## 一、 实验目的和要求：

自选一张清晰图像
增加高斯噪声
分别用高斯滤波和双边滤波器对两类噪声图像去噪
比较滤波器对各类噪声的去噪效果

## 二、 实验环境：

设备：MacBook Pro (Retina, 13-inch, Early 2015)
操作系统：macOS High Sierra 10.13.4
IDE: Clion
编程语言：C 语言
外部库：OpenCV

## 三、 实验内容以及实验结果：

由于实验用到了外部函数库 OpenCV，这里需要用 cmake 进行关联
对应 cmakelist.txt 如下

```
project(vision_noise)
cmake_minimum_required (VERSION 2.8)

#opencv
find_package(OpenCV 3.3.1 REQUIRED)


SET(CMAKE_BUILD_TYPE "RELEASE")


add_executable(vision_noise main.cpp)
target_link_libraries (vision_noise ${OpenCV_LIBS} )
```

实验用到的函数声明

```
1.  //为图像添加椒盐噪声
2.  Mat add_Salt_Noise(const Mat srcImage, int n);
3.  //产生高斯噪声
4.  double generate_Gaussian_Noise();
5.  //为图像添加高斯噪声
6.  Mat add_Gaussian_Noise(Mat& srcImage);
```

其中，实验对图像增添椒盐噪声以及高斯噪声

***椒盐噪声***包含两种噪声，一种是盐噪声（salt noise），另一种是胡椒噪声（pepper noise）。盐噪声为白色，椒噪声为黑色。前者是高灰度噪声，灰度值为 255，后者属于低灰度噪声，灰度值为 0。一般两种噪声同时出现，呈现在图像上就是黑白杂点。

因此我们随机选取行列点，然后对其进行灰度值赋值，使之被黑白点替换，从而得到最终的加入噪声图。

对应的具体函数如下。这里由于我们实验简化目标为对于灰度图片进行处理，实际上也可以对于彩色图片进行处理，区别在于图片通道数的不同，我们在这里留下对彩色图的处理部分，便于后续探究。

```
1.  Mat add_Salt_Noise(const Mat srcImage, int n)
2.  {
3.      Mat dstImage = srcImage.clone();
4.      for (int k = 0; k < n; k++)
5.      {
6.          //随机取值行列
7.          int i = rand() % dstImage.rows;
8.          int j = rand() % dstImage.cols;
9.          //图像通道判定
10.         if (dstImage.channels() == 1)
11.         {
12.             dstImage.at<uchar>(i, j) = 255;       //盐噪声
```

```
13.         }
14.         else
15.         {
16.             //留下彩色图片通道，方便后续代码拓展
17.             dstImage.at<Vec3b>(i, j)[0] = 255;
18.             dstImage.at<Vec3b>(i, j)[1] = 255;
19.             dstImage.at<Vec3b>(i, j)[2] = 255;
20.         }
21.     }
22.     for (int k = 0; k < n; k++)
23.     {
24.         //随机取值行列
25.         int i = rand() % dstImage.rows;
26.         int j = rand() % dstImage.cols;
27.         //图像通道判定
28.         if (dstImage.channels() == 1)
29.         {
30.             dstImage.at<uchar>(i, j) = 0;      //椒噪声
31.         }
32.         else
33.         {
34.             //留下彩色图片通道，方便后续代码拓展
35.             dstImage.at<Vec3b>(i, j)[0] = 0;
36.             dstImage.at<Vec3b>(i, j)[1] = 0;
37.             dstImage.at<Vec3b>(i, j)[2] = 0;
38.         }
39.     }
40.     return dstImage;
41. }
```

效果图如下，噪声点为 2000

下面是***高斯噪声***

高斯噪声是指它的概率密度函数服从高斯分布（即正态分布）的一类噪声。常见的高斯噪声包括起伏噪声、宇宙噪声、热噪声和散粒噪声等等。除常用抑制噪声的方法外，对高斯噪声的抑制方法常常采用数理统计方法。如果一个噪声，它的幅度分布服从高斯分布，而它的功率谱密度又是均匀分布的，则称它为高斯白噪声。

首先我们根据原理产生高斯噪声

高斯噪声分布满足

```
z0 = sqrt(-2.0*log(u1))*cos(2 * CV_PI * u2);
z1 = sqrt(-2.0*log(u1))*sin(2 * CV_PI * u2);
```

因此高斯噪声生成部分对应代码为

```
double generate_Gaussian_Noise()
{
    //定义小值
    const double epsilon = numeric_limits<double>::min();
    static double z0, z1;
    static bool flag = false;
    flag = !flag;
    //构造高斯随机变量 X
    if (!flag) {
        //printf("%f\n", z1);
```

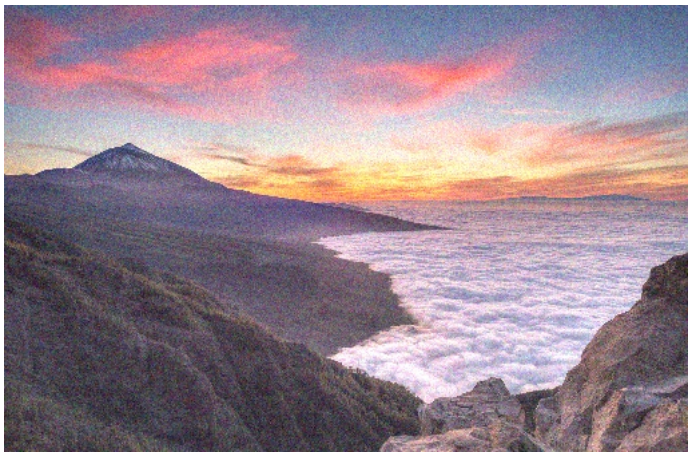
```
        return z1;
    }
    double u1, u2;
    do
    {
        u1 = rand() * (1.0 / RAND_MAX);
        u2 = rand() * (1.0 / RAND_MAX);
    } while (u1 <= epsilon);
    //构造高斯随机变量 Y
    z0 = sqrt(-2.0*log(u1))*cos(2 * CV_PI * u2);
    z1 = sqrt(-2.0*log(u1))*sin(2 * CV_PI * u2);
    //printf("%f\n",z0);
    return z0;
}
```

将其添加到图片中为

```
//为图像添加高斯噪声
Mat add_Gaussian_Noise(Mat& srcImage)
{
    Mat resultImage = srcImage.clone();
    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] + (generate_Gaussian_Noise()*0.8+2) * 20;
            //防溢出
            if (val < 0)
                val = 0;
            if (val > 255)
                val = 255;
            resultImage.ptr<uchar>(i)[j] = (uchar)val;
        }
    }
    return resultImage;
}
```

效果图如下



下面来对不同的滤波方法进行讨论

首先是***均值滤波***。

均值滤波的基本原理是，数字图像或数字序列中一点的值用该点的一个邻域中各点值的平均值代替，让周围的像素值接近的真实值。

对应的 OpenCV 函数为

```
blur(dstImage1,blur_out_1,Size(5, 5));
```

其中 Size(5, 5))代表的是邻域的大小的选择，随着邻域大小的改变滤波对于细节的保留程度也会对应发生改变，一般来讲邻域越大，细节保留越少。

当大小为 3*3 时

当大小为 5*5 时

可见随着 size 的增大，椒盐噪声的影响越来越小，但是同时图片的细节保留情况也逐渐变差。因此均值滤波的参数也要根据图片实际情况进行调节。

对于高斯噪声来说，效果图如下

可以看到对于高斯噪声有一定的消除，整体变得更为平滑，但是效果不是很明显，滤波方法不是十分适用，或者有待改进。

然后是***中值滤波***。

中值滤波的基本原理是，数字图像或数字序列中一点的值用该点的一个邻域中各点值的中值代替，让周围的像素值接近的真实值，从而消除孤立的噪声点，因此较为适合处理椒盐噪声一类由孤立点构成的噪声。

对应的 OpenCV 代码为

```
1.  medianBlur(dstImage1, median_blur_out_1, 3);
```

其中 3 对应的也为邻域的大小

当邻域大小为 3 时

当邻域大小为 5 时

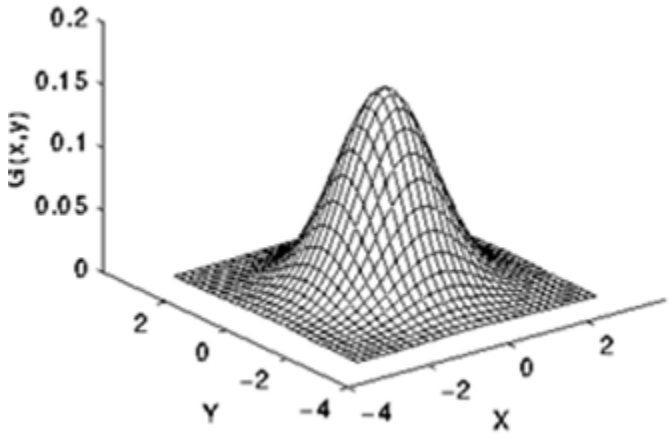
可以看到中值滤波对于孤立点的处理相对于均值滤波更好，因为中值的选择可以更好的避开椒盐噪声对应的极端点，在统计学的原理上避开了极端情况。但是它也存在均值滤波的细节保留问题，随着邻域选择的增大，图片逐渐变得模糊，因此在实际应用中要加以考虑。

对于高斯噪声来讲，处理效果如下

可以看到中值滤波对于高斯噪声有一定的作用，整体图片变得更为平滑，噪声影响有一定的降低。

下面是***高斯滤波***。

显然高斯滤波对于高斯噪声有更好的效果，具体原理为利用高斯核对图片进行卷积，对邻域内像素进行卷积以及归一化从而获得选定点的像素值。对应的权值分布如下




对应的 OpenCV 函数为

```
1.  GaussianBlur(dstImage1, gaussian_blur_out_1, Size(3, 3), 0, 0);
```

同样对于不同的邻域大小

当邻域为 3*3 时

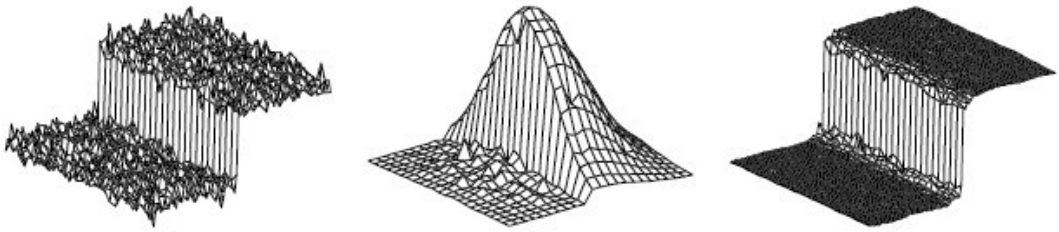
当邻域为 7*7 时

可以看到高斯滤波有效的减少了高斯噪声的影响，并且较好的保留了图片的细节。但是同样也会随着邻域选择的增大而影响到整体的图片细节保留。

同时，高斯滤波实际上也是采用的加权平均的方式，不可避免的受到极值点的影响，因此对于椒盐噪声的处理效果不佳。效果如下。

最后是***双边滤波***。

双边滤波的最大特点是对于滤波中遇到的较大色差像素点采用不同的滤波权值，使得颜色相近的点的权值更大，色差较大的点的权值较小。这样的策略在边缘处得到更好的体现，效果如图所示。这样可以对于图像中的边缘做出很好的保留。



(a) (b) (c)

那么双边滤波的具体操作也即遍历像素后对于每个像素，利用其邻域内的像素值的加权平均对原像素进行替换，如下式

$$g(i,j)=\frac{\sum_{k,l} f(k,l)w(i,j,k,l)}{\sum_{k,l} w(i,j,k,l)}$$

其中权值对应有两部分

首先是几何空间距离对应权重

$$d(i,j,k,l)=\exp\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2}\right)$$

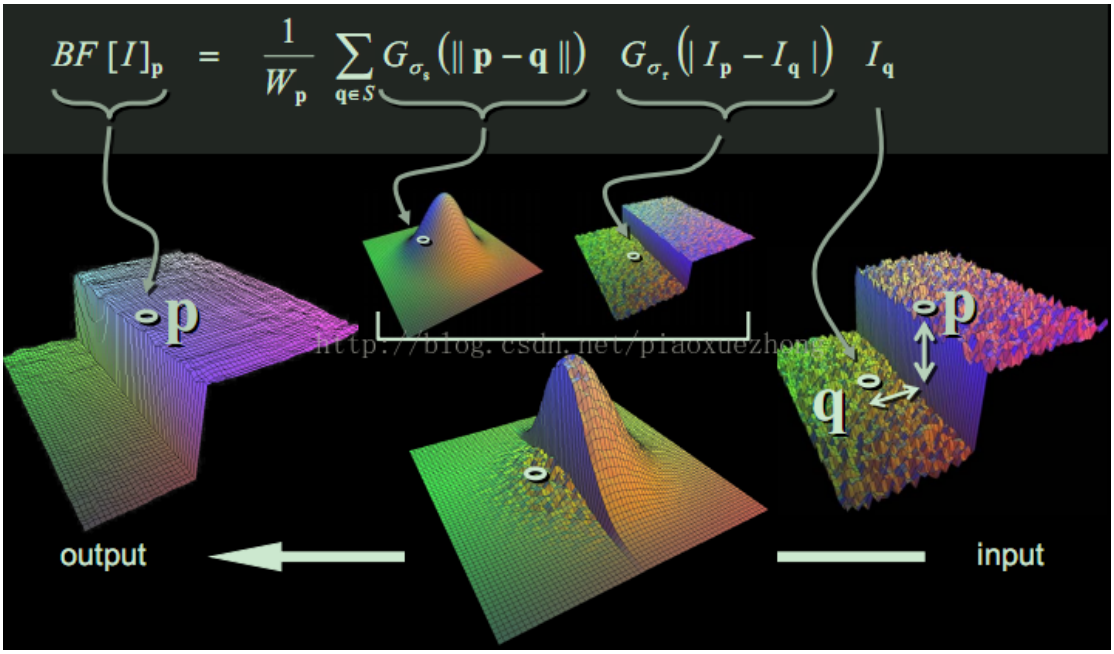
其次是像素值差异对应权重

$$r(i,j,k,l)=\exp\left(-\frac{\|f(i,j)-f(k,l)\|^2}{2\sigma_r^2}\right)$$

最终对应整体权重为

$$w(i,j,k,l)=\exp\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2}-\frac{\|f(i,j)-f(k,l)\|^2}{2\sigma_r^2}\right)$$

下图为原论文中作者提供的说明图片，可以看到在像素值存在较大差异时，像素值小的部分几何距离产生的权重被削弱，使得原有的边沿得到了保留，而在像素值差异边界的两端仍保持高斯滤波特性，也即实现了降噪的同时保持边沿。




对应前文所述公式可以简单分析如下

随着$\sigma_r$的增加可以看到像素差异对应权重逐渐趋近于 1，也即像素值差异起到的作用逐渐变小，也就是逐渐逼近高斯滤波。

而随着$\sigma_d$的增加实际上对应的高斯滤波中模糊范围的增大，选取过大会导致图像信息的丢失。

对应 OpenCV 代码为

```
1.  bilateralFilter(dstImage1, bilateral_blur_out_1, 5, 50, 10);
```

代码运行结果如下

可以看到滤波后对于色块的分区仍较为明显，对于图片细节的保留较好，且对于噪声点有明显的去除，对比来看在几种滤波方法中最好。

但是由于仍采用加权平均的方式，双边滤波仍会受到极值点的影响较大，因此对于椒盐噪声处理效果不佳。效果如下

**四、 实验探究：**

整体来讲，椒盐噪声对应的是位置相对随机，颜色较为极端的噪声情况，而高斯噪声对应的是幅值以及位置都相对较为随机但是较为平均的情况。

滤波方法上，均值噪声对于各种噪声都有一定的处理能力，但是效果一般。中值滤波从统计学原理上削弱了极端值的影响，对于椒盐噪声效果更为明显，对于高斯噪声也有一定的处理能力。高斯滤波对于高斯噪声有更好的处理能力，而对于椒盐噪声仍采用了加权平均的方式因此处理效果不佳。双边滤波更关注于边缘的保留，对于椒盐噪声以及高斯噪声都有一定的处理能力，对于图片的细节保留更好。

因此总结下来，中值滤波更适用于椒盐噪声，高斯滤波更适用于高斯噪声，双边滤波更适用于对于边缘保留要求高的情况，均值滤波相对更为平均，也具有一定的处理能力。

**五、 问题总结以及心得体会：**

本次实验相对较为简单，帮助我们对于图片中存在的各种噪声以及处理方法有了初步的理解，对于图片的处理方法也有了一定的认识，对于处理的工具也进行了初步的应用，收获良多。

**六、 代码：**

见附件