生物医学数字信号处理

作业1 二维卷积、高斯噪声与非负逆卷积

罗正潮U201813614

使用语言：C++

时间：2021.3.14

---

1) 生成论文中样例图片

依赖：

`using namespace cv;`

思路：

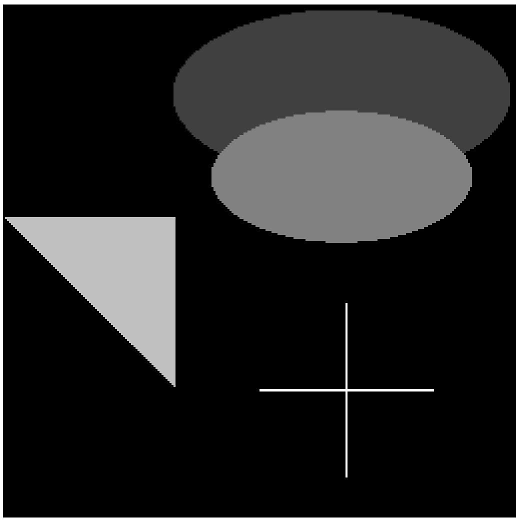
在windows 10的visual \ studio 2019上新建项目并设置项目属性，配置好opencv3相关连接器、库等路径。

首先利用Adobe阅读器保存论文中图片截图，再使用代码读入截图、存储和保存图片。

代码：

```
    //step 1画出图片
    Mat GT = imread("screenshot.bmp", 0);
    if (!GT.data)return -1;
    GT.convertTo(GT, CV_8UC1);
    imshow("GT", GT);
    imwrite("GT.bmp", GT);
```

结果：



2）对1）产生的图进行卷积运算（卷积函数是二位的高斯函数）

依赖:

`cv::Mat`、`cmath`、`cv::PI`、`assert.h`

思路:

根据上课全老师所讲授知识，按照两二维离散序列卷积获得结果的公式编写二维卷积函数。

二维卷积函数需要传入x和h两个Mat矩阵，根据二维高斯函数公式编写生成函数。

二维高斯函数的公式如下:

$$G_\sigma = \frac{1}{\sqrt{(2\pi\sigma^2)}} e^{(\frac{-(x^2+y^2)}{2\sigma^2})}$$

对于二维高斯函数，查找经验可知一般取窗口（也就是h的维度）一般为6σ+1。

为使得计算量不至于太大，取σ=1，也即h为7*7矩阵。

为了保证卷积质量（不降低图像的整体灰度值），生成函数在返回二维高斯函数之前会进行归一化处理。

此外，在生成函数中对h进行了奇数维度`assert`，确保程序不错误调用。

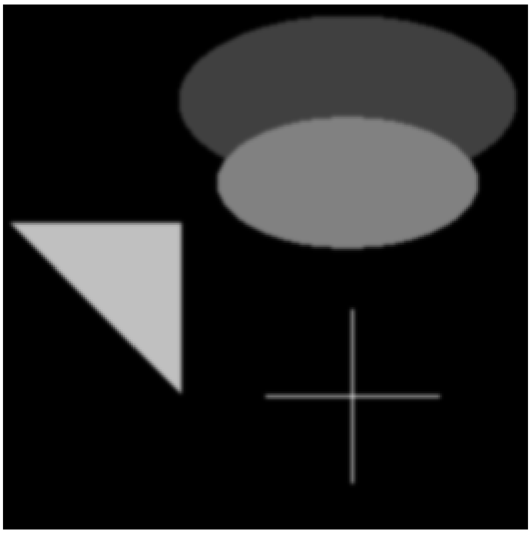
代码:

```
    //step 2卷积
    Mat srcDC = imread("GT.bmp", 0);
    if (!srcDC.data)return -1;
    Mat DCtmp = convSecondGaussian(srcDC);
    Mat resDC;
    DCtmp.convertTo(resDC, CV_8UC1);
    imshow("DoConv", resDC);//262*262
    imwrite("DoConv.bm, resDC);

//将图像与二维高斯函数作卷积
Mat convSecondGaussian(Mat& srcImage){
    Mat h_Gaussian = generateSecondGaussianMat(3, 1);
    //Mat resultImage = srcImage.clone();     //深拷贝,克隆
    Mat resultImage = secondConv(srcImage, h_Gaussian);
    return resultImage;
}
Mat generateSecondGaussianMat(int p, double sigma, bool normalization = true) {
```

```
    Mat res(2*p+1, 2*p+1, CV_64F);
    for (int i = 0;i <= p;++i) {
        for (int j = 0;j <= p;++j) {
            double tmp = 1.0 / (2.0 * CV_PI * sigma * sigma) * exp(-(j * j + i *
i) / (2.0 * sigma * sigma));
            res.at<double>(p+i, p + j) = tmp;
        }
    }
    for (int i = 0;i <= p;++i) {
        for (int j = -1;j >= -p;--j) {
            res.at<double>(p + i, p + j) = res.at<double>(p + i, p -j);
        }
    }
    for (int i = -1;i >= -p;--i) {
        for (int j = -p;j <= p;++j) {
            res.at<double>(p + i, p + j) = res.at<double>(p - i, p + j);
        }
    }
    if (normalization) {
        double s = cv::sum(res).val[0];
        res = res / s;
    }
    return res;
}

//两矩阵做卷积，前者为<uchar/double>x，后者为<double>h，注意h行列一定为奇数
Mat secondConv(Mat& x, Mat& h)
{
    int m = x.rows;
    int n = x.cols;
    int p1 = h.rows;
    assert(p1 % 2 == 1);
    p1 = (p1 - 1) / 2;
    int p2 = h.cols;
    assert(p2 % 2 == 1);
    p2 = (p2 - 1) / 2;
    double temp;
    //typedef var x.type() ? double : uchar;
    Mat res(m + 2 * p1, n + 2 * p2, CV_64F);
    if (x.type()!=0) {
        for (int s1 = 0; s1 < res.rows;++s1) {
            for (int s2 = 0;s2 < res.cols;++s2) {
                temp = 0;
                for (int i = max(s1 - 2 * p1, 0);i <= min(s1, m - 1);++i) {
                    for (int j = max(s2 - 2 * p2, 0);j <= min(s2, n - 1);++j) {
                        temp += x.at<double>(i, j) * h.at<double>(s1 - i, s2 -
j);
                    }
                }
                res.at<double>(s1, s2) = temp;
            }
        }
    }
    else{
        for (int s1 = 0; s1 < res.rows;++s1) {
            for (int s2 = 0;s2 < res.cols;++s2) {
                temp = 0;
                for (int i = max(s1 - 2 * p1, 0);i <= min(s1, m - 1);++i) {
```

```
                    for (int j = max(s2 - 2 * p2, 0);j <= min(s2, n - 1);++j) {
                        temp += double(x.at<uchar>(i, j)) * h.at<double>(s1 - i, s2 - j);
                    }
                }
                res.at<double>(s1, s2) = temp;
            }
        }
    }
    return res;
}
```

结果：



3）对2）产生的图叠加高斯噪声

依赖：

`stdlib.h`

思路：

在输入图片拉伸后的维度上利用`rand()`随机添加噪点，满足所有噪点的灰度值分布符合所规定的高斯分布即可。

已知，求服从N(μ, σ) 的X 相当与对服从N(0,1)的Z进行如下线性变换：

$$x = z * \sigma + \mu$$

对于z，易知其概率密度函数：

$$f(z) = \frac{1}{\sqrt{2\pi}} e^{\frac{-z^2}{2}}$$

在代码生成上，参考Java8的API，使用Box-Muller变换原理的改进方法——Marsaglia polar method，简要原理如下：

$$若随机变量U,V服从U(-1,1)，且S = U^2 + V^2 < 1，则：$$

$$X = U\sqrt{\frac{-2lnS}{S}}, Y = V\sqrt{\frac{-2lnS}{S}} X, Y独立且满足标准正态分布。$$

详细数学推导见参考资料。

该方法的优点是不用计算`sin`和`cos`。

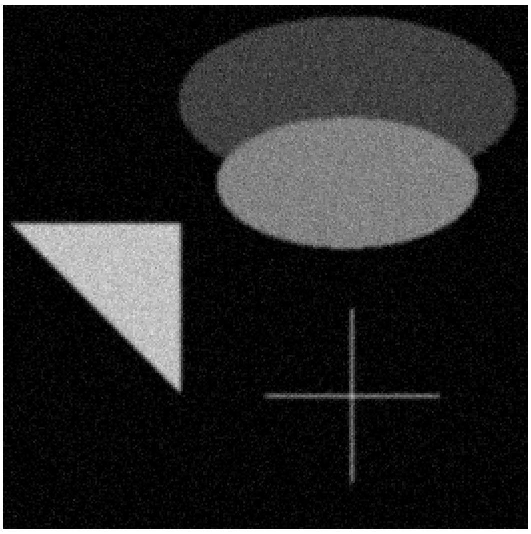
添加噪声均值为0，标准差为10，在图像灰度变为负值时使用`clip`操作。

代码：

```
    //step 3加噪声
    Mat srcAGN = imread("DoConv.bmp", 0);
    //cout << srcImg2.type();
    if (!srcAGN.data)return -1;
    imshow("srcAGN", srcAGN);
    Mat AGNtmp = addGaussianNoise(srcAGN);
    Mat resAGN;
    AGNtmp.convertTo(resAGN, CV_8UC1);
    imshow("AddGaussianNoise", resAGN);//262*262
    imwrite("AddGaussianNoise.bmp", resAGN);

Mat addGaussianNoise(Mat& srcImage)
{
    Mat resultImage = srcImage.clone();    //深拷贝,克隆
    int channels = resultImage.channels();    //获取图像的通道
    int nRows = resultImage.rows;    //图像的行数

    int nCols = resultImage.cols * channels;   //图像的总列数
    //判断图像的连续性
    if (resultImage.isContinuous())    //判断矩阵是否连续，若连续，我们相当于只需要遍历一
个一维数组
    {
        nCols *= nRows;
        nRows = 1;
    }
    for (int i = 0; i < nRows; i++)
    {
        for (int j = 0; j < nCols; j++)
        {    //添加高斯噪声，同时做clip操作
            int val = resultImage.ptr<uchar>(i)[j] + generateGaussianNoise(0,
10);
            if (val < 0)
                val = 0;
            if (val > 255)
                val = 255;
            resultImage.ptr<uchar>(i)[j] = (uchar)val;
        }
    }
    return resultImage;
}

double generateGaussianNoise(double mu, double sigma)
{
    const double epsilon = numeric_limits<double>::min();
    double u1, u2, U, V, S;
    do
    {
        u1 = rand() * (1.0 / RAND_MAX);
        u2 = rand() * (1.0 / RAND_MAX);
        U = 2.0 * (u1 - 0.5);
        V = 2.0 * (u2 - 0.5);
        S = U * U + V * V;
```

```
    } while (u1 <= epsilon||S>1||S<=epsilon);

    double Z = U * sqrt((-2.0 * log(S) / S));
    return Z * sigma + mu;
}
```

结果:



4）利用非负逆卷积方法对3）的图进行处理

依赖:

`cv::Mat`、`cv::Scalar`、`cv::Range`

思路:

根据老师上课讲的内容，通过取中间块和倒序翻转h进行快速逆卷积操作，再在最外层进行梯度递降迭代达到优化目的。

梯度递降函数使用了函数参数，方便日后工作对于损失函数和函数梯度的改写。

在如何验证本程序的效力上，笔者考虑逆卷积问题的定义是已知Y和h，求x，因此梯度递降起始点x_0 的选取非常重要。共设置了三种不同的起始图像:

1. 全0图像
2. 总体均匀分布图像（U(0，1))
3. 总体高斯分布图像（N(0,1))

迭代次数设置了100、200、300三种情况。

代码:

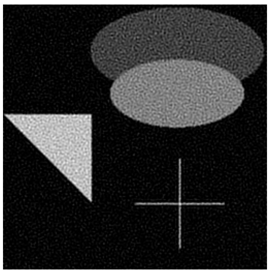
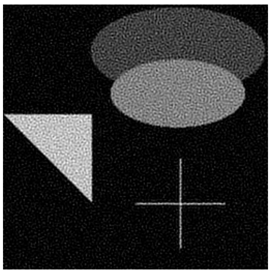
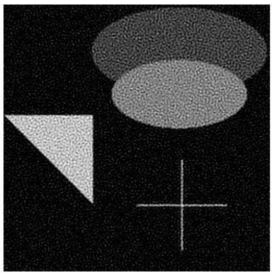
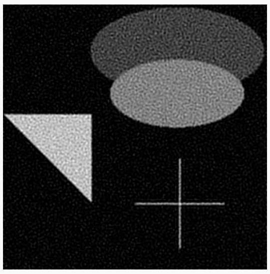
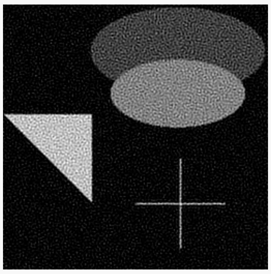
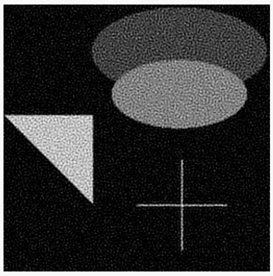
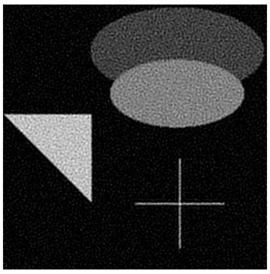
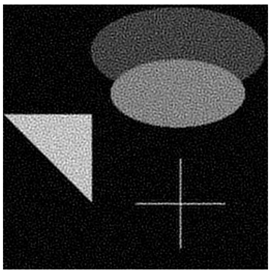
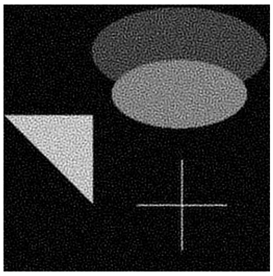
```
    //step4逆卷积
    Mat Y;
    imread("AddGaussianNoise.bmp", 0).convertTo(Y, CV_64F);
    if (!Y.data)return -1;
    //x1为0
    Mat DICtmp1 = Mat::zeros(256, 256, CV_64F);
    Mat resDIC1;

    DICtmp1 = GD(F, dF, 0.3, 100, DICtmp1, Y);
    DICtmp1.convertTo(resDIC1, CV_8UC1);
    imshow("DoInverseConv_InputZero_100", resDIC1);
    imwrite("DoInverseConv_InputZero_100.bmp", resDIC1);//100
```

```
    DICtmp1 = Mat::zeros(256, 256, CV_64F);
    DICtmp1 = GD(F, dF, 0.3, 200, DICtmp1, Y);
    DICtmp1.convertTo(resDIC1, CV_8UC1);
    imshow("DoInverseConv_InputZero_200", resDIC1);
    imwrite("DoInverseConv_InputZero_200.bmp", resDIC1);//200
    DICtmp1 = Mat::zeros(256, 256, CV_64F);
    DICtmp1 = GD(F, dF, 0.3, 300, DICtmp1, Y);
    DICtmp1.convertTo(resDIC1, CV_8UC1);
    imshow("DoInverseConv_InputZero_300", resDIC1);
    imwrite("DoInverseConv_InputZero_300.bmp", resDIC1);//300
    //x1为均匀随机分布
    Mat DICtmp2(256, 256, CV_64F);
    randu(DICtmp2, Scalar::all(0.), Scalar::all(1.));
    Mat resDIC2;

    DICtmp2 = GD(F, dF, 0.3, 100, DICtmp2, Y);
    DICtmp2.convertTo(resDIC2, CV_8UC1);
    imshow("DoInverseConv_InputUnion_100", resDIC2);
    imwrite("DoInverseConv_InputUnion_100.bmp", resDIC2);//100
    randu(DICtmp2, Scalar::all(0.), Scalar::all(1.));
    DICtmp2 = GD(F, dF, 0.3, 200, DICtmp2, Y);
    DICtmp2.convertTo(resDIC2, CV_8UC1);
    imshow("DoInverseConv_InputUnion_200", resDIC2);
    imwrite("DoInverseConv_InputUnion_200.bmp", resDIC2);//200
    randu(DICtmp2, Scalar::all(0.), Scalar::all(1.));
    DICtmp2 = GD(F, dF, 0.3, 300, DICtmp2, Y);
    DICtmp2.convertTo(resDIC2, CV_8UC1);
    imshow("DoInverseConv_InputUnion_300", resDIC2);
    imwrite("DoInverseConv_InputUnion_300.bmp", resDIC2);//300
    //x1为高斯随机分布
    Mat DICtmp3(256, 256, CV_64F);
    randn(DICtmp3, Scalar::all(0.), Scalar::all(1.));
    Mat resDIC3;

    DICtmp3 = GD(F, dF, 0.3, 100, DICtmp3, Y);
    DICtmp3.convertTo(resDIC3, CV_8UC1);
    imshow("DoInverseConv_InputGaussian_100", resDIC3);
    imwrite("DoInverseConv_InputGaussian_100.bmp", resDIC3);//100
    randn(DICtmp3, Scalar::all(0.), Scalar::all(1.));
    DICtmp3 = GD(F, dF, 0.3, 200, DICtmp3, Y);
    DICtmp3.convertTo(resDIC3, CV_8UC1);
    imshow("DoInverseConv_InputGaussian_200", resDIC3);
    imwrite("DoInverseConv_InputGaussian_200.bmp", resDIC3);//200
    randn(DICtmp3, Scalar::all(0.), Scalar::all(1.));
    DICtmp3 = GD(F, dF, 0.3, 300, DICtmp3, Y);
    DICtmp3.convertTo(resDIC3, CV_8UC1);
    imshow("DoInverseConv_InputGaussian_300", resDIC3);
    imwrite("DoInverseConv_InputGaussian_300.bmp", resDIC3);//300
Mat GD(double(*f)(Mat& h, Mat& x, Mat& Y), Mat(*df)(Mat& h, Mat& x, Mat& Y),
double delta, int iter,Mat x,Mat& Y) {
    Mat h_Gaussian = generateSecondGaussianMat(3, 1);
    //Mat tmp(x.rows,x.cols,CV_64F);
    while (iter--) {
        cout << iter<<endl;
        Mat tmp = x-delta*df(h_Gaussian, x, Y);
        if (f(h_Gaussian, tmp, Y) <= f(h_Gaussian, x, Y))
            x = tmp;
        else {
```

```
            iter++;
            delta /= 2;
        }
    }
    return x;
}
double F(Mat& h, Mat& x, Mat& Y) {
    //secondConv定义见作业步骤2)
    Mat diff = Y - secondConv(x, h);
    int nRows = diff.rows;
    int nCols = diff.cols * diff.channels();
    if (diff.isContinuous()) {
        nCols *= nRows;
        nRows = 1;
    }
    for (int h = 0; h < nRows; ++h)
    {
        double* ptr = diff.ptr<double>(h);
        for (int w = 0; w < nCols; ++w)
        {
            ptr[w] *= ptr[w];
        }
    }
    return sum(diff).val[0];
}
Mat dF(Mat& h, Mat& x, Mat& Y) {
    //secondConv定义见作业步骤2)
    Mat tmpConv = secondConv(x, h);
    tmpConv -= Y;
    Mat h_bar = bar(h);
    Mat tmp = secondConv(tmpConv, h_bar);
    Mat res = M(tmp, h.rows-1, h.cols-1,x.rows,x.cols);
    return res;
}
//将h倒序
Mat bar(Mat& src) {
    Mat src_bar = src.clone();
    int nRows = src.rows;
    int nCols = src.cols * src.channels();
    /*
    if (src_bar.isContinuous()&&src.isContinuous()) {
        nCols *= nRows;
        nRows = 1;
    }
    */
    for (int h = 0; h < nRows; ++h)
    {
        double* ptr = src_bar.ptr<double>(h);
        //double* ptro = src.ptr<double>(h);
        for (int w = 0; w < nCols; ++w)
        {
            ptr[w] = src.at<double>(nRows-h-1,nCols-w-1);
        }
    }
    return src_bar;
}
//取中间块
Mat M(Mat& src, int doublep1, int doublep2,int m,int n) {
```

```
    Mat m0 = src(cv::Range(doublep1, m+ doublep1), cv::Range(doublep2,
n+doublep2));//注意，Range(s,e)得到[s,e)
    return m0;
}
```

结果：

起始图像\迭代次数	**100**	**200**	**300**
全0图像			
总体均匀分布图像（U(0，1))			
总体高斯分布图像（N(0,1))			

参考资料：

Beck A, Teboulle M. A fast iterative shrinkage-thresholding algorithm for linear inverse problems[J]. SIAM journal on imaging sciences, 2009, 2(1): 183-202

[高斯分布的生成 - 简书]

https://zhuanlan.zhihu.com/p/143264646

https://en.wikipedia.org/wiki/Marsaglia_polar_method#Theoretical_basis

附录：所有代码

```
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <cstring>
#include <cstdio>
#include <string>
#include <sstream>
#include <fstream>
#include <opencv2/opencv.hpp>
#include <opencv2/core/core.hpp>
```

```
#include <opencv2/highgui/highgui.hpp>
#include <iostream>
#include <assert.h>
#define PI CV_PI

typedef long long ll;
using namespace cv;
using namespace std;


Mat matrix_multiply(Mat& mat_a, Mat& mat_b);
Mat M(Mat& src,int doublep1,int doublep2,int m,int n);
Mat bar(Mat& src);
double generateGaussianNoise(double mu, double sigma);
Mat generateSecondGaussianMat(int p, double sigma, bool normalization);
Mat secondConv(Mat& x, Mat& h);
Mat addGaussianNoise(Mat& srcImage);
Mat convSecondGaussian(Mat& srcImage);
Mat GD(double(*f)(Mat& h, Mat& x, Mat& Y),Mat (*df)(Mat&h,Mat&x,Mat&Y),double
delta,int iter,Mat x,Mat& Y);
double F(Mat& h, Mat& x, Mat& Y);
Mat dF(Mat& h, Mat& x, Mat& Y);

int main(int argc, char** argv) {
    //step 1画出图片
    Mat GT = imread("screenshot.bmp", 0);
    if (!GT.data)return -1;
    GT.convertTo(GT, CV_8UC1);
    imshow("GT", GT);
    imwrite("GT.bmp", GT);

    //step 2卷积
    Mat srcDC = imread("GT.bmp", 0);
    if (!srcDC.data)return -1;
    Mat DCtmp = convSecondGaussian(srcDC);
    Mat resDC;
    DCtmp.convertTo(resDC, CV_8UC1);
    imshow("DoConv", resDC);//262*262
    imwrite("DoConv.bmp", resDC);
    //waitKey(0);

    //step 3加噪声
    Mat srcAGN = imread("DoConv.bmp", 0);
    //cout << srcImg2.type();
    if (!srcAGN.data)return -1;
    imshow("srcAGN", srcAGN);
    Mat AGNtmp = addGaussianNoise(srcAGN);
    Mat resAGN;
    AGNtmp.convertTo(resAGN, CV_8UC1);
    imshow("AddGaussianNoise", resAGN);//262*262
    imwrite("AddGaussianNoise.bmp", resAGN);
    //waitKey(0);

    //step4逆卷积
    Mat Y;
    imread("AddGaussianNoise.bmp", 0).convertTo(Y, CV_64F);
    if (!Y.data)return -1;
    //x1为0
```

```
    Mat DICtmp1 = Mat::zeros(256, 256, CV_64F);
    Mat resDIC1;

    DICtmp1 = GD(F, dF, 0.3, 100, DICtmp1, Y);
    DICtmp1.convertTo(resDIC1, CV_8UC1);
    imshow("DoInverseConv_InputZero_100", resDIC1);
    imwrite("DoInverseConv_InputZero_100.bmp", resDIC1);//100
    DICtmp1 = Mat::zeros(256, 256, CV_64F);
    DICtmp1 = GD(F, dF, 0.3, 200, DICtmp1, Y);
    DICtmp1.convertTo(resDIC1, CV_8UC1);
    imshow("DoInverseConv_InputZero_200", resDIC1);
    imwrite("DoInverseConv_InputZero_200.bmp", resDIC1);//200
    DICtmp1 = Mat::zeros(256, 256, CV_64F);
    DICtmp1 = GD(F, dF, 0.3, 300, DICtmp1, Y);
    DICtmp1.convertTo(resDIC1, CV_8UC1);
    imshow("DoInverseConv_InputZero_300", resDIC1);
    imwrite("DoInverseConv_InputZero_300.bmp", resDIC1);//300
    //x1为均匀随机分布
    Mat DICtmp2(256, 256, CV_64F);
    randu(DICtmp2, Scalar::all(0.), Scalar::all(1.));
    Mat resDIC2;

    DICtmp2 = GD(F, dF, 0.3, 100, DICtmp2, Y);
    DICtmp2.convertTo(resDIC2, CV_8UC1);
    imshow("DoInverseConv_InputUnion_100", resDIC2);
    imwrite("DoInverseConv_InputUnion_100.bmp", resDIC2);//100
    randu(DICtmp2, Scalar::all(0.), Scalar::all(1.));
    DICtmp2 = GD(F, dF, 0.3, 200, DICtmp2, Y);
    DICtmp2.convertTo(resDIC2, CV_8UC1);
    imshow("DoInverseConv_InputUnion_200", resDIC2);
    imwrite("DoInverseConv_InputUnion_200.bmp", resDIC2);//200
    randu(DICtmp2, Scalar::all(0.), Scalar::all(1.));
    DICtmp2 = GD(F, dF, 0.3, 300, DICtmp2, Y);
    DICtmp2.convertTo(resDIC2, CV_8UC1);
    imshow("DoInverseConv_InputUnion_300", resDIC2);
    imwrite("DoInverseConv_InputUnion_300.bmp", resDIC2);//300
    //x1为高斯随机分布
    Mat DICtmp3(256, 256, CV_64F);
    randn(DICtmp3, Scalar::all(0.), Scalar::all(1.));
    Mat resDIC3;

    DICtmp3 = GD(F, dF, 0.3, 100, DICtmp3, Y);
    DICtmp3.convertTo(resDIC3, CV_8UC1);
    imshow("DoInverseConv_InputGaussian_100", resDIC3);
    imwrite("DoInverseConv_InputGaussian_100.bmp", resDIC3);//100
    randn(DICtmp3, Scalar::all(0.), Scalar::all(1.));
    DICtmp3 = GD(F, dF, 0.3, 200, DICtmp3, Y);
    DICtmp3.convertTo(resDIC3, CV_8UC1);
    imshow("DoInverseConv_InputGaussian_200", resDIC3);
    imwrite("DoInverseConv_InputGaussian_200.bmp", resDIC3);//200
    randn(DICtmp3, Scalar::all(0.), Scalar::all(1.));
    DICtmp3 = GD(F, dF, 0.3, 300, DICtmp3, Y);
    DICtmp3.convertTo(resDIC3, CV_8UC1);
    imshow("DoInverseConv_InputGaussian_300", resDIC3);
    imwrite("DoInverseConv_InputGaussian_300.bmp", resDIC3);//300

    waitKey(0);
    return 0;
```

```
}

Mat matrix_multiply(Mat& mat_a, Mat& mat_b) {//矩阵乘法
    assert(mat_a.cols == mat_b.rows);
    int dimi = mat_a.rows;
    int dimj = mat_b.rows;
    int dimk = mat_b.cols;
    Mat mat_c(dimi,dimk,CV_64F);
    for (int i = 0; i < dimi; ++i) {
        for (int k = 0; k < dimk; ++k) {
            mat_c.at<double>(i,k) = 0.0;
            for (int j = 0; j < dimj; ++j) {
                mat_c.at<double>(i,k) += mat_a.at<double>(i,j) *
mat_b.at<double>(j,k);
            }
        }
    }
    return mat_c;
}
//取中间块
Mat M(Mat& src, int doublep1, int doublep2,int m,int n) {
    Mat m0 = src(cv::Range(doublep1, m+ doublep1), cv::Range(doublep2,
n+doublep2));//注意，Range(s,e)得到[s,e)
    return m0;
}
//倒序矩阵
Mat bar(Mat& src) {
    Mat src_bar = src.clone();
    int nRows = src.rows;
    int nCols = src.cols * src.channels();
    /*
    if (src_bar.isContinuous()&&src.isContinuous()) {
        nCols *= nRows;
        nRows = 1;
    }
    */
    for (int h = 0; h < nRows; ++h)
    {
        double* ptr = src_bar.ptr<double>(h);
        //double* ptro = src.ptr<double>(h);
        for (int w = 0; w < nCols; ++w)
        {
            ptr[w] = src.at<double>(nRows-h-1,nCols-w-1);
        }
    }
    return src_bar;
}
//两矩阵做卷积，前者为<uchar/double>x，后者为<double>h，注意h行列一定为奇数
Mat secondConv(Mat& x, Mat& h)
{
    int m = x.rows;
    int n = x.cols;
    int p1 = h.rows;
    assert(p1 % 2 == 1);
    p1 = (p1 - 1) / 2;
    int p2 = h.cols;
    assert(p2 % 2 == 1);
    p2 = (p2 - 1) / 2;
```

```
    double temp;
    //typedef var x.type() ? double : uchar;
    Mat res(m + 2 * p1, n + 2 * p2, CV_64F);
    if (x.type()!=0) {
        for (int s1 = 0; s1 < res.rows;++s1) {
            for (int s2 = 0;s2 < res.cols;++s2) {
                temp = 0;
                for (int i = max(s1 - 2 * p1, 0);i <= min(s1, m - 1);++i) {
                    for (int j = max(s2 - 2 * p2, 0);j <= min(s2, n - 1);++j) {
                        temp += x.at<double>(i, j) * h.at<double>(s1 - i, s2 -
j);
                    }
                }
                res.at<double>(s1, s2) = temp;
            }
        }
    }
    else{
        for (int s1 = 0; s1 < res.rows;++s1) {
            for (int s2 = 0;s2 < res.cols;++s2) {
                temp = 0;
                for (int i = max(s1 - 2 * p1, 0);i <= min(s1, m - 1);++i) {
                    for (int j = max(s2 - 2 * p2, 0);j <= min(s2, n - 1);++j) {
                        temp += double(x.at<uchar>(i, j)) * h.at<double>(s1 - i,
s2 - j);
                    }
                }
                res.at<double>(s1, s2) = temp;
            }
        }
    }
    return res;
}

double generateGaussianNoise(double mu, double sigma)
{
    const double epsilon = numeric_limits<double>::min();
    double u1, u2, U, V, S;
    do
    {
        u1 = rand() * (1.0 / RAND_MAX);
        u2 = rand() * (1.0 / RAND_MAX);
        U = 2.0 * (u1 - 0.5);
        V = 2.0 * (u2 - 0.5);
        S = U * U + V * V;
    } while (u1 <= epsilon||S>1||S<=epsilon);

    double Z = U * sqrt((-2.0 * log(S) / S));
    return Z * sigma + mu;
}

Mat generateSecondGaussianMat(int p, double sigma, bool normalization = true) {
    Mat res(2*p+1, 2*p+1, CV_64F);
    for (int i = 0;i <= p;++i) {
        for (int j = 0;j <= p;++j) {
            double tmp = 1.0 / (2.0 * CV_PI * sigma * sigma) * exp(-(j * j + i *
i) / (2.0 * sigma * sigma));
            res.at<double>(p+i, p + j) = tmp;
```

```
        }
    }
    for (int i = 0;i <= p;++i) {
        for (int j = -1;j >= -p;--j) {
            res.at<double>(p + i, p + j) = res.at<double>(p + i, p -j);
        }
    }
    for (int i = -1;i >= -p;--i) {
        for (int j = -p;j <= p;++j) {
            res.at<double>(p + i, p + j) = res.at<double>(p - i, p + j);
        }
    }
    if (normalization) {
        double s = cv::sum(res).val[0];
        res = res / s;
    }
    return res;
}
//为图像添加高斯噪声
Mat addGaussianNoise(Mat& srcImage)
{
    Mat resultImage = srcImage.clone();    //深拷贝,克隆
    int channels = resultImage.channels();    //获取图像的通道
    int nRows = resultImage.rows;    //图像的行数

    int nCols = resultImage.cols * channels;   //图像的总列数
    //判断图像的连续性
    if (resultImage.isContinuous())    //判断矩阵是否连续，若连续，我们相当于只需要遍历一
个一维数组
    {
        nCols *= nRows;
        nRows = 1;
    }
    for (int i = 0; i < nRows; i++)
    {
        for (int j = 0; j < nCols; j++)
        {    //添加高斯噪声
            int val = resultImage.ptr<uchar>(i)[j] + generateGaussianNoise(0,
10);
            if (val < 0)
                val = 0;
            if (val > 255)
                val = 255;
            resultImage.ptr<uchar>(i)[j] = (uchar)val;
        }
    }
    return resultImage;
}
//将图像与二维高斯函数作卷积
Mat convSecondGaussian(Mat& srcImage){
    Mat h_Gaussian = generateSecondGaussianMat(3, 1);
    //Mat resultImage = srcImage.clone();    //深拷贝,克隆
    Mat resultImage = secondConv(srcImage, h_Gaussian);
    return resultImage;
}
Mat GD(double(*f)(Mat& h, Mat& x, Mat& Y), Mat(*df)(Mat& h, Mat& x, Mat& Y),
double delta, int iter,Mat x,Mat& Y) {
    Mat h_Gaussian = generateSecondGaussianMat(3, 1);
```

```
    //Mat tmp(x.rows,x.cols,CV_64F);
    while (iter--) {
        cout << iter<<endl;
        Mat tmp = x-delta*df(h_Gaussian, x, Y);
        if (f(h_Gaussian, tmp, Y) <= f(h_Gaussian, x, Y))
            x = tmp;
        else {
            iter++;
            delta /= 2;
        }
    }
    return x;
}
double F(Mat& h, Mat& x, Mat& Y) {
    Mat diff = Y - secondConv(x, h);
    int nRows = diff.rows;
    int nCols = diff.cols * diff.channels();
    if (diff.isContinuous()) {
        nCols *= nRows;
        nRows = 1;
    }
    for (int h = 0; h < nRows; ++h)
    {
        double* ptr = diff.ptr<double>(h);
        for (int w = 0; w < nCols; ++w)
        {
            ptr[w] *= ptr[w];
        }
    }
    return sum(diff).val[0];
}
Mat dF(Mat& h, Mat& x, Mat& Y) {
    Mat tmpConv = secondConv(x, h);
    tmpConv -= Y;
    Mat h_bar = bar(h);
    Mat tmp = secondConv(tmpConv, h_bar);
    Mat res = M(tmp, h.rows-1, h.cols-1,x.rows,x.cols);
    return res;
}
```