

# 添加噪声并滤波

姓名：胡天扬

学号：3190105708

专业：自动化（控制）

课程：数字图像处理与机器视觉

指导教师：姜伟

## 一、题目要求

任意选取1张图像，添加高斯噪声和椒盐噪声。实现均值滤波、中值滤波和双边滤波等去噪方法，比较各方法去噪结果，并分析各去噪方法的特点。

## 二、原图

原图是一张三通道的彩色图，像素为 `1080 × 1440`。



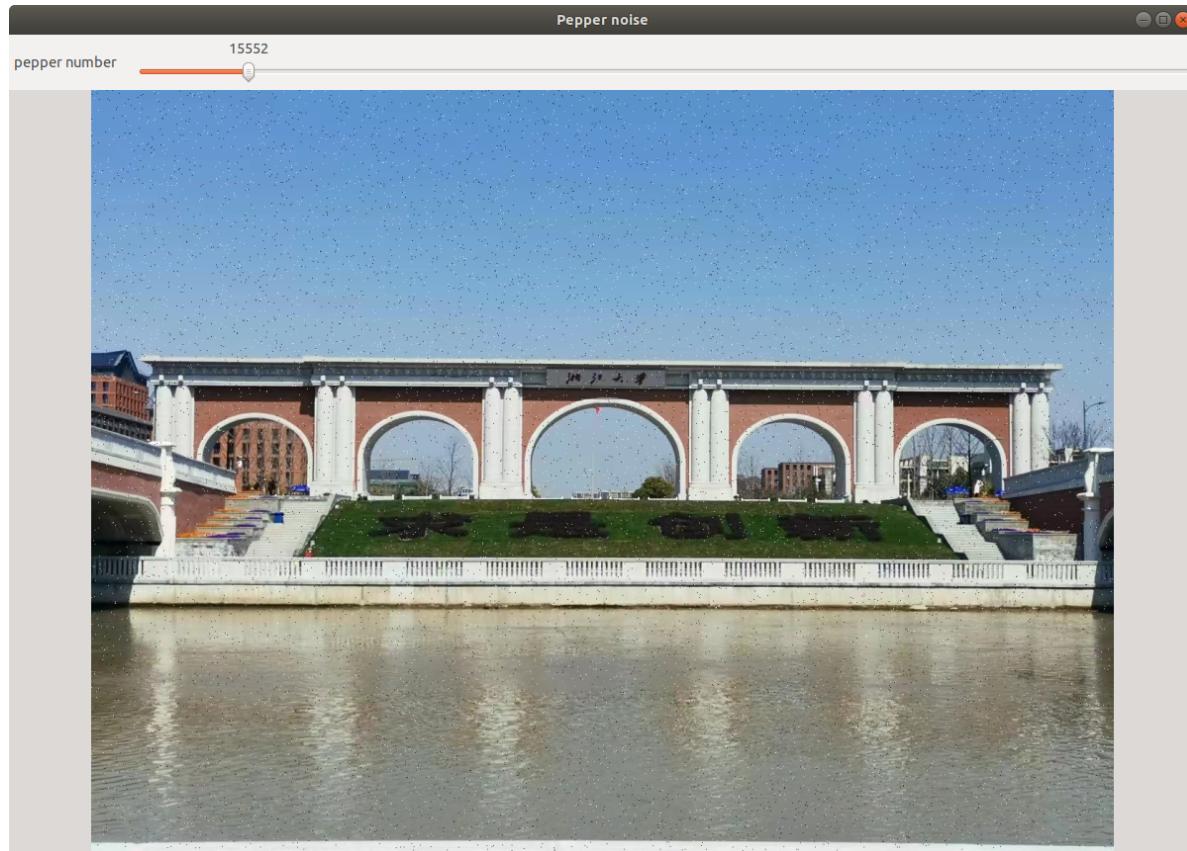
## 三、噪声

添加噪声的相关函数封装在了 `Noise` 类中。

## 3.1 椒盐噪声

椒盐噪声即黑色和白色的像素点。使用 `rand()` 函数随机选取像素点，并根据传入的噪声像素点数量 `num_noise` 分别添加黑点和白点。

```
1 void Noise::addPepperNoise(const cv::Mat & src, cv::Mat & dst, int
2 num_noise)
3 {
4     dst = src.clone();
5
6     // add black and white dot
7     for (int i = 0; i < num_noise; i++)
8     {
9         int x = rand() % src.rows;
10        int y = rand() % src.cols;
11        if (i % 2 == 0)      // half for black
12            for (int j = 0; j < 3; j++)
13                dst.at<cv::Vec3b>(x, y)[j] = 0;
14        else                  // half for white
15            for (int j = 0; j < 3; j++)
16                dst.at<cv::Vec3b>(x, y)[j] = 255;
17    }
18 }
```



## 3.2 高斯噪声

高斯分布的概率密度函数为  $p(z) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(z-\bar{z})^2/2\sigma^2}$ ，这里用 *Box – Muller* 变换来构建服从高斯分布的随机变量。具体描述为：选取两个服从  $[0, 1]$  上均匀分布的随机变量  $U_1$ 、 $U_2$ ，则有

$$X = \cos(2\pi U_1) \sqrt{-2 \ln U_2}$$
$$Y = \sin(2\pi U_1) \sqrt{-2 \ln U_2}$$

$X$ 、 $Y$ 服从均值为0、方差为1的高斯分布。具体的数学证明就略过了。

代码如下，首先根据上述公式生成服从指定均值和方差的高斯分布序列，然后对每个像素点添加高斯噪声。

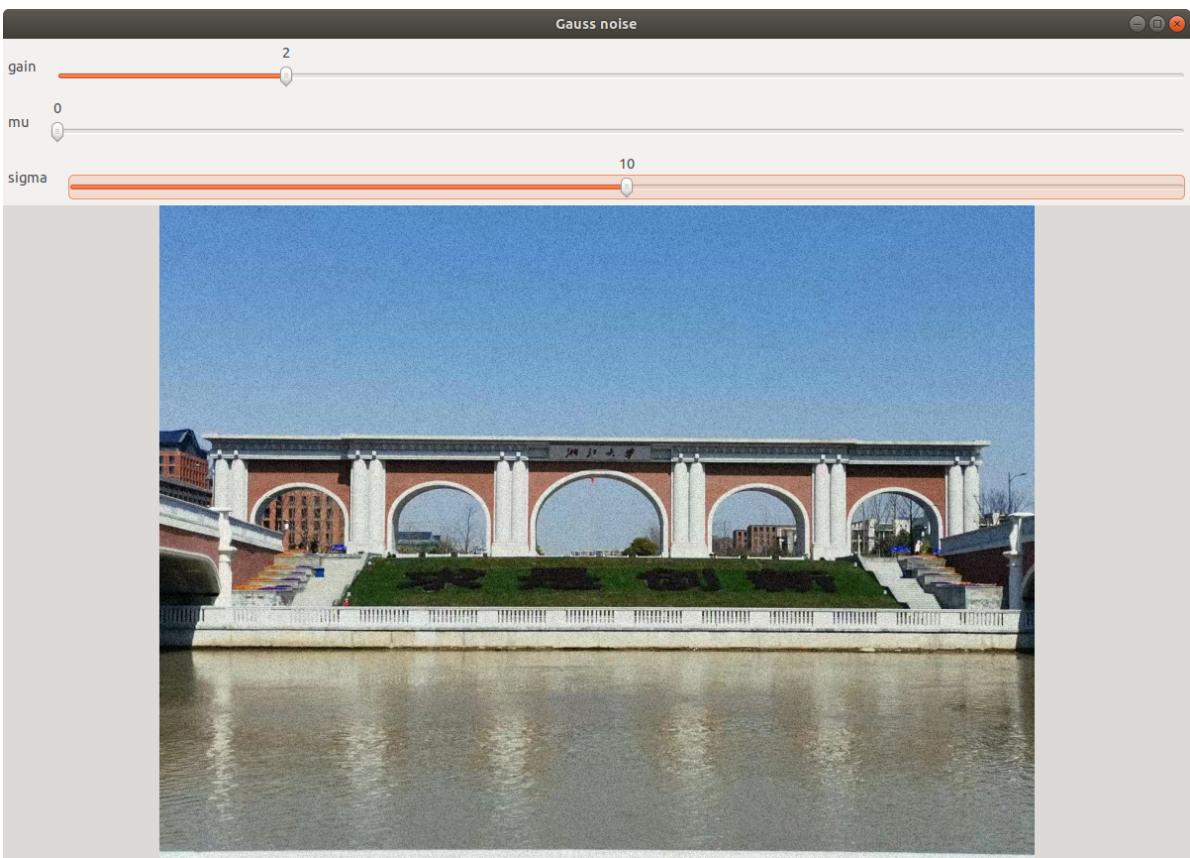
```
1 double Noise::generateGaussSeq(double mu, double sigma) {
2     // Box-Muller transform
3     double u1, u2, std_norm_result;
4     do {
5         u1 = (double) rand() / RAND_MAX;
6         u2 = (double) rand() / RAND_MAX;
7         std_norm_result = cos(2 * CV_PI * u1) * sqrt(-2.0 * log(u2));
8     } while (std_norm_result > 1.0 || std_norm_result == 0.0);
9     return mu + sigma * std_norm_result;
10 }
11
12 void Noise::addGaussNoise(const cv::Mat & src, cv::Mat & dst, double gain,
13 double mu, double sigma)
14 {
15     dst = src.clone();
16
17     int pixel_value;
18     for (int i = 0; i < dst.rows; i++)
19         for (int j = 0; j < dst.cols; j++)
20         {
21             pixel_value = gain * generateGaussSeq(mu, sigma);
22             for (int channel = 0; channel < 3; channel++)
23             {
24                 // limit range to [0, 255]
25                 if (dst.at<cv::Vec3b>(i, j)[channel] + pixel_value > 255)
26                     dst.at<cv::Vec3b>(i, j)[channel] = 255;
27                 else if (dst.at<cv::Vec3b>(i, j)[channel] + pixel_value < 0)
28                     dst.at<cv::Vec3b>(i, j)[channel] = 0;
29                 else
30                     dst.at<cv::Vec3b>(i, j)[channel] += pixel_value;
31             }
32         }
33 }
```

这里先前遇到了一个问题，生成的高斯噪声图片会产生彩色斑点（左下角和右下角），观察发现斑点全部都产生在原本的黑色或白色区域，且方差越大产生越多。经过研究，是因为生成的高斯噪声加入原图像后数值上超出了  $[0, 255]$  的范围，导致某一通道的像素值取摸赋值后产生两极突变，从而形成彩色斑点，因此在代码中加入了限制像素值上下限。

**修改前**



修改后



## 四、滤波

滤波的相关函数封装在了 `Filter` 类中。

### 4.1 均值滤波

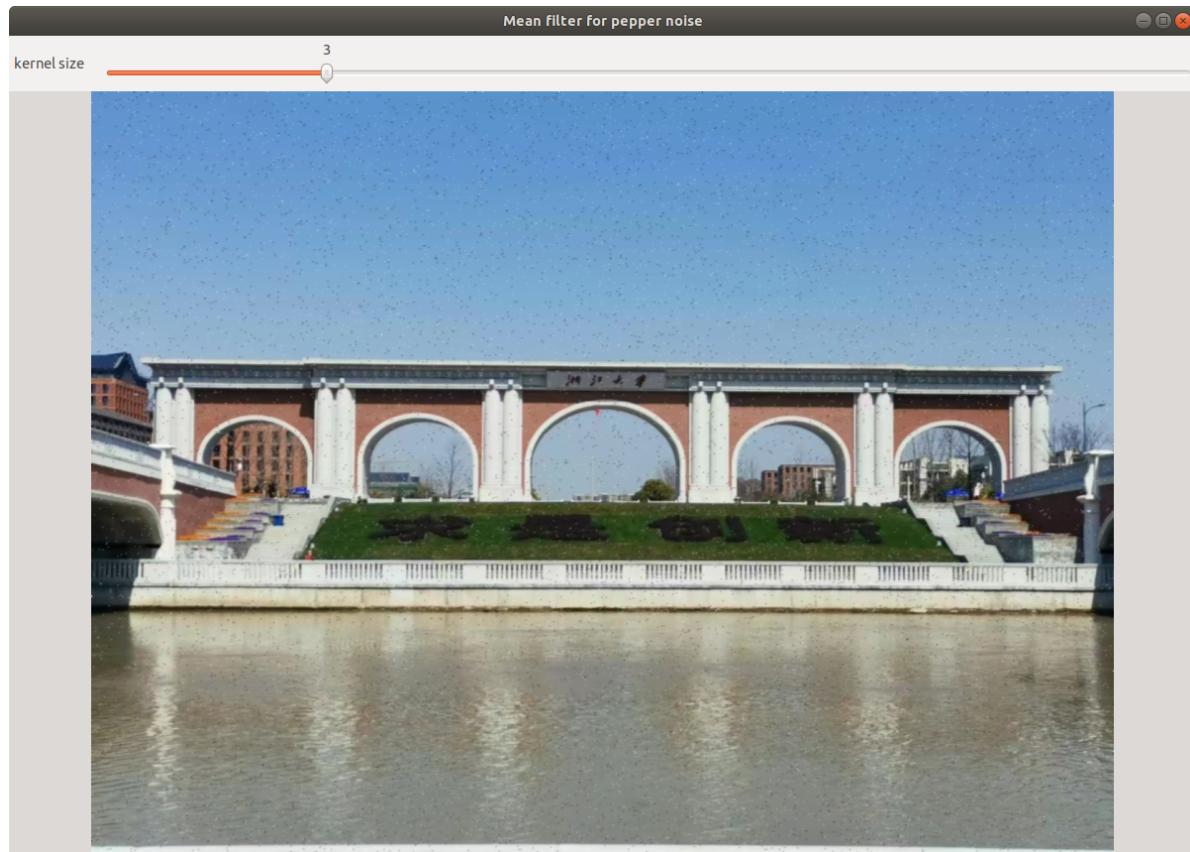
均值滤波里面的种类也有很多，比如算术均值、几何均值、谐波均值、逆谐波均值，这里就用最常规的算术均值滤波，公式如下：

$$f(x, y) = \frac{1}{mn} \sum_{(s,t) \in S_{x,y}} g(s, t)$$

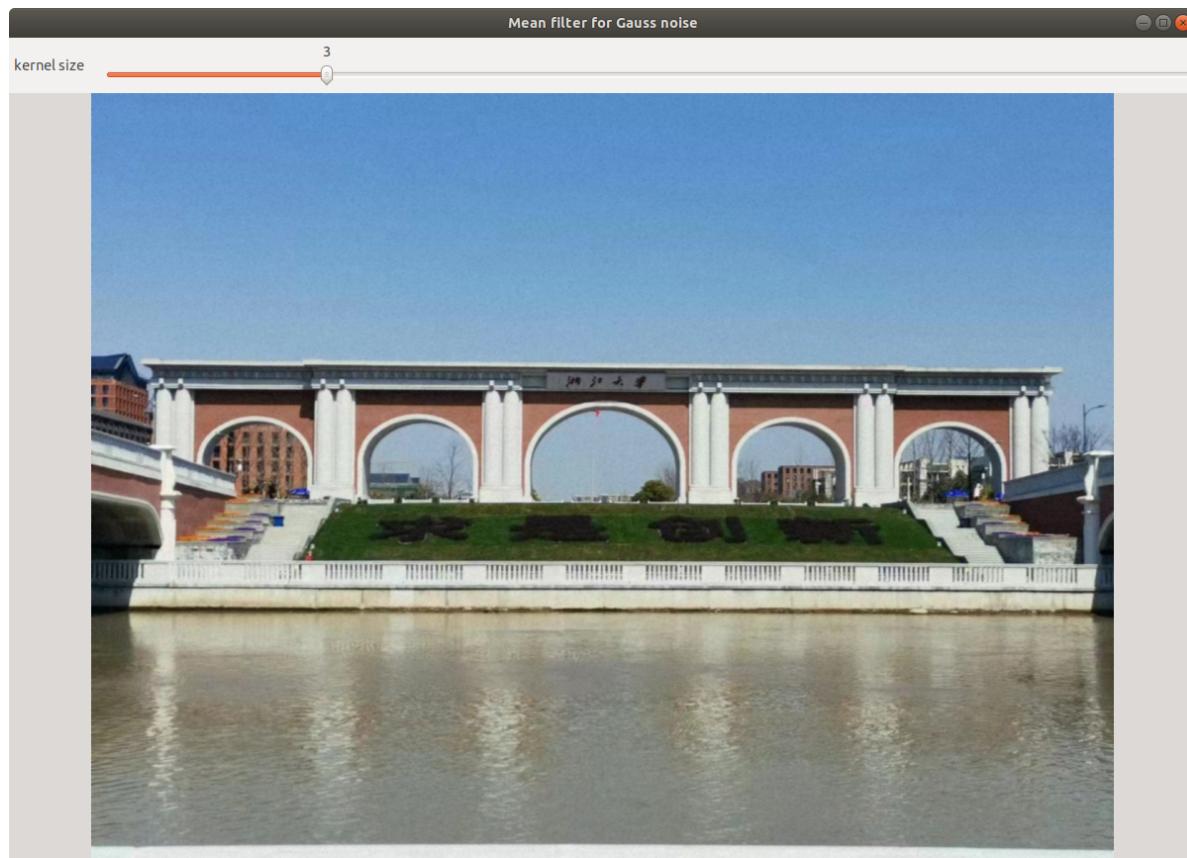
首先计算出待求像素到核边缘的距离 `center2edge`，从而忽略边缘点的处理。为了代码的简洁考虑，在为 `kernel` 区域的像素值求和时，使用了 `ROI` 区域+`cv::sum()` 的方法，可能时间复杂度会增加，但实际运行时没有影响。

```
1 void Filter::meanFilter(const cv::Mat & src, cv::Mat & dst, int kernel_size)
2 {
3     dst = src.clone();
4     cv::Mat ROI;
5
6     if (kernel_size % 2 == 0)
7         kernel_size += 1;
8
9     // ignore edge pixel
10    if (kernel_size > 1)
11    {
12        int c2e = (int) (kernel_size / 2); // center2edge
13        for (int i = c2e; i < dst.rows - c2e; i++)
14            for (int j = c2e; j < dst.cols - c2e; j++)
15                for (int k = 0; k < 3; k++)
16                {
17                    // choose kernel area
18                    ROI = dst(cv::Range(i - c2e, i + c2e + 1), cv::Range(j -
c2e, j + c2e + 1));
19                    // substitute for original pixel
20                    dst.at<cv::Vec3b>(i, j)[k] = (int) cv::sum(ROI).val[k] /
pow(kernel_size, 2);
21                }
22    }
23 }
```

### 椒盐噪声均值滤波



### 高斯噪声均值滤波



由于椒盐噪声的噪点像素值是0或255两个极值，因此使用均值滤波时，核半径越大黑白噪声越小，但是整体图片也越模糊。对高斯噪声而言，均值滤波的效果要比椒盐噪声好，因为添加高斯噪声时的均值为零，因此使用均值滤波后起到了复原的作用。

总体来看，均值滤波主要降低了图像的尖锐变化，去除了图像中的不相关细节，然而由于图像的边缘也是由图像灰度的尖锐变化带来的特性，所以均值滤波处理还是存在着边缘模糊的负面效应。

## 4.2 中值滤波

中值滤波就是把区域中的中值作为新的像素值，这部分代码略长，因为 `opencv` 中的排序函数 `cv::sort()` 只能针对行或列，不能同时针对整片区域，所以实际操作时先根据 `kernel` 取出 `ROI` 区域，`flatten` 之后再交给 `cv::sort` 处理，但是有三点需要注意：

1. `cv::sort()` 只能针对单通道，所以需要先拆分三通道。
2. `reshape()` 只能针对连续值矩阵，而 `ROI` 恰好不是连续的，所以需要 `clone()` 出新的连续矩阵。
3. `cv::Mat` 变量的定义要放在最外层循环外，否则会导致运行效率非常低。

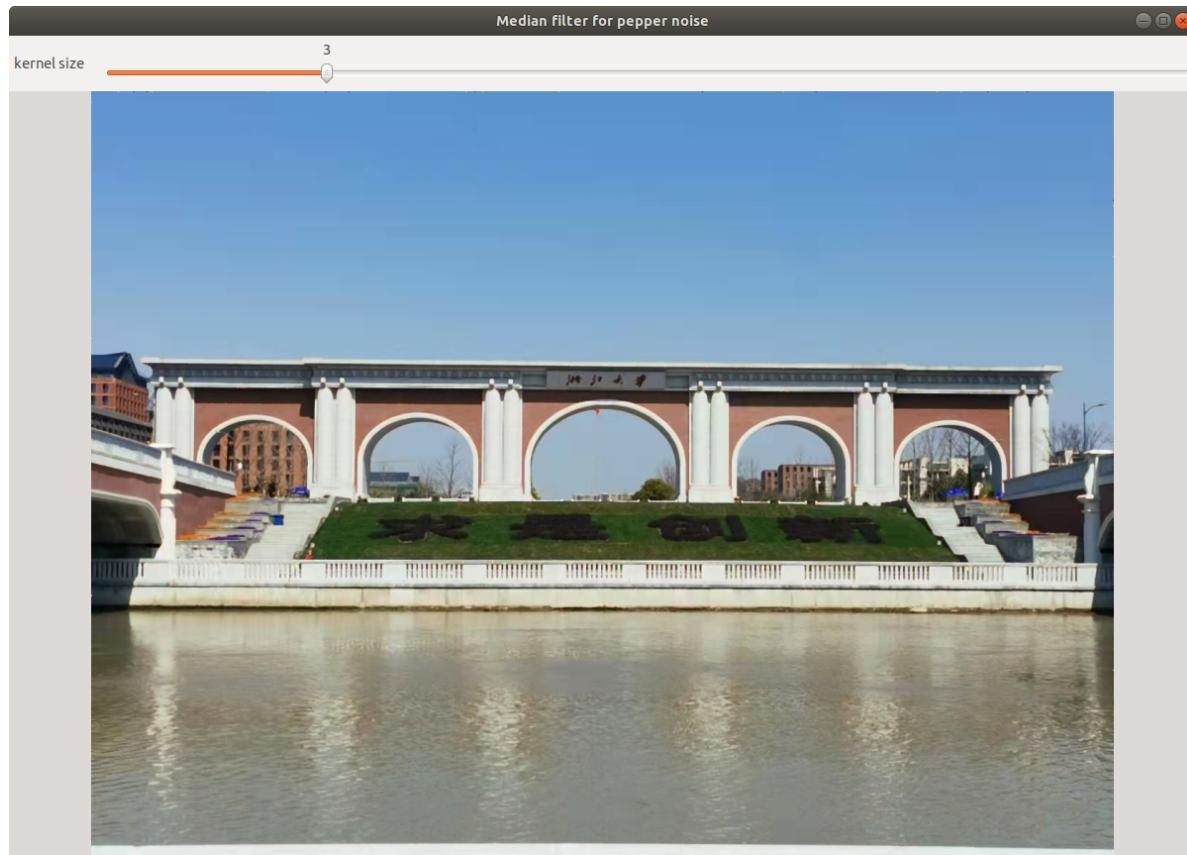
```
1 void Filter::medianFilter(const cv::Mat & src, cv::Mat & dst, int
2 kernel_size)
3 {
4     dst = src.clone();
5     cv::Mat flattened, sorted, ROI, channels[3];
6
7     // split channels
8     cv::split(dst, channels);
9     if (kernel_size % 2 == 0)
10        kernel_size += 1;
11
12     // ignore edge pixel
13     if (kernel_size > 1)
14     {
15         int c2e = (int) (kernel_size / 2); // center2edge
```

```

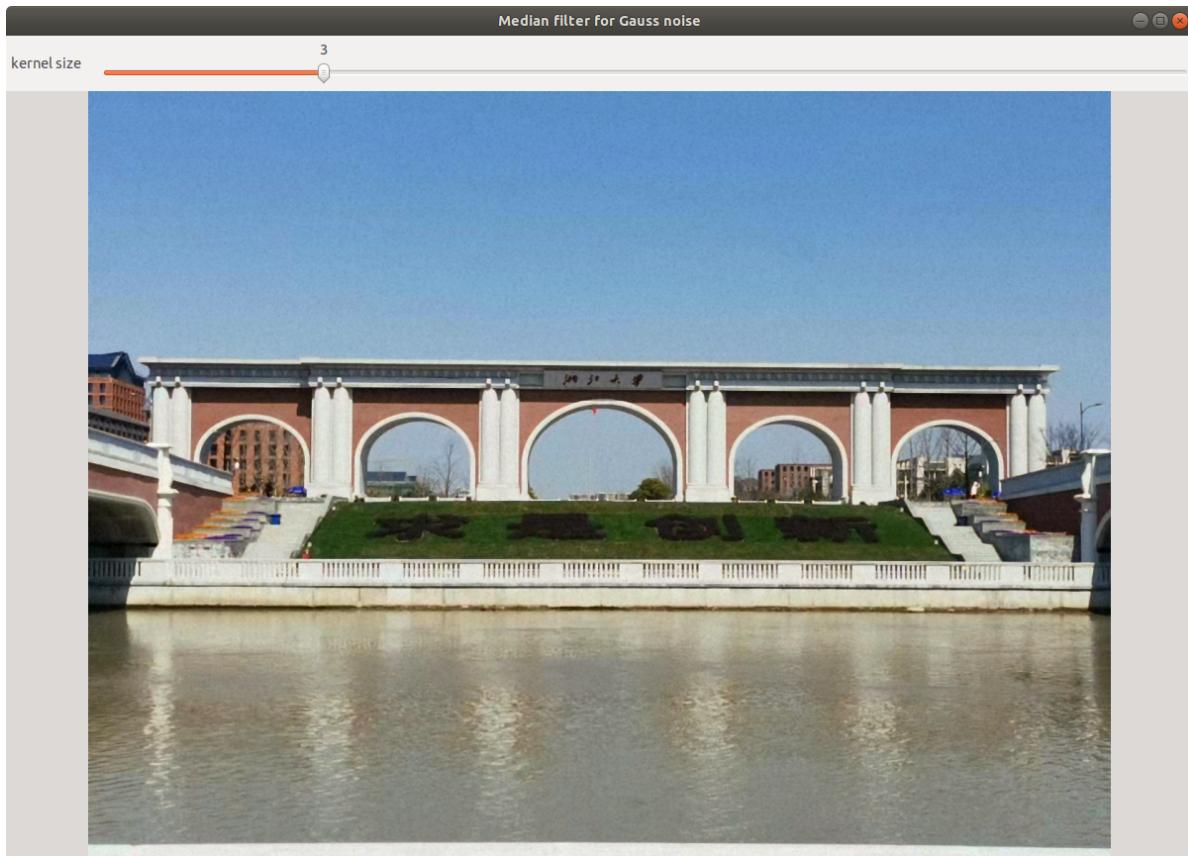
15     for (int i = c2e; i < dst.rows - c2e; i++)
16         for (int j = c2e; j < dst.cols - c2e; j++)
17             for (int ch = 0; ch < 3; ch++)
18             {
19                 // choose kernel area
20                 ROI = channels[ch](cv::Range(i - c2e, i + c2e + 1),
21                                     cv::Range(j - c2e, j + c2e + 1));
22                 // ROI region is not continuous, but 'reshape' requires
23                 // continuous matrix
24                 if (!ROI.isContinuous())
25                     ROI = ROI.clone();
26                 // flatten and sort
27                 flattened = ROI.reshape(1, 1);
28                 cv::sort(flattened, sorted, cv::SORT_EVERY_ROW +
29                         cv::SORT_ASCENDING);
30                 // substitute for original pixel
31                 dst.at<cv::Vec3b>(i, j)[ch] = sorted.at<cv::Vec3b>(0,
c2e)[ch];
32             }
33         }
34     }

```

### 椒盐噪声中值滤波



### 高斯噪声中值滤波



中值滤波对椒盐噪声的效果极好，这一结论很容易从原理得出，由于代码没有处理边缘点，因此除了核半径的边缘图像，其余椒盐噪声被全部滤除。而对高斯噪声来说中值滤波的效果就有限。

总体来看，中值滤波对于滤除脉冲干扰及图像扫描噪声最为有效，不过中值滤波的窗口形状和尺寸对滤波效果影响很大，如果椒盐噪声的数量很多，就需要加大中值滤波的核半径。

### 4.3 双边滤波

双边滤波是是非线性滤波中的一种，其权重对高斯滤波的计算方法进行了优化，为**空间临近度计算的权值和像素值相似度计算的权值**的乘积，优化后的权值再与图像作卷积运算。因此在滤波时，该滤波方法同时考虑空间临近信息与颜色相似信息，在滤除噪声、平滑图像的同时，又做到边缘保存。

每个像素点的计算公式为：

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

其中， $g(i, j)$  代表输出点像素； $S(i, j)$  代表以  $(i, j)$  为中心的邻域； $f(k, l)$  代表输入点像素； $w(i, j, k, l)$  代表权重，而权重又取决于定义域核和值域核。

定义域核：

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

值域核：

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

权重：

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

在编程时严格对应了以上符号。同时，定义域核的权重仅与核半径相关，与邻域像素值无关，因此可以提前计算，提高运行效率。

```

1 void Filter::bilateralFilter(const cv::Mat & src, cv::Mat & dst, int
2 kernel_size, double sigma_domain, double sigma_range)
3 {
4     using namespace std;
5     dst = src.clone();
6
7     double weight_domain[kernel_size][kernel_size],
8     weight_range[kernel_size][kernel_size];
9     if (kernel_size % 2 == 0)
10         kernel_size += 1;
11     // center pixel point index 'i' & 'j'
12     int i = (int) kernel_size / 2, j = i;
13     // surrounding pixel point index 'k' & 'l'
14     int k, l;
15
16     // calculate domain kernel, only up to kernel size
17     for (k = 0; k < kernel_size; k++)
18         for (l = 0; l < kernel_size; l++)
19             weight_domain[k][l] = exp(-(pow(i-k, 2) + pow(j-l, 2)) / (2 *
20 pow(sigma_domain, 2)));
21
22     // change pixel value
23     if (kernel_size > 1)
24     {
25         double sum_numerator = 0, sum_denominator = 0;
26         int c2e = (int) (kernel_size / 2); // center2edge
27         for (i = c2e; i < dst.rows - c2e; i++)
28             for (j = c2e; j < dst.cols - c2e; j++)
29             {
30                 for (int ch = 0; ch < 3; ch++)
31                 {
32                     // calculate range kernel
33                     for (k = 0; k < kernel_size; k++)
34                         for (l = 0; l < kernel_size; l++)
35                         {
36                             weight_domain[k][l] =
37                                 exp(-pow(src.at<cv::Vec3b>(i, j)[ch] -
38                                     src.at<cv::Vec3b>(i-c2e+k, j-
39                                     c2e+l)[ch], 2)
40                                         / (2 * pow(sigma_range, 2)));
41                         }
42                     for (k = 0; k < kernel_size; k++)
43                         for (l = 0; l < kernel_size; l++)
44                         {
45                             sum_numerator += src.at<cv::Vec3b>(i-c2e+k, j-
46                             c2e+l)[ch] *
47                                 weight_domain[k][l] * weight_range[k]
48 [l];
49                         sum_denominator += weight_domain[k][l] *
50 weight_range[k][l];
51                     }
52                 }
53             }
54         }
55     }
56 }
```

```
45         dst.at<cv::Vec3b>(i, j)[ch] = sum_numerator /  
46         sum_denominator;  
47     }  
48 }  
49 }  
50 }
```

椒盐噪声双边滤波



高斯噪声双边滤波



双边滤波对椒盐噪声的效果不如中值滤波，但优于均值滤波。而双边滤波本身就是由高斯滤波衍化而来，因此对高斯噪声的效果比前两个滤波器更好。

本质上来说，双边滤波同时考虑了邻域像素间的距离和相似度，相比于普通的低通滤波器，双边滤波可以在保留边缘信息的同时滤除噪声，所以滤波后的效果类似于，区分了像素值差别大的区域，同时模糊了区域内的图像。

## 五、程序架构

### 5.1 滑动条

由于椒盐噪声的像素个数、高斯噪声的方差、滤波的核半径等都是可调变量，因此使用滑动条来增加手动调整的功能。该函数可以在指定窗口上方添加滑杆，并在滑杆改变时调用回调函数，呈现不同参数的图片。

注意：拖动滑杆改变核半径进行滤波时会卡，加载双边滤波的图像时也会卡，请耐心等待。

```
1 int createTrackbar(const String& trackbarname, const String& winname,
2                               int* value, int count,
3                               TrackbarCallback onChange = 0,
4                               void* userdata = 0);
```



### 5.2 避免全局变量

一般添加滑杆后都是通过设置全局变量来供回调函数内部使用，但是本程序如果使用全局变量会非常冗长（代码很丑），查看手册后得知，回调函数的第二个参数是通过创建滑杆的最后一个参数传递的，但这是一个 `void*` 指针，如果只传一个 `cv::Mat*` 指针的话，会由于上层函数中变量作用域的问题，导致传入回调函数中的指针指向不存在的内存空间。因此最后是通过打包成一个类传递的方式实现的，将参数封装在 `Data` 类中，然后内层再提取出来使用。

```
1 void on_Trackbar(int pos, void *);
```

```
1 class Filter {
2 public:
3     class Data
4     {
5         public:
6             cv::Mat src;
7             std::string win_name;
8     };
9 }
```

## 5.3 函数指针

由于不同图片的滑杆需求不一致，但同时又考虑到代码的简洁性，因此使用函数指针的方式对不同图片创建滑杆，主要把类中的回调函数声明为静态函数，否则在主函数中调用时会因为 `this` 指针失去上下文而报错。

```
1 void showImage(cv::Mat & mat,
2                 const std::string & win_name,
3                 cv::Size size,
4                 int wait_key=0,
5                 const std::string & save_path="",
6                 void (*pTrackbar)(cv::Mat & src, const std::string &
7                 win_name)=nullptr);
```

例如，对于添加椒盐噪声，函数指针和回调函数分别如下：

```
1 Noise::Data pepper_data;
2 void Noise::pepperTrackbar(cv::Mat & src, const std::string & win_name)
3 {
4     pepper_data.src = src;
5     pepper_data.win_name = win_name;
6     int max_pos = src.cols * src.rows / 10;
7     int cur_pos = src.cols * src.rows / 100;
8     cv::createTrackbar("pepper number", win_name, nullptr, max_pos,
9                         pepperCallback, (void*)& pepper_data);
10    cv::setTrackbarPos("pepper number", win_name, cur_pos);
11 }
12
13 void Noise::pepperCallback(int pepper_num, void * data)
14 {
15     Data* extracted_data = (Data*) data;
16     cv::Mat dst, src = extracted_data->src;
17     std::string win_name = extracted_data->win_name;
18
19     Noise noise;
20     noise.addPepperNoise(src, dst, pepper_num);
21     cv::imshow(win_name, dst);
22 }
```

## 5.4 不足

程序不够完善的地方是运行效率，尤其是双边滤波，需要十几秒的时间才能运行完，这是因为在计算像素时没有做并行处理，导致时间复杂度与像素个数和卷积核半径呈指数级相关。这也是没有在双边函数输出图像上添加滑杆的原因，会直接卡死。

## 六、总结

本次作业的主要难点并不在于噪声和滤波器的实现，而在于整体的代码框架，如何通过函数指针、回调函数等方式来合理调用不同的滤波器，最终的效果符合预期结论，但是感觉这张图选的不是很好，对双边滤波而言没有展现得特别清晰。

