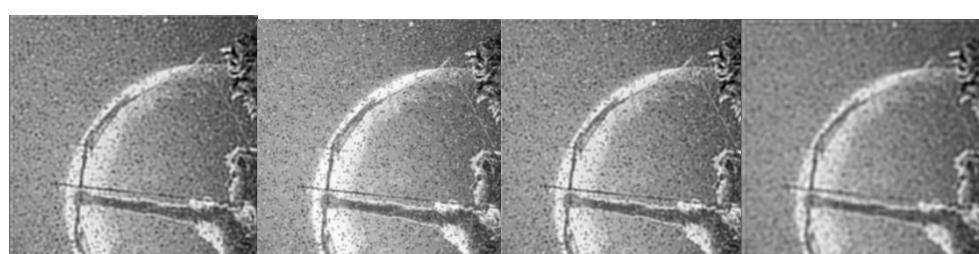
第二张图： $d=70$ ，阶数 5



第三张图： $d=50$ ，阶数 5



第四张图： $d=30$ ，阶数 5



各种频域低通滤波器的作用的差异：

巴特沃斯滤波器可以调节阶数，来实现接近理想滤波器的效果。

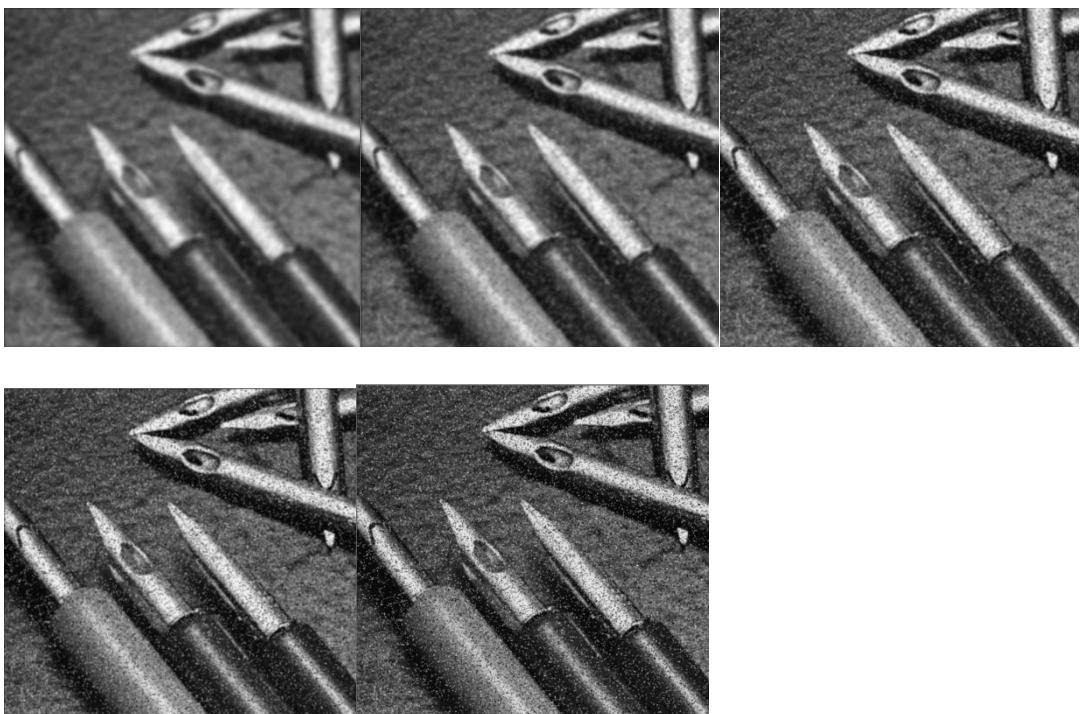
高斯滤波器可以过滤一些自然界的随机噪声，效果很好。

指数滤波器可以根据阶数来保留完全的低频滤波。

梯形滤波器则是可以按照自己的需求来保留低频滤波。

3、比较并说明截止半径对平滑结果的影响。说明使用截止半径时应该注意哪些问题。

举例：高斯滤波器对第三幅图片平滑处理  $d=20/30/50/70/90$



可以看到  $d$  越小图像特征保留就越少，图像越模糊，例如  $d=20$  已经特征缺失严重了。而  $d$  越大图像特征保留完全，可是图像平滑处理确完全没有效果了。例如  $d=90$  时噪声完全保留。

使用截止半径的大小是因噪声情况而异的，应多次尝试，不断缩小可以用的截止频率的范围(例如使用二分法)来达到最好的效果。

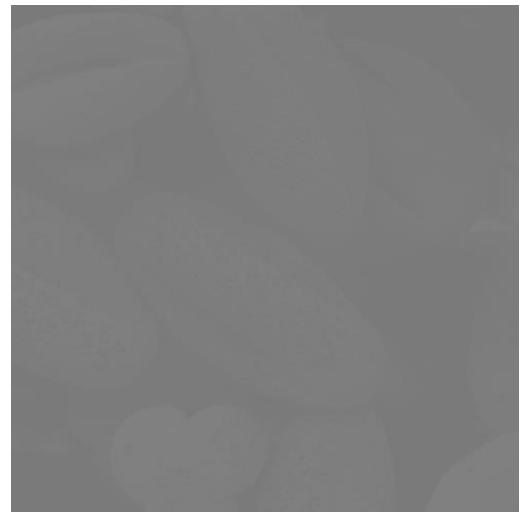
### W10

1、利用理想高通滤波器，对下列图像进行锐化处理，并显示处理前后的图像。

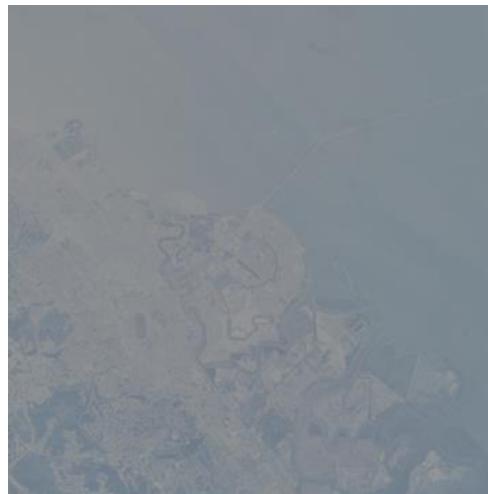
(1) Man



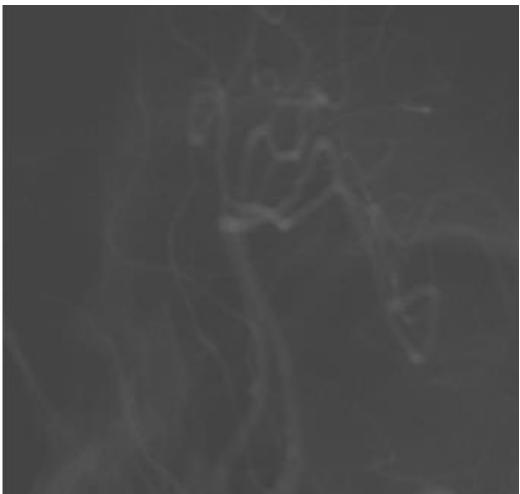
(2) Nut



(3) Aerial photography

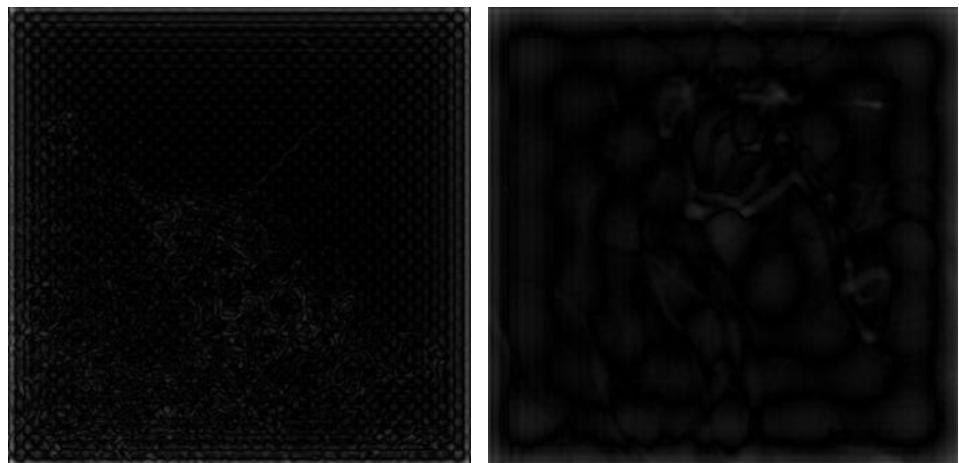


(4) MedicalImage



Ans: d0 分别为 10, 5, 10, 4





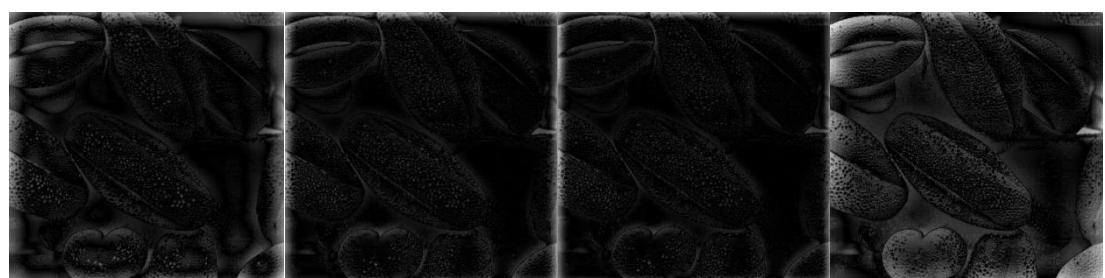
理想高通过滤器的效果非常差，因为低频成分全部被过滤掉了，所以图像成分保留很少很少，图像几乎看不清

2、利用巴特沃思高通滤波器、高斯高通滤波器、指数形高通滤波器、梯形高通滤波器分别对第 1 题中的图像进行锐化处理，并比较各种频域高通滤波器的作用的差异。

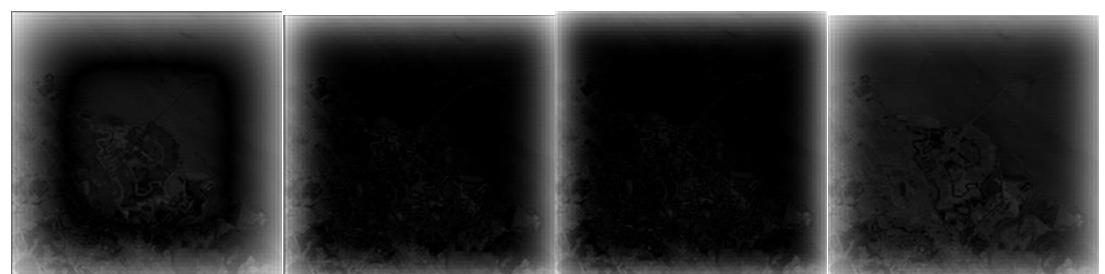
图片 1:  $d=5$ , 阶数取 2



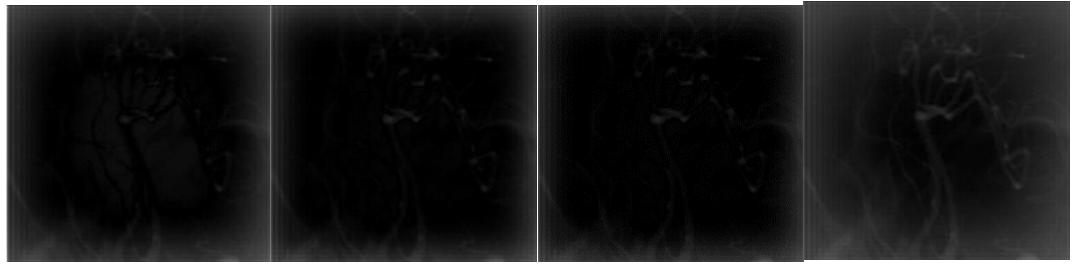
图片 2:  $d=5$ , 阶数取 2



图片 3:  $d=2$ , 阶数取 2



图片 4:  $d=2$ , 阶数取 2

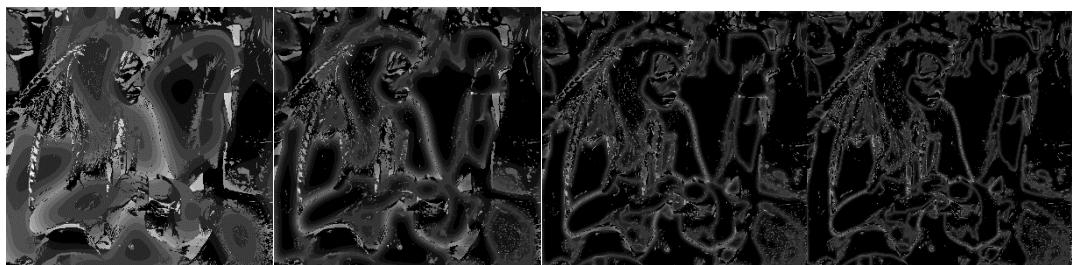


Ans:巴特沃思高通滤波器、高斯高通滤波器、指数形高通滤波器中效果都不是太好，因为对低频成分都保留得比较少，所以会显得很黑。但是这里梯形高通滤波器发挥了很好的效果，将低频成分线性地保留下来，将高频成分完全保留，信息丢失少，同时锐化效果好。

然而事实上，高通滤波器对高低频分布的图片具有不用的效果。例如第一张图和第三张图就是一个很好的比较。对于第一张高频也分布了很多成分的图片，截至半径可以选择一个比较大的值，同时也保留下较多的照片中的成分。而对于成分全部堆积在低频中的图片，选择截止频率变成为一个很困难的事情。因为1的变化将导致成分完全的缺失（即图片过黑）或者成分完全保留（即没有锐化效果）。

3、比较并说明截止半径对锐化结果的影响。说明使用截止半径时应该注意哪些问题。

对图片 1，基于高斯高通滤波器，分别实验  $d=2, 5, 10, 15$  的效果



在选择截止半径时，如果锐化半径太小，高通性能不好，几乎所有频率的成分都能通过，则锐化效果欠佳。但是如果锐化半径取得太大，几乎所有频率的成分都不能通过，则信号都被阻止。

选择截止半径时，首先要注意集中低频区域的半径是多大，如果太模糊则适当扩大截止半径，否则限制截止半径。可以使用二分的形式来逐渐缩小得到适合的截止半径。

## W12

1、利用不同算子（Roberts 算子、Sobel 算子、Prewitt 算子、Laplace 算子、Canny 算子）对下列图像进行分割处理，比较不同方法的分割效果。如果是彩色图像，先灰度化处理。

(1) House



(2) Pen



(3) Lenna



(4) Pepper



(5) Flower1

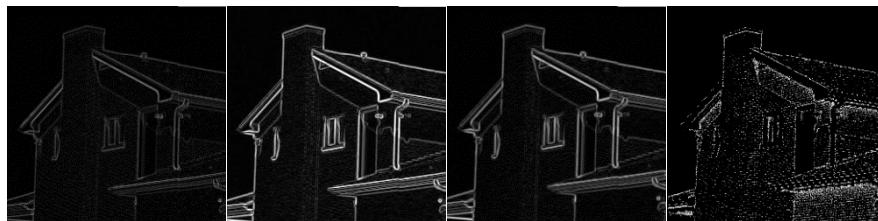


(6) Flower2

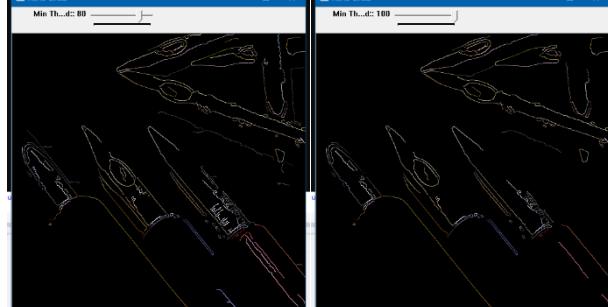
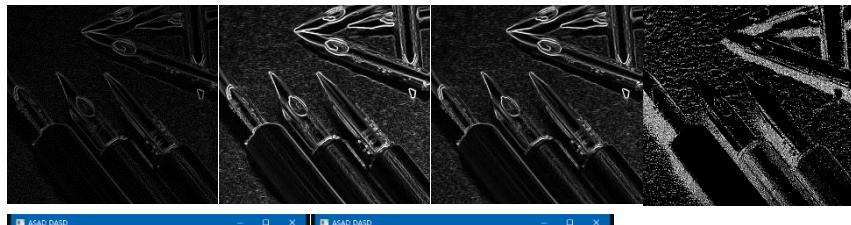


Ans: 顺序分别为 Roberts 算子、Sobel 算子、Prewitt 算子、Laplace 算子、Canny 算子

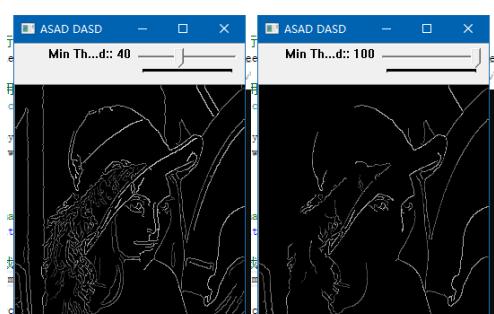
(1) fvalue=0.5f Lap-f=5.5f



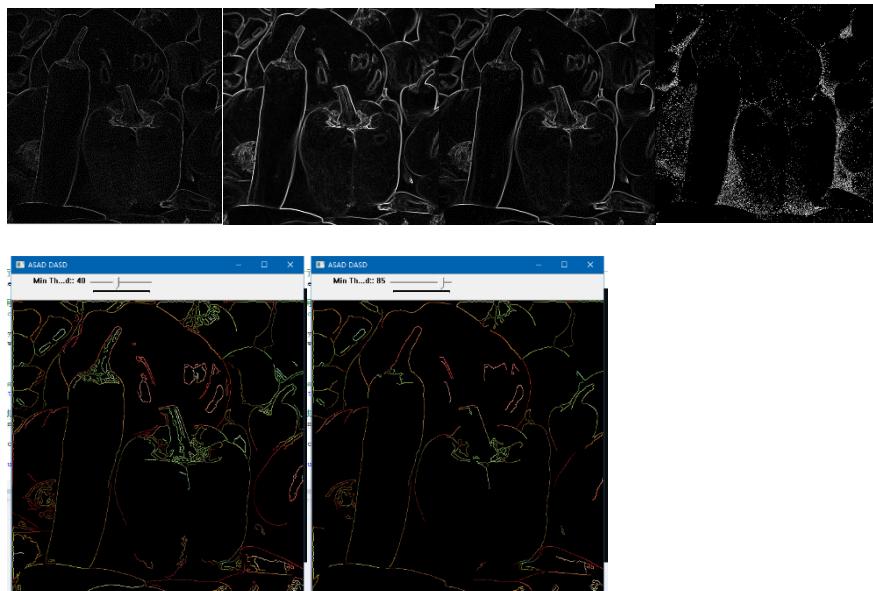
(2) fvalue=0.7f Lap-f=6.0f



(3) fvalue=0.5f Lap-f=6.0f



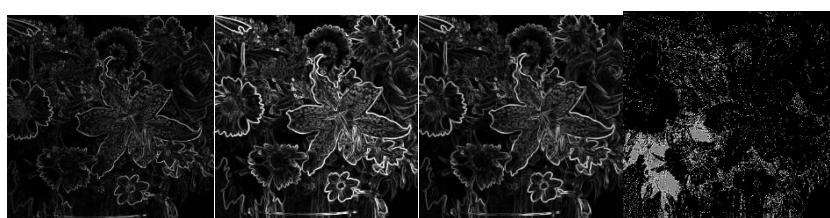
(4) fvalue=0.5f Lap-f=6.0f

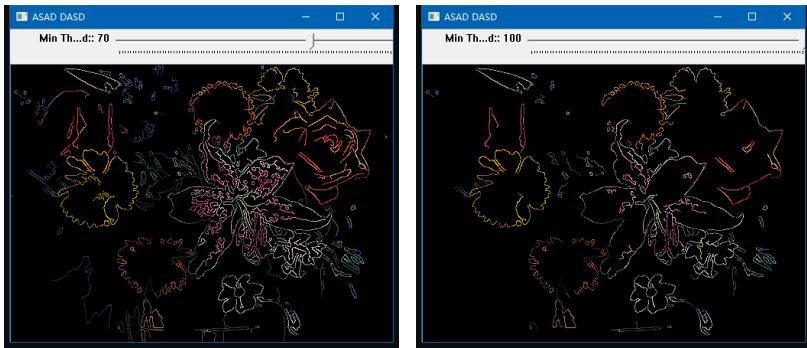


(5) fvalue=0.5f Lap-f=10.0f



(6) fvalue=0.5f Lap-f=10.0f





Roberts 算子仅是简单模拟一阶的梯度检测边缘，效果不明确。  
Sobel 算子可以采用不同领域对边缘增强的结果影响很大。同时也具有方向性。从实验中可以看出它效果明显，但是又对噪音明显，所以我认为只在低噪的时候适用。

Prewitt 算子：同样具有方向性和对噪音敏感的效果，但是都相对效果比 sobel 弱，各有取舍。

Laplace 算子：并非 LOG 算子，没有进行高斯模糊所以对噪声极其敏感。可以从图中看出每一此实验都是充斥着噪声点，被误认为是边缘。有噪声情况下效果很差。

上述几个算子是基础算子，较为简单，但是其中 Canny 算子的实现最为复杂，因为它的具体步骤为：

#### 一、用高斯滤波器平滑图像

高斯滤波是一种线性平滑滤波，适用于消除高斯噪声，特别是对抑制或消除服从正态分布的噪声非常有效。

#### 二、用 Sobel 等梯度算子计算梯度幅值和方向

Sobel 梯度算子即具有水平和竖直方向性的边缘检测算子，它可以计算出 x、y 方向梯度和梯度角，在下面过程中

#### 三、对梯度幅值进行非极大值抑制

求幅值图像进行非极大值抑制，可以进一步消除非边缘的噪点，更重要的是，可以细化边缘。抑制逻辑是：沿着该点梯度方向，比较前后两个点的幅值大小，若该点大于前后两点，则保留，若该点小于前后两点，则置为 0；

#### 四、用双阈值算法检测和连接边缘

指定一个低阈值 A，一个高阈值 B，一般取 B 为图像整体灰度级分布的 70%，且 B 为 1.5 到 2 倍大小的 A；灰度值大于 B 的，置为 255，灰度值小于 A 的，置为 0；灰度值介于 A 和 B 之间的，考察改像素点临近的 8 像素是否有灰度值为 255 的，若没有 255 的，表示这是一个孤立的局部极大值点，予以排除，置为 0；若有 255 的，表示这是一个跟其他边缘有“接壤”的可造之材，置为 255，之后重复执行该步骤，直到考察完之后一个像素点。

这里使用了 OpenCv 来实现 Canny 算子，可以看到效果非常的好，不受噪音限制同时分割线条也很细。同时各个方向的线条都可以分辨。

代码如下：

```
int main(int argc, char** argv)
{
    /// 装载图像
    src = imread("6.png");

    if (!src.data)
    {
        return -1;
    }

    /// 创建与src同类型和大小的矩阵(dst)
    dst.create(src.size(), src.type());

    /// 原图像转换为灰度图像
    cvtColor(src, src_gray, CV_BGR2GRAY);

    /// 创建显示窗口
    namedWindow(window_name, CV_WINDOW_AUTOSIZE);

    /// 创建trackbar
    createTrackbar("Min Threshold:", window_name, &lowThreshold, max_lowThreshold, CannyThreshold);

    /// 显示图像
    CannyThreshold(0, 0);

    /// 等待用户反应
    waitKey(0);

    return 0;
}

void CannyThreshold(int, void*)
{
    /// 使用 3x3内核降噪
    blur(src_gray, detected_edges, Size(3, 3));

    /// 运行Canny算子
    Canny(detected_edges, detected_edges, lowThreshold, lowThreshold*ratio, kernel_size);

    /// 使用 Canny算子输出边缘作为掩码显示原图像
    dst = Scalar::all(0);

    src.copyTo(dst, detected_edges);
    imshow(window_name, dst);
}
```

2、利用全局阈值迭代分割方法对题目 1 的各个图像分别处理，并显示分割的结果。并说明全局阈值迭代分割方法的效果如何。





可以从实验中看出，在图像直方图几乎为两个峰值的时候，均值迭代的方法得到的效果非常好。但是其他时候前景后景分的不是太明确，因为它只会产生一个阈值而图片中有多个峰值。

3、利用最大类间方差法（OTSU 方法）对题目 1 的各个图像分别处理，并显示分割的结果。并说明 OTSU 方法方法的处理效果如何。





可以从实验中看出，类似与全局阈值迭代分割方法，在图像直方图几乎为两个峰值的时候，均值迭代的方法得到的效果非常好。但是其他时候前景后景分的不是太明确，因为它只会产生一个阈值而图片中有多个峰值。

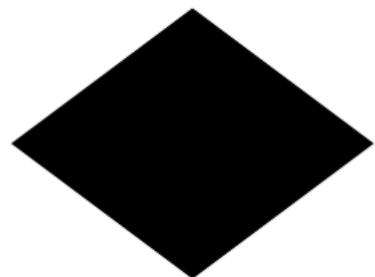
#### 4、比较全局阈值迭代分割方法与 OTSU 方法方法的分割结果，并加以说明。

OTSU 比起全局阈值方法有一个缺点，就是他的对于复杂峰值的图片处理效果非常差。例如对图 4 和图 6 的处理，前景不是变得太多就是变得太少。但是同时他也有一定优势，例如蕾哈娜的图像，OTSU 对噪音敏感度小，所以不会把很多噪音、小点当成边缘。

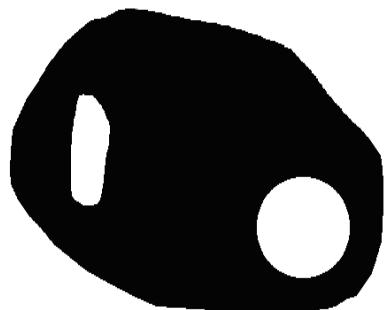
### W13

1、分别利用区域生长法算法和主动轮廓模型(蛇模型)分别对下列图像进行分割处理，如果是彩色图像，先灰度化处理，并显示分割结果。

(1) Retangle



(2) Object1



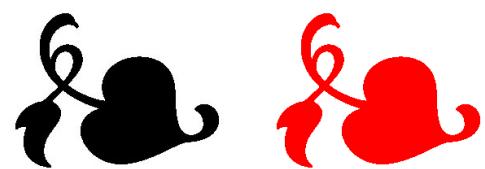
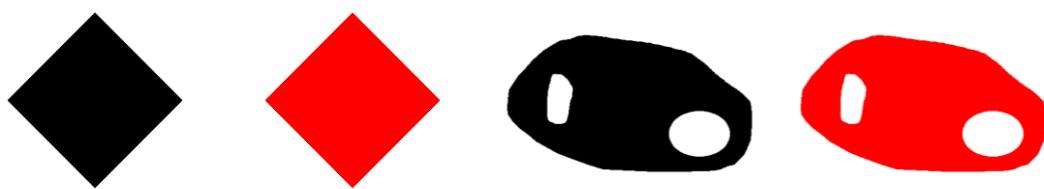
(3) Object2



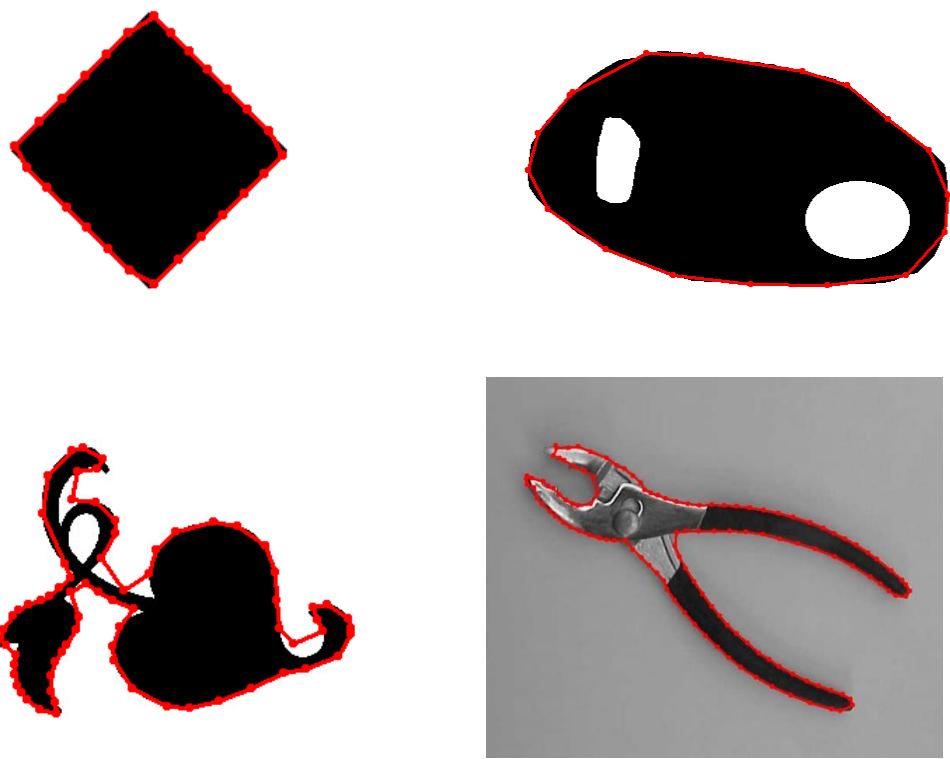
(4) Tool



区域生长算法结果：



蛇模型结果

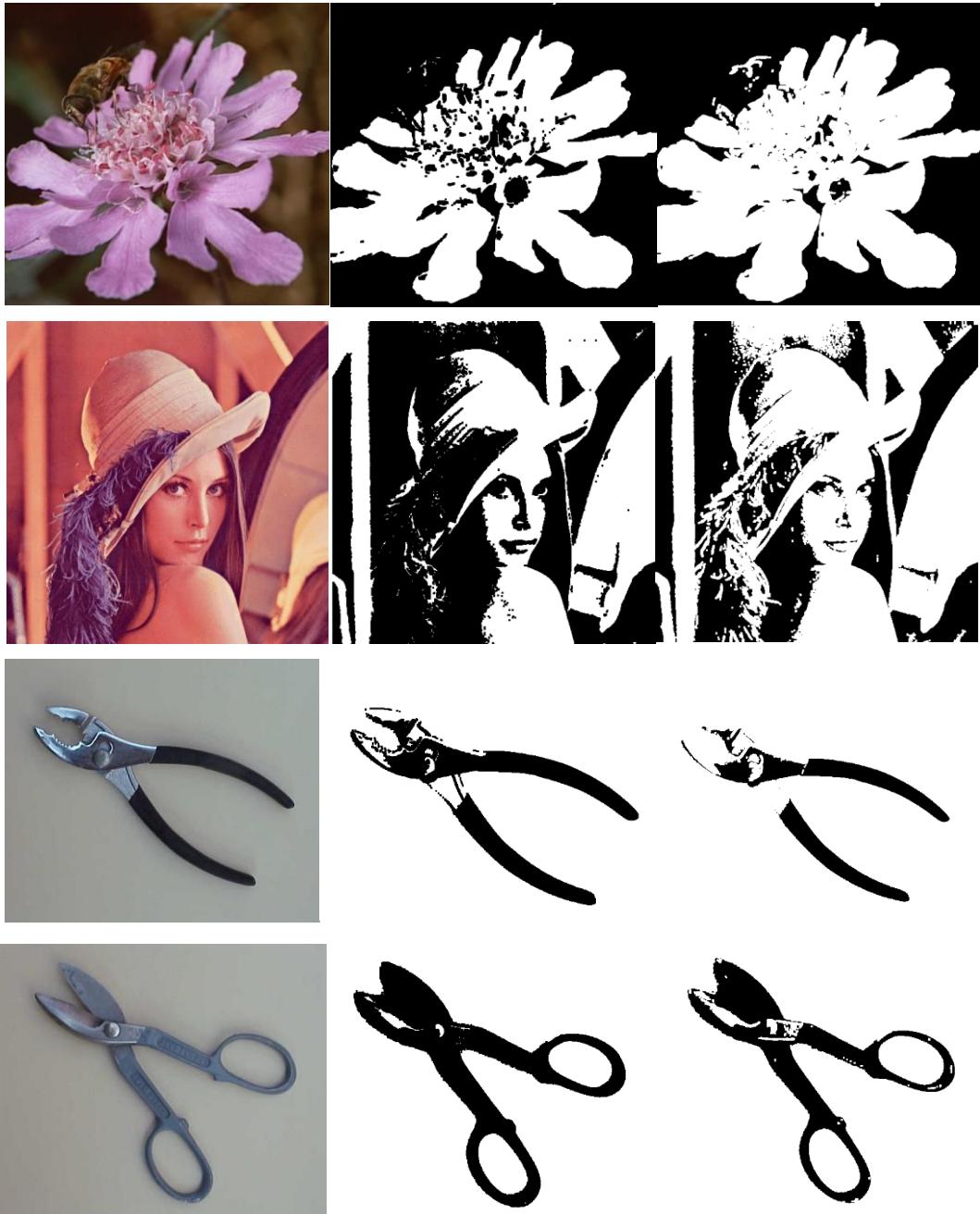


2、从上述题目 1 的处理结果，说明利用主动轮廓模型分割的有效性。

对于前三幅二值图像，肯定是生长模型的效果更好，因为所有前景都会和种子点是同一个灰度值，但是对于第四幅真实照片来说，区域生长算法比起蛇模型效果差了很多。可以看到蛇模型大致包裹了整个老虎钳，而生长算法错将下壁的金属部分当成背景了。主动轮廓算法是根据轮廓点的外能量(梯度)、内能力(弹性势能和弯曲势能之和)得到点的受力情况，根据点的受力情况来对曲线进行形变。所以在迭代的过程中可以得到比生长算法更好的效果。

**W14**

1、对下列图像分别进行膨胀和腐蚀的操作, 显示处理的结果, 并对比说明膨胀和腐蚀对图像处理的作用和效果.



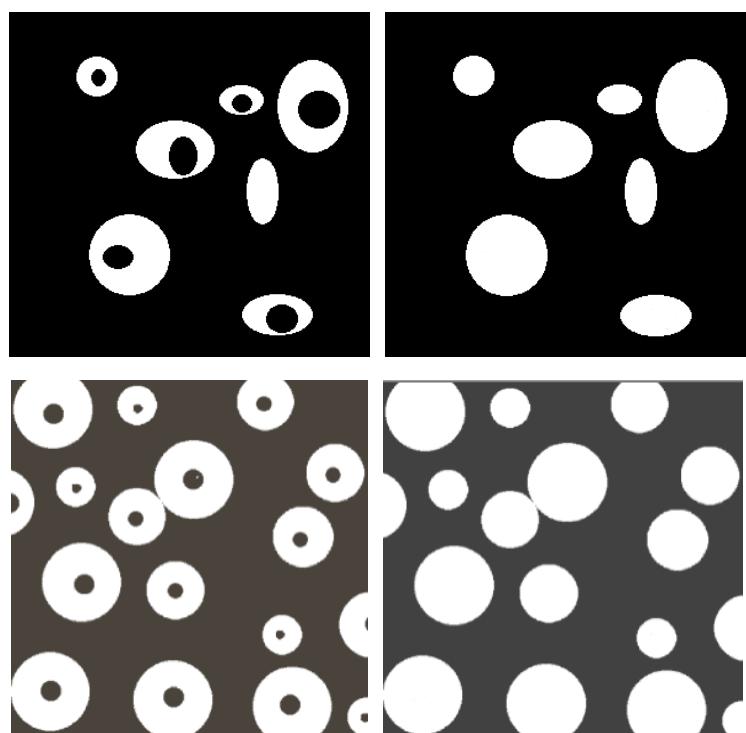
2、对下列图像分别进行开操作和闭操作,并显示处理的结果,对比说明开操作和闭操作对图像处理的作用和效果.



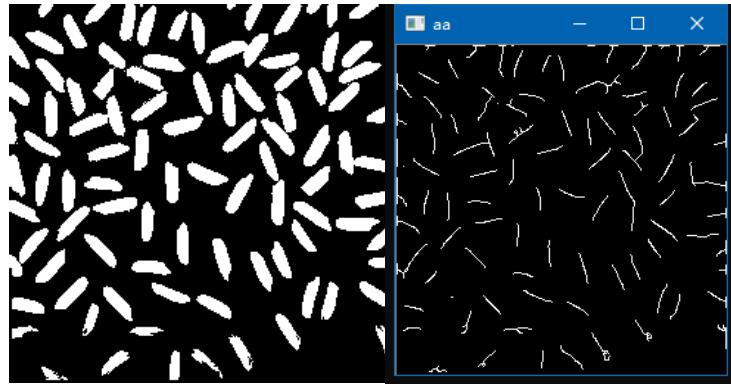
3、对下列图像利用数学形态学方法分别检测出内边缘和外边缘. 并显示检测的结果.



4、利用数学形态学方法对下列图像进行孔填充处理, 并显示检测的结果.



5、利用数学形态学方法,求取下列图像的骨架,并显示骨架的结果.



骨架生成使用了 opencv 的方法, 相关代码如下:

```
void thinimage(Mat &srcimage)//单通道、二值化后的图像
{
    vector<Point> deletelist1;
    int Structure[9];
    int nl = srcimage.rows;
    int nc = srcimage.cols;
    while (true)
    {
        for (int j = 1; j < (nl - 1); j++)
        {
            uchar* data_last = srcimage.ptr<uchar>(j - 1);
            uchar* data = srcimage.ptr<uchar>(j);
            uchar* data_next = srcimage.ptr<uchar>(j + 1);
            for (int i = 1; i < (nc - 1); i++)
            {
                if (data[i] == 255)
                {
                    Structure[0] = 1;
                    if (data_last[i] == 255) Structure[1] = 1;
                    else Structure[1] = 0;
                    if (data[i + 1] == 255) Structure[2] = 1;
                    else Structure[2] = 0;
                    if (data[i + 1] == 255) Structure[3] = 1;
                    else Structure[3] = 0;
                    if (data_next[i + 1] == 255) Structure[4] = 1;
                    else Structure[4] = 0;
                    if (data_next[i] == 255) Structure[5] = 1;
                    else Structure[5] = 0;
                    if (data_next[i - 1] == 255) Structure[6] = 1;
                    else Structure[6] = 0;
                    if (data[i - 1] == 255) Structure[7] = 1;
                    else Structure[7] = 0;
                    if (data_last[i - 1] == 255) Structure[8] = 1;
                    else Structure[8] = 0;
                    int whitepointtotal = 0;
                    for (int k = 1; k < 9; k++)
                    {
                        whitepointtotal = whitepointtotal + Structure[k];
                    }
                    if ((whitepointtotal >= 2) && (whitepointtotal <= 6))
                    {
                        int ap = 0;
                        if ((Structure[1] == 0) && (Structure[2] == 1)) ap++;
                        if ((Structure[2] == 0) && (Structure[3] == 1)) ap++;
                        if ((Structure[3] == 0) && (Structure[4] == 1)) ap++;
                        if ((Structure[4] == 0) && (Structure[5] == 1)) ap++;
                        if ((Structure[5] == 0) && (Structure[6] == 1)) ap++;
                        if ((Structure[6] == 0) && (Structure[7] == 1)) ap++;
                        if ((Structure[7] == 0) && (Structure[8] == 1)) ap++;
                        if ((Structure[8] == 0) && (Structure[1] == 1)) ap++;
                        if (ap == 1)
                        {
                            deletelist1.push_back(Point(i, j));
                        }
                    }
                }
            }
        }
    }
}
```

```

        {
            if ((Structure[1] * Structure[7] * Structure[5] == 0) && (Structure[3] * Structure[5] * Structure[7] == 0))
            {
                deletelist1.push_back(Point(i, j));
            }
        }
    }
}

if (deletelist1.size() == 0) break;
for (size_t i = 0; i < deletelist1.size(); i++)
{
    Point tem;
    tem = deletelist1[i];
    uchar* data = srcimage.ptr<uchar>(tem.y);
    data[tem.x] = 0;
}
deletelist1.clear();

for (int j = 1; j < (nl - 1); j++)
{
    uchar* data_last = srcimage.ptr<uchar>(j - 1);
    uchar* data = srcimage.ptr<uchar>(j);
    uchar* data_next = srcimage.ptr<uchar>(j + 1);
    for (int i = 1; i < (nc - 1); i++)
    {
        if (data[i] == 255)
        {
            Structure[0] = 1;
            if (data_last[i] == 255) Structure[1] = 1;
            else Structure[1] = 0;
            if (data_last[i + 1] == 255) Structure[2] = 1;
            else Structure[2] = 0;
            if (data[i + 1] == 255) Structure[3] = 1;
            else Structure[3] = 0;
            if (data_next[i + 1] == 255) Structure[4] = 1;
            else Structure[4] = 0;
            if (data_next[i] == 255) Structure[5] = 1;
            else Structure[5] = 0;
            if (data_next[i - 1] == 255) Structure[6] = 1;
            else Structure[6] = 0;
            if (data[i - 1] == 255) Structure[7] = 1;
            else Structure[7] = 0;
            if (data_last[i - 1] == 255) Structure[8] = 1;
            else Structure[8] = 0;
            int whitepointtotal = 0;
            for (int k = 1; k < 9; k++)
            {
                whitepointtotal = whitepointtotal + Structure[k];
            }
            if ((whitepointtotal >= 2) && (whitepointtotal <= 6))
            {
                if ((whitepointtotal >= 2) && (whitepointtotal <= 6))
                {
                    int ap = 0;
                    if ((Structure[1] == 0) && (Structure[2] == 1)) ap++;
                    if ((Structure[2] == 0) && (Structure[3] == 1)) ap++;
                    if ((Structure[3] == 0) && (Structure[4] == 1)) ap++;
                    if ((Structure[4] == 0) && (Structure[5] == 1)) ap++;
                    if ((Structure[5] == 0) && (Structure[6] == 1)) ap++;
                    if ((Structure[6] == 0) && (Structure[7] == 1)) ap++;
                    if ((Structure[7] == 0) && (Structure[8] == 1)) ap++;
                    if ((Structure[8] == 0) && (Structure[1] == 1)) ap++;
                    if (ap == 1)
                    {
                        if ((Structure[1] * Structure[3] * Structure[5] == 0) && (Structure[3] * Structure[1] * Structure[7] == 0))
                        {
                            deletelist1.push_back(Point(i, j));
                        }
                    }
                }
            }
        }
    }
}

if (deletelist1.size() == 0) break;
for (size_t i = 0; i < deletelist1.size(); i++)
{
    Point tem;
    tem = deletelist1[i];
    uchar* data = srcimage.ptr<uchar>(tem.y);
    data[tem.x] = 0;
}
deletelist1.clear();
}

```

### W15

1. 对下列图像进行亮度取反操作，并显示处理结果。

(1) Playboy



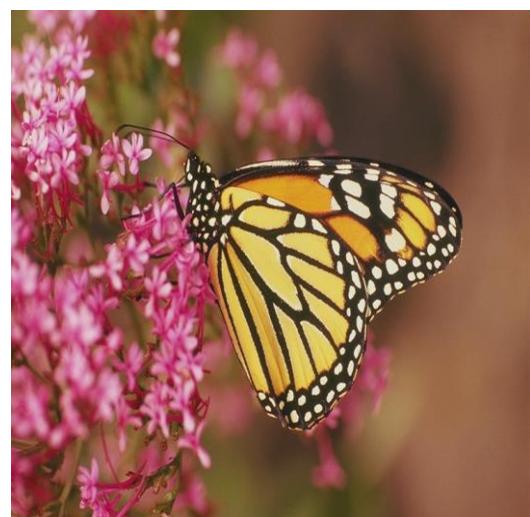
(2) Flower



(3) Flower2



(4) monarch

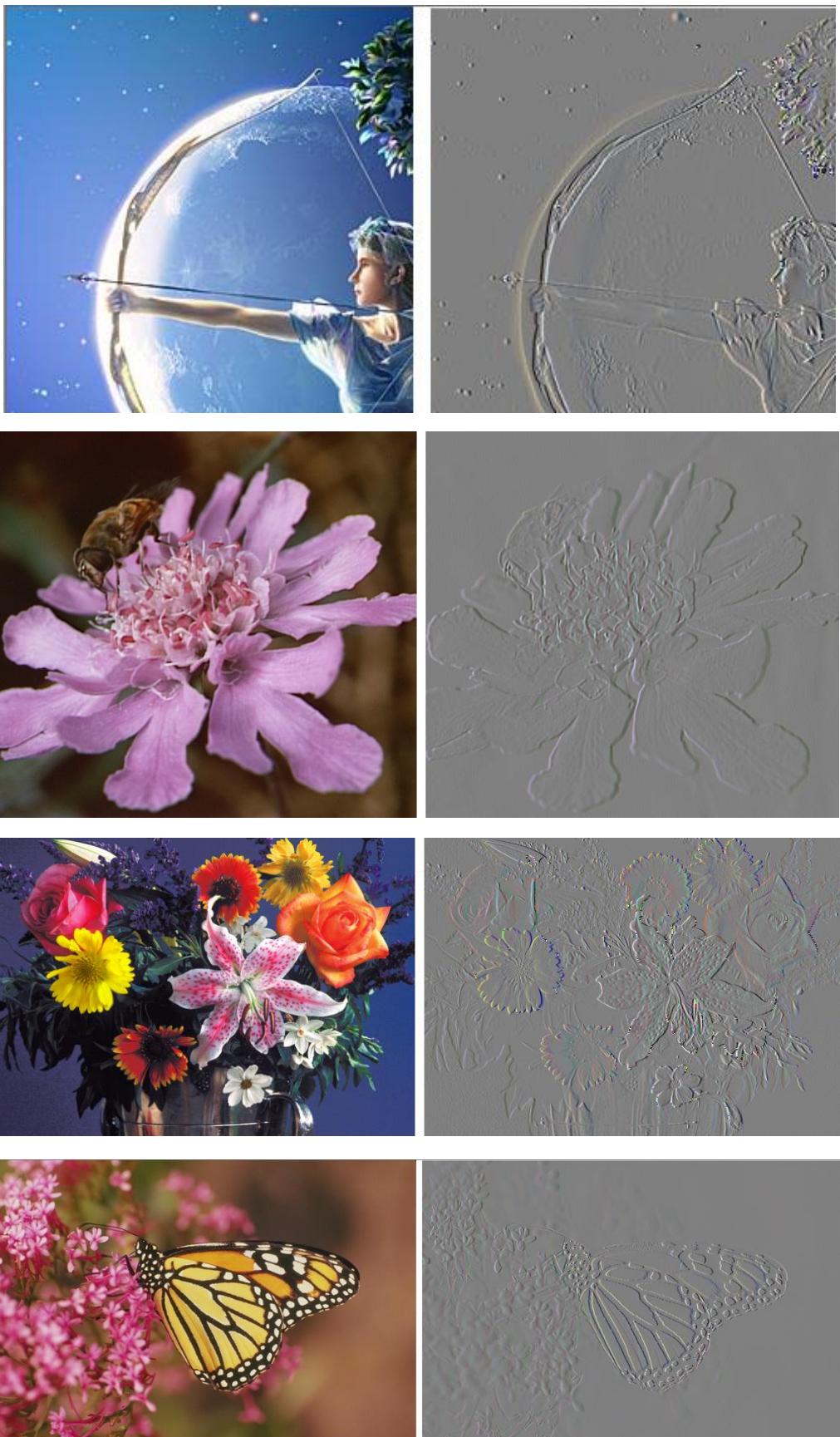


Ans:





2、对上述题目 1 的图像进行浮雕操作，并显示图像浮雕处理结果。



3、对上述题目 1 的图像进行马赛克处理，并显示图像处理结果。



4、在第 3 题的马赛克处理中，分别设置不同块的规模，根据马赛克处理结果说明块的规模对处理结果的影响。

模块大小顺序为：

$3 \times 3$

$5 \times 5$

$7 \times 7$





马赛克的原理是将图像从形式上划分为很多小块，在每块内的各个像素都取到相同的红、绿、蓝颜色值，从而对某些细节进行模糊化处理，使图像粗糙化，即采取  $n \times n$  的模板进行区域内分别对 R、G、B 通道进行均值化处理。而  $n$  的大小就决定了马赛克模糊块的大小。实验分别设置  $n$  为 3、5、7，可以从实验结果中看到， $n$  越大块越大，模糊效果越好。

对应代码如下，step 的大小为模块的  $n$  的大小：

```

void CW15View::OnStartMosaic()
{
    // TODO: 在此添加命令处理程序代码
    newbmp.CreateCBitmap(sizeimage, 24);
    int step = 3;
    for (int x = step; x < sizeimage.cx - step; x += (step*2+1))
    {
        for (int y = step; y < sizeimage.cy - step; y += (step * 2 + 1))
        {
            RGBQUAD color;
            color = mybmp.GetPixel(x, y);

            int r = 0, g = 0, b = 0, num = 0;
            for (int m1 = -step; m1 <= step; m1++)
                for (int m2 = -step; m2 <= step; m2++)
                {
                    if (x + m1 >= sizeimage.cx || x + m1 < 0 || y + m2 >= sizeimage.cy || y + m2 < 0)
                        continue;
                    num++;
                    RGBQUAD color1;
                    color1 = mybmp.GetPixel(x + m1, y + m2);
                    r += color1.rgbRed;
                    g += color1.rgbGreen;
                    b += color1.rgbBlue;
                }

            color.rgbRed = (unsigned char)(r*1.0 / num);
            color.rgbGreen = (unsigned char)(g*1.0 / num);
            color.rgbBlue = (unsigned char)(b*1.0 / num);

            for (int m1 = -step; m1 <= step; m1++)
                for (int m2 = -step; m2 <= step; m2++)
                {
                    newbmp.WritePixel(x + m1, y + m2, color);
                }
        }
    }
    Invalidate();
}

```