实验四：图像去噪

学号：SA22225286 姓名：孟寅磊 日期：20221006

实验内容

> 1. 均值滤波
>
>   具体内容：利用OpenCV对灰度图像像素进行操作，分别利用算术均值滤波器、 几何均值滤波器、 谐波和逆谐波均值滤波器进行图像去噪。 模板大小为$5*5$。（注： 请分别为图像添加高斯噪声、 胡椒噪声、 盐噪声和椒盐噪声， 并观察滤波效果）
>
> 2. 中值滤波
>
>   具体内容： 利用 OpenCV 对灰度图像像素进行操作，分别利用 $5*5$ 和 $9*9$尺寸的模板对图像进行中值滤波。（注： 请分别为图像添加胡椒噪声、 盐噪声和椒盐噪声， 并观察滤波效果）
>
> 3. 自适应均值滤波
>
>   具体内容： 利用 OpenCV 对灰度图像像素进行操作， 设计自适应局部降低噪声滤波器去噪算法。 模板大小$7*7$（对比该算法的效果和均值滤波器的效果）
>
> 4. 自适应中值滤波
>
>   具体内容： 利用 OpenCV 对灰度图像像素进行操作，设计自适应中值滤波算法对椒盐图像进行去噪。 模板大小$7*7$（对比中值滤波器的效果）
>
> 5. 彩色图像均值滤波
>
>   具体内容： 利用 OpenCV 对彩色图像 RGB 三个通道的像素进行操作， 利用算术均值滤波器和几何均值滤波器进行彩色图像去噪。 模板大小为$5*5$。

实验完成情况

当一幅图像仅被加性噪声退化时，可用空间滤波的方法来估计$f(x,y)$（即对图像$g(x,y)$去噪）。

1. 添加噪声

噪声分量中的灰度值可视为随机变量，而随机变量可由概率密度函数（PDF）来表征。

高斯噪声

高斯随机变量z的PDF为

$$p(z)=\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(z-\bar{z})^2}{2\sigma^2}} \tag{4.1}$$

式中，z表示灰度，$\bar{z}$是z的均值，σ是z的标准差。添加加性高斯噪声的函数如下

```
void add_gaus_noise(const Mat &src, Mat &dst, double sigma) {
    CV_Assert(src.type() == CV_8UC1 || src.type() == CV_8UC3);
    const int rows = src.rows;
    const int cols = src.cols;
    default_random_engine generator;
    normal_distribution<double> noise {0, 1};
    double gaussian_noise;
    dst.create(src.size(), src.type());
```

```
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            gaussian_noise = noise(generator);
            if (src.channels() == 1) {
                dst.at<uchar>(i, j) =
                saturate_cast<uchar>(src.at<uchar>(i, j) +
                static_cast<int>(gaussian_noise * sigma));
            } else {
                for (int ch = 0; ch < 3; ++ch)
                    dst.at<Vec3b>(i, j)[ch] =
                    saturate_cast<uchar>(src.at<Vec3b>(i, j)[ch] +
                    static_cast<int>(gaussian_noise * sigma));
            }
        }
    }
}
```

椒盐噪声

椒盐噪声的PDF为

$$p(z) = \begin{cases} P_s, & z = 2^k - 1 \\ P_p, & z = 0 \\ 1 - (P_s + P_p), & z = V \end{cases} \tag{4.2}$$

式中，V是区间$0 < V < 2^k - 1$内的任意整数。添加椒盐噪声的函数如下

```
void add_pesa_noise(const Mat &src, Mat &dst,
                    double p_pepr, double p_salt) {
    CV_Assert((src.type() == CV_8UC1 || src.type() == CV_8UC3) &&
              p_pepr >= 0 && p_pepr <= 1 && p_salt >= 0 && p_salt <= 1);
    dst = src.clone();
    const int rows = src.rows;
    const int cols = src.cols;
    int i = 0;
    int j = 0;
    default_random_engine generator;
    uniform_int_distribution<int> random_row(0, rows - 1);
    uniform_int_distribution<int> random_col(0, cols - 1);
    int pepr_polluted = static_cast<int>(p_pepr * rows * cols);
    int salt_polluted = static_cast<int>(p_salt * rows * cols);
    int kp = 0, ks = 0;
    // add noise
    while (kp++ < pepr_polluted) {
        i = random_row(generator);
        j = random_col(generator);
        if (src.channels() == 1) {
            dst.at<uchar>(i, j) = 0;
        } else {
            dst.at<Vec3b>(i, j)[0] = 0;
            dst.at<Vec3b>(i, j)[1] = 0;
            dst.at<Vec3b>(i, j)[2] = 0;
        }
    }
```

```
    while (ks++ < salt_polluted) {
        i = random_row(generator);
        j = random_col(generator);
        if (src.channels() == 1) {
            dst.at<uchar>(i, j) = 255;
        } else {
            dst.at<Vec3b>(i, j)[0] = 255;
            dst.at<Vec3b>(i, j)[1] = 255;
            dst.at<Vec3b>(i, j)[2] = 255;
        }
    }
}
```

2. 均值滤波

算术平均滤波器

算术平均滤波器是最简单的均值滤波器，它在由S_{xy}定义的区域中，计算被污染图像$g(x,y)$的平均值。复原的图像$\hat{f}$在(x,y)处的值使用该区域中像素的算术平均值，即

$$\hat{f}(x,y)=\frac{1}{mn}\sum_{(r,c)\in S_{xy}}g(r,c) \tag{4.3}$$

式中，r和c是邻域中所包含像素的行坐标和列坐标。这一运算可以使用大小为$m\times n$，所有系数都是$\frac{1}{mn}$的一个空间核来实现。均值滤波平滑图像中的局部变化，它会降低图像中的噪声，但会模糊图像。

谐波平均滤波器

谐波平均滤波器由下式给出

$$\hat{f}(x,y)=\frac{mn}{\sum_{(r,c)\in S_{xy}}\frac{1}{g(r,c)}} \tag{4.4}$$

谐波平均滤波器既能处理盐粒噪声，又能处理高斯噪声。但不能处理胡椒噪声。

反谐波平均滤波器

反谐波平均滤波器由下式给出

$$\hat{f}(x,y)=\frac{\sum_{(r,c)\in S_{xy}}g(r,c)^{Q+1}}{\sum_{(r,c)\in S_{xy}}g(r,c)^{Q}} \tag{4.5}$$

Q称为滤波器阶数。这种滤波器适用于消除椒盐噪声。$Q>0$时，该滤波器消除胡椒噪声；$Q<0$时，该滤波器消除盐粒噪声。但不能同时消除这两种噪声。当$Q=0$时，该滤波器简化为算术平均滤波器；$Q=-1$时，该滤波器简化为谐波平均滤波器。基于(4.5)式的上述三种滤波器实现如下

```
/*
 * Q > 0, eliminates pepper noise
 * Q < 0, eliminates salt noise
 * Q = 0 --> arithmetic mean filter, eliminates noise
 * Q =-1 --> harmonic_mean_filter, eliminates gauss and salt noise
 */
void contraharmonic_mean_filter(const Mat &src, Mat &dst,
                                int m, int n, double Q) {
    Mat filled;
    dst.create(src.size(), src.type());
    int row_border = (m - 1) / 2;
```

```
    int col_border = (n - 1) / 2;
    copyMakeBorder(src, filled, row_border, row_border,
                                col_border, col_border,
                                BORDER_REFLECT);
    int i_dst = 0;
    int j_dst = 0;
    double s1 = 0;
    double s2 = 0;
    for (int i = row_border; i < filled.rows - row_border; ++i) {
        for (int j = col_border; j < filled.cols - col_border; ++j) {
            s1 = s2 = 0;
            i_dst = i - row_border;
            j_dst = j - col_border;
            for (int x = i - row_border; x <= i + row_border; ++x) {
                for (int y = j - col_border; y <= j + col_border; ++y) {
                    s1 += pow(filled.at<uchar>(x, y), Q + 1);
                    s2 += pow(filled.at<uchar>(x, y), Q);
                }
            }
            dst.at<uchar>(i_dst, j_dst) = saturate_cast<uchar>(s1 / s2);
        }
    }
}
```

几何均值滤波器

使用几何均值滤波器复原的图像由下式给出

$$\hat{f}(x,y) = \left[\prod_{(r,c)\in S_{xy}} g(r,c) \right]^{\frac{1}{mn}} \tag{4.6}$$

每个复原的像素是图像区域中所有像素之积的$\frac{1}{mn}$次幂。几何均值滤波器实现的平滑可与算术平均滤波器相比，但损失的图像细节更少。几何均值滤波器的实现如下

```
void geometric_mean_filter(const Mat &src, Mat &dst, int m, int n) {
    Mat filled;
    dst.create(src.size(), src.type());
    int row_border = (m - 1) / 2;
    int col_border = (n - 1) / 2;
    copyMakeBorder(src, filled, row_border, row_border,
                                col_border, col_border,
                                BORDER_REFLECT);
    int i_dst = 0;
    int j_dst = 0;
    double pd = 1;
    const double exponent = 1.0 / m / n;
    for (int i = row_border; i < filled.rows - row_border; ++i) {
        for (int j = col_border; j < filled.cols - col_border; ++j) {
            i_dst = i - row_border;
            j_dst = j - col_border;
            pd = 1;
            for (int x = i - row_border; x <= i + row_border; ++x)
                for (int y = j - col_border; y <= j + col_border; ++y)
                    pd *= filled.at<uchar>(x, y);
```

```
            dst.at<uchar>(i_dst, j_dst) =
            saturate_cast<uchar>(pow(pd, exponent));
        }
    }
}
```

3. 中值滤波

中值滤波器是最著名的统计排序滤波器，它用一个预定义的像素邻域中的灰度中值来替代像素的值，即

$$\hat{f}(x,y) = \underset{(r,c)\in S_{xy}}{median}\{g(r,c)\} \tag{4.7}$$

中值滤波器应用广泛，因为与大小相同的线性平滑滤波器相比，它能有效地降低某些随机噪声，且模糊度要小得多。对于单极和双极冲激噪声，中值滤波器的效果更好。基本的中值滤波器实现如下

```
void median_filter(const Mat &src, Mat &dst, int m, int n) {
    Mat filled;
    dst.create(src.size(), src.type());
    int row_border = (m - 1) / 2;
    int col_border = (n - 1) / 2;
    copyMakeBorder(src, filled, row_border, row_border,
                                col_border, col_border,
                                BORDER_REFLECT);
    int i_dst = 0;
    int j_dst = 0;
    vector<uchar> Sxy;
    int median_index = m * n / 2;
    for (int i = row_border; i < filled.rows - row_border; ++i) {
        for (int j = col_border; j < filled.cols - col_border; ++j) {
            i_dst = i - row_border;
            j_dst = j - col_border;
            Sxy.clear();
            for (int x = i - row_border; x <= i + row_border; ++x)
                for (int y = j - col_border; y <= j + col_border; ++y)
                    Sxy.push_back(filled.at<uchar>(x, y));
            sort(Sxy.begin(), Sxy.end());
            dst.at<uchar>(i_dst, j_dst) = Sxy[median_index];
        }
    }
}
```

4. 自适应均值滤波

自适应滤波器的特性会根据$m \times n$矩形邻域S_{xy}定义的滤波区域内的图像的统计特性变化。自适应局部降噪滤波器具有如下性能：

> 1. 若σ_η^2为0，则滤波器返回(x,y)处的值g。因为噪声为0时，(x,y)处的g等于f。
> 2. 若局部方差$\sigma_{S_{xy}}^2$与σ_η^2高度相关，则滤波器返回(x,y)处的一个接近于g的值。高局部方差通常与边缘相关，且应保留这些边缘。
> 3. 若两个方差相等，则希望滤波器返回S_{xy}中像素的算术平均值。当局部区域的性质与整个图像的性质相同时会出现这个条件，且平均运算会降低局部噪声。

根据这些假设得到的$\hat{f}(x,y)$的自适应表达式可以写为

$$\hat{f}(x,y) = g(x,y) - \frac{\sigma_\eta^2}{\sigma_{S_{xy}}^2}[g(x,y) - \bar{z}_{S_{xy}}] \tag{4.8}$$

σ_η^2由噪声图像估计得到。其他参数由邻域S_{xy}中的像素计算得到。注意当$\sigma_\eta^2 > \sigma_{S_{xy}}^2$时比率应设为1，这样可以阻止因缺少图像噪声方差的知识而产生无意义的结果。基于上述思路实现的自适应均值滤波器实现如下

```
void adaptive_mean_filter(const Mat &src, Mat &dst, int m, int n) {
    int rows = src.rows;
    int cols = src.cols;
    // 计算噪声图像的标准差
    double sum      = 0;
    double mean_src = 0;
    double sdev_src = 0;
    for (int i = 0; i < rows; ++i)
        for (int j = 0; j < cols; ++j)
            sum += src.at<uchar>(i, j);
    mean_src = sum / (rows * cols);
    sum = 0;
    for (int i = 0; i < rows; ++i)
        for (int j = 0; j < cols; ++j)
            sum += pow(1.0 * src.at<uchar>(i, j) - mean_src, 2);
    sdev_src = sqrt(sum / (rows * cols));

    Mat filled;
    dst.create(src.size(), src.type());
    int row_border = (m - 1) / 2;
    int col_border = (n - 1) / 2;
    copyMakeBorder(src, filled, row_border, row_border,
                                col_border, col_border,
                                BORDER_REFLECT);
    int i_dst = 0;
    int j_dst = 0;
    Mat Sxy;
    Mat mean_Sxy;
    Mat sdev_Sxy;
    double k = 0;
    for (int i = row_border; i < filled.rows - row_border; ++i) {
        for (int j = col_border; j < filled.cols - col_border; ++j) {
            i_dst = i - row_border;
            j_dst = j - col_border;
            Sxy = Mat(filled, Rect(j - col_border, i - row_border, n, m));
            meanStdDev(Sxy, mean_Sxy, sdev_Sxy);
            k = (pow(sdev_src, 2)) /
                (pow(sdev_Sxy.at<double>(0, 0), 2) + 1e-6);
            if (k < 1)
                dst.at<uchar>(i_dst, j_dst) =
                saturate_cast<uchar>(filled.at<uchar>(i, j) -
                k * (filled.at<uchar>(i, j) - mean_Sxy.at<double>(0, 0)));
            else
                dst.at<uchar>(i_dst, j_dst) = mean_Sxy.at<double>(0, 0);
        }
    }
}
```

5. 自适应中值滤波

自适应中值滤波能够处理具有更大概率的噪声，且会在试图保留图像细节的同时平滑非冲激噪声。自适应中值滤波器也工作在矩形邻域S_{xy}内，但是它会根据下面列出的一些条件来改变S_{xy}的大小。自适应中值滤波器的工作原理如下

> 层次A：若$z_{min} < z_{med} < z_{max}$，则转到层次B
>
> 否则，增大S_{xy}的尺寸
>
> 若$S_{xy} \leq S_{max}$，则重复层次A
>
> 否则，输出z_{med}
>
> 层次B：若$z_{min} < z_{xy} < z_{max}$，则输出$z_{xy}$
>
> 否则，输出z_{med}

其中，z_{min}是S_{xy}中的最小灰度值；z_{max}是S_{xy}中的最大灰度值；z_{med}是S_{xy}中的灰度值的中值；z_{xy}是坐标(x, y)处的灰度值；S_{max}是S_{xy}的最大允许尺寸。基于上述思路实现的代码如下

```cpp
/// @brief the implementation of an adaptive median filter,
///        the in-place processing is not supported.
/// @param src input image
/// @param dst output image
/// @param init_m the rows of the initial neighborhood S_xy
/// @param init_n the cols of the initial neighborhood S_xy
/// @param max_m the rows of the maximal neighborhood S_max
/// @param max_n the cols of the maximal neighborhood S_max
void adaptive_median_filter(const Mat &src, Mat &dst,
                            int init_m, int init_n, int max_m, int max_n) {
    CV_Assert(src.type() == CV_8UC1 && init_m <= max_m && init_n <= max_n);
    dst.create(src.size(), CV_8UC1);
    int max_row_border = (max_m - 1) / 2;
    int max_col_border = (max_n - 1) / 2;
    Mat filled;
    // 按照S_xy的最大允许尺寸S_max进行边界填充，filled为填充结果，填充方式为镜像填充
    copyMakeBorder(src, filled, max_row_border, max_row_border,
                                max_col_border, max_col_border,
                                BORDER_REFLECT);
    int x, y;
    int r_bias, c_bias;
    int curr_m = init_m;
    int curr_n = init_n;
    vector<uchar> S_xy;
    uchar z_xy, z_min, z_med, z_max;
    bool is_Sxy_greater_than_Smax = false;

    for (int i = 0; i < dst.rows; ++i) {
        for (int j = 0; j < dst.cols; ++j) {
            // 计算z_xy
            x = i + max_row_border; // dst图像中的像素在filled中所处的行
            y = j + max_col_border; // dst图像中的像素在filled中所处的列
            z_xy = filled.at<uchar>(x, y);

            // 计算z_min、z_med和z_max
            S_xy.clear();
```

```
            r_bias = (curr_m - 1) / 2;
            c_bias = (curr_n - 1) / 2;
            for (int si = x - r_bias; si <= x + r_bias; ++si)
                for (int sj = y - c_bias; sj <= y + c_bias; ++sj)
                    S_xy.push_back(filled.at<uchar>(si, sj));
            sort(S_xy.begin(), S_xy.end());
            z_min = S_xy[0];
            z_med = S_xy[curr_m * curr_n / 2];
            z_max = S_xy[curr_m * curr_n - 1];

            // 当不满足z_min < z_med < z_max时，扩大S_xy
            is_Sxy_greater_than_Smax = false;
            while (!(z_min < z_med && z_med < z_max)) {
                curr_m += 2;
                curr_n += 2;
                // 如果S_xy > S_max，转移到Sxy_greater_than_Smax语句
                if (curr_m > max_m || curr_n > max_n) {
                    is_Sxy_greater_than_Smax = true;
                    goto Sxy_greater_than_Smax;
                }
                // 扩大S_xy，即将当前S_xy中外边一圈的像素包括进来
                ++r_bias;
                ++c_bias;
                for (int c = y - c_bias; c <= y + c_bias; ++c)
                    S_xy.push_back(filled.at<uchar>(x - r_bias, c));
                for (int c = y - c_bias; c <= y + c_bias; ++c)
                    S_xy.push_back(filled.at<uchar>(x + r_bias, c));
                for (int r = x - r_bias + 1; r <= x + r_bias - 1; ++r)
                    S_xy.push_back(filled.at<uchar>(r, y - c_bias));
                for (int r = x - r_bias + 1; r <= x + r_bias - 1; ++r)
                    S_xy.push_back(filled.at<uchar>(r, y + c_bias));
                // 计算新的z_min、z_med和z_max
                sort(S_xy.begin(), S_xy.end());
                z_min = S_xy[0];
                z_med = S_xy[curr_m * curr_n / 2];
                z_max = S_xy[curr_m * curr_n - 1];
            }
            // S_xy > S_max时输出z_med
            Sxy_greater_than_Smax:
            dst.at<uchar>(i, j) = z_med;
            // 满足z_min < z_med < z_max且S_xy <= S_max时输出z_xy或z_med
            if (!is_Sxy_greater_than_Smax)
                dst.at<uchar>(i, j) =
                (z_min < z_xy && z_xy < z_max) ? z_xy : z_med;
        }
    }
}
```

6. 彩色图像均值滤波

基于前边均值滤波器的原理，对彩色图像的RGB三个通道操作实现滤波。

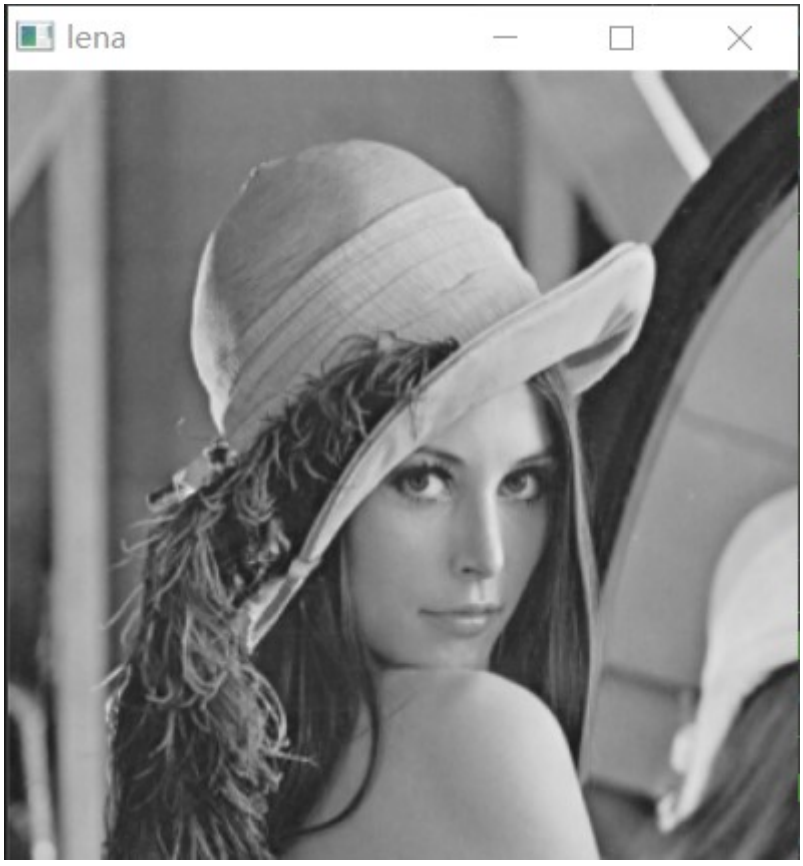
```
void rgb_arithmetic_mean_filter(const Mat &src, Mat &dst, int m, int n) {
    Mat filled;
    dst.create(src.size(), src.type());
```

```
    int row_border = (m - 1) / 2;
    int col_border = (n - 1) / 2;
    copyMakeBorder(src, filled, row_border, row_border,
                                col_border, col_border,
                                BORDER_REFLECT);
    int i_dst = 0;
    int j_dst = 0;
    vector<double> sum(3, 0);
    for (int i = row_border; i < filled.rows - row_border; ++i) {
        for (int j = col_border; j < filled.cols - col_border; ++j) {
            i_dst = i - row_border;
            j_dst = j - col_border;
            sum[0] = sum[1] = sum[2] = 0;
            for (int x = i - row_border; x <= i + row_border; ++x)
                for (int y = j - col_border; y <= j + col_border; ++y)
                    for (int ch = 0; ch < 3; ++ch)
                        sum[ch] += filled.at<Vec3b>(x, y)[ch];
            for (int ch = 0; ch < 3; ++ch)
                dst.at<Vec3b>(i_dst, j_dst)[ch] =
                saturate_cast<uchar>(sum[ch] / (m * n));
        }
    }
}

void rgb_geometric_mean_filter(const Mat &src, Mat &dst, int m, int n) {
    Mat filled;
    dst.create(src.size(), src.type());
    int row_border = (m - 1) / 2;
    int col_border = (n - 1) / 2;
    copyMakeBorder(src, filled, row_border, row_border,
                                col_border, col_border,
                                BORDER_REFLECT);
    int i_dst = 0;
    int j_dst = 0;
    vector<double> pd(3, 1);
    const double exponent = 1.0 / m / n;
    for (int i = row_border; i < filled.rows - row_border; ++i) {
        for (int j = col_border; j < filled.cols - col_border; ++j) {
            i_dst = i - row_border;
            j_dst = j - col_border;
            pd[0] = pd[1] = pd[2] = 1;
            for (int x = i - row_border; x <= i + row_border; ++x)
                for (int y = j - col_border; y <= j + col_border; ++y)
                    for (int ch = 0; ch < 3; ++ch)
                        pd[ch] *= filled.at<Vec3b>(x, y)[ch];
            for (int ch = 0; ch < 3; ++ch)
                dst.at<Vec3b>(i_dst, j_dst)[ch] =
                saturate_cast<uchar>(pow(pd[ch], exponent));
        }
    }
}
```

实验结果

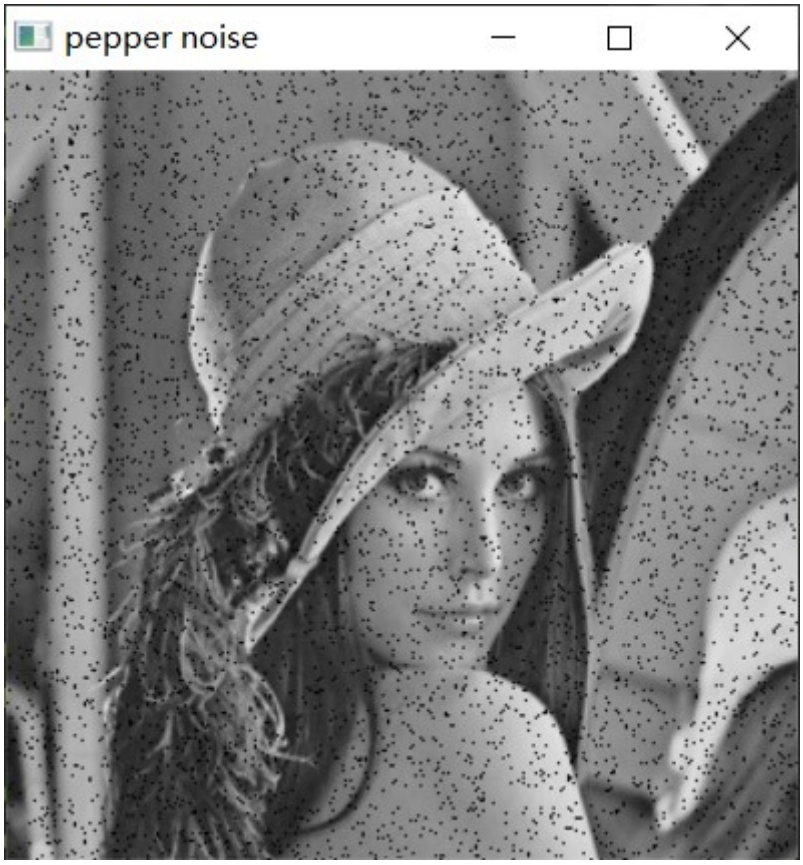
> 用于实验的灰度图像



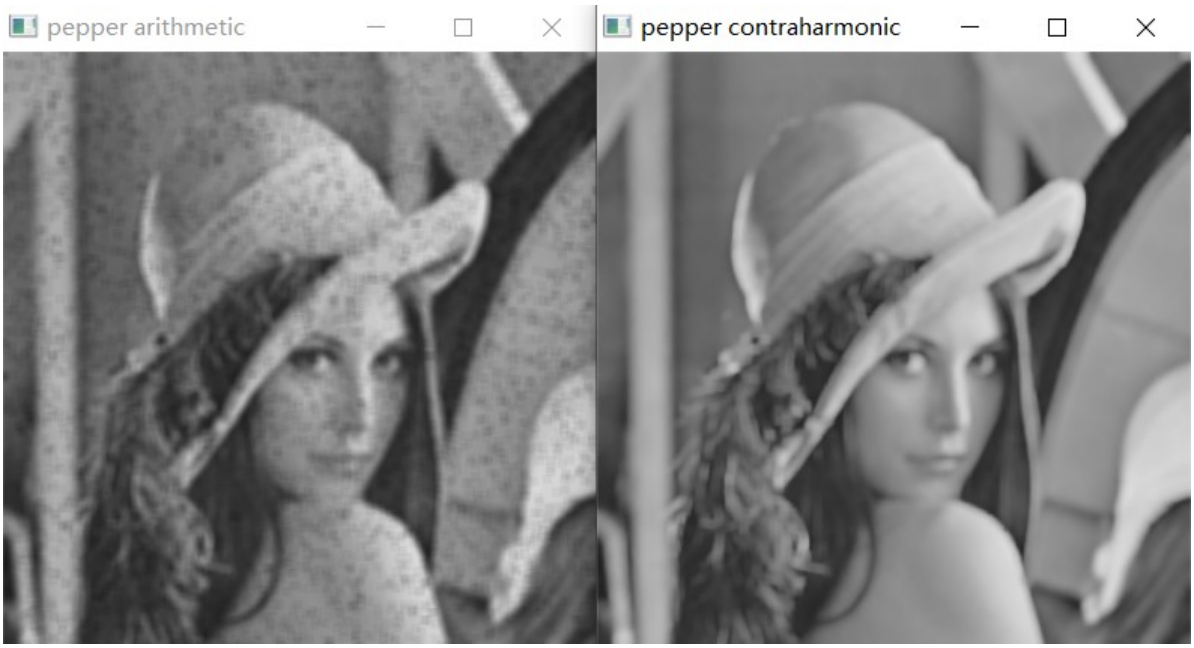
- [x] 均值滤波

均值滤波使用的模板大小均为5×5。

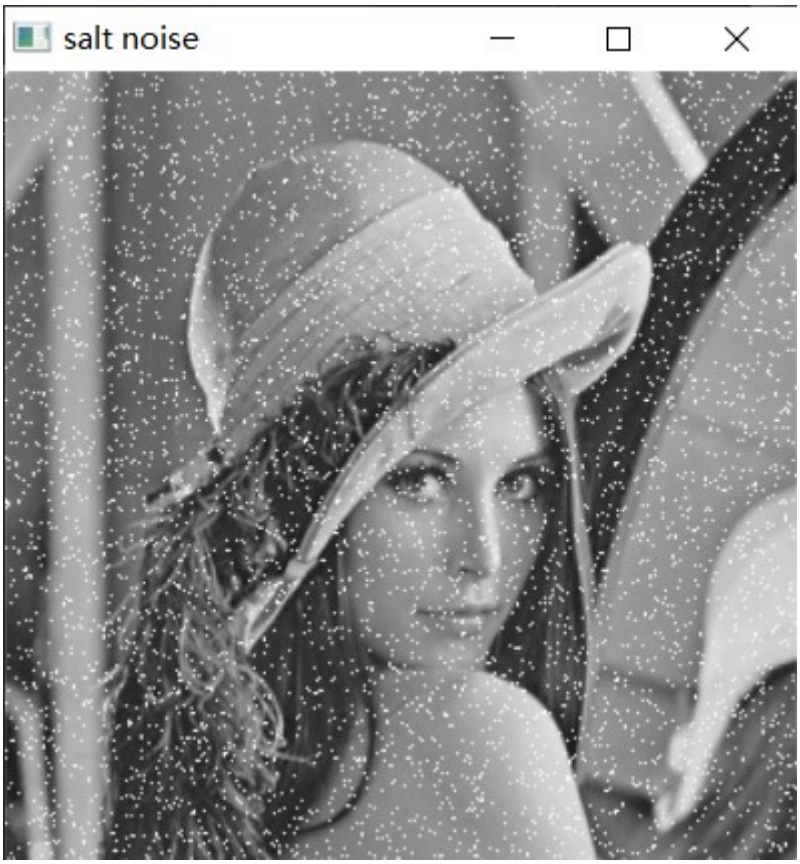
> 被$P_{pepper} = 0.05$的胡椒噪声污染的图像



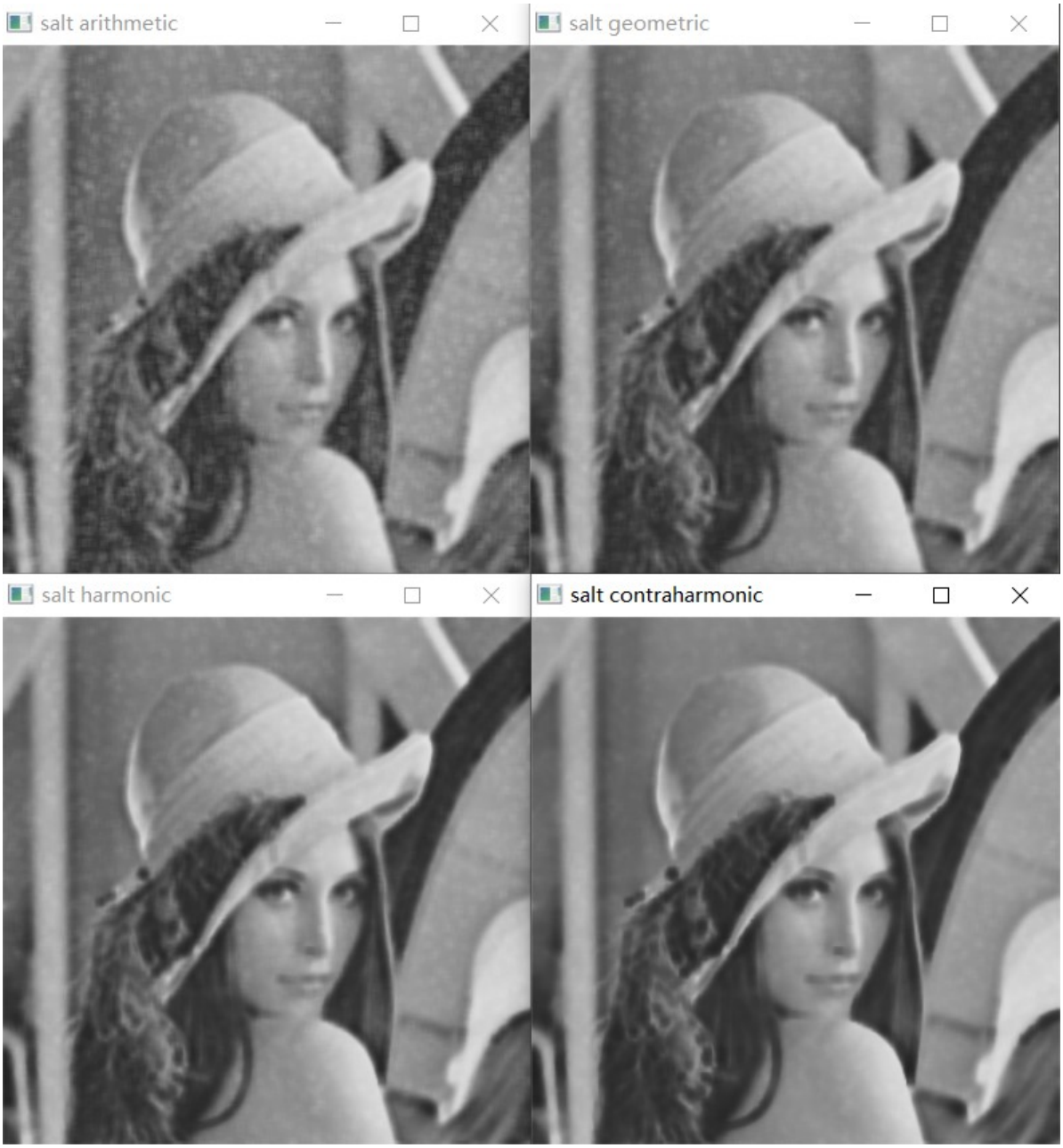
> 左边的图像是算术平均滤波的结果，右边的图像是反谐波平均滤波的结果



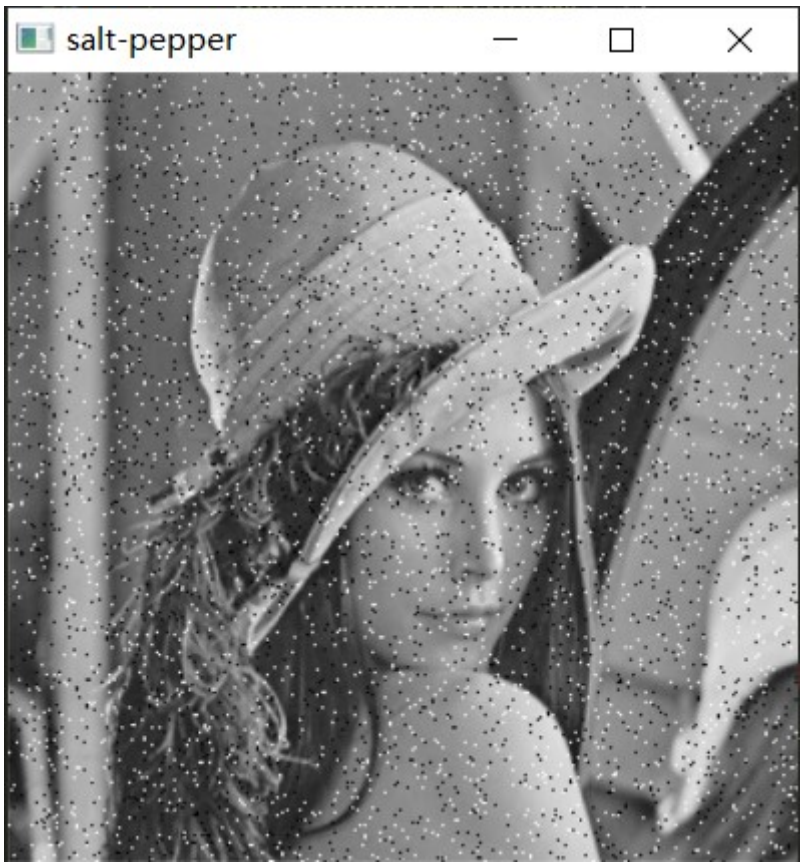
> 被$P_{salt} = 0.05$的盐噪声污染的图像



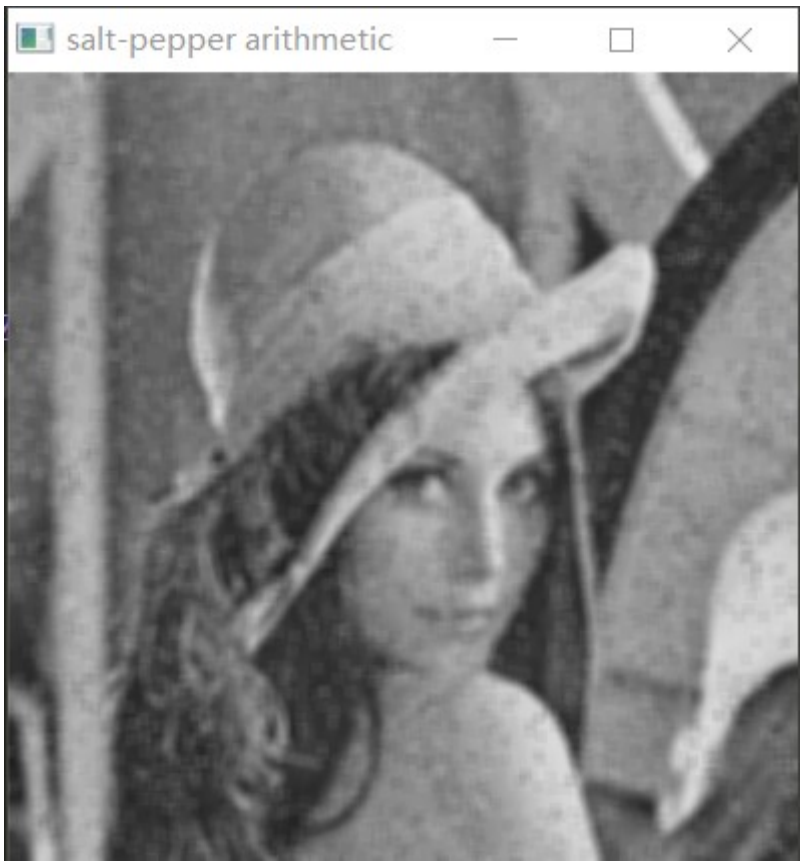
> 左上图像是算术平均滤波的结果；右上图像是几何平均滤波的结果；左下图像是谐波平均滤波的结果；左下图像是反谐波平均滤波的结果



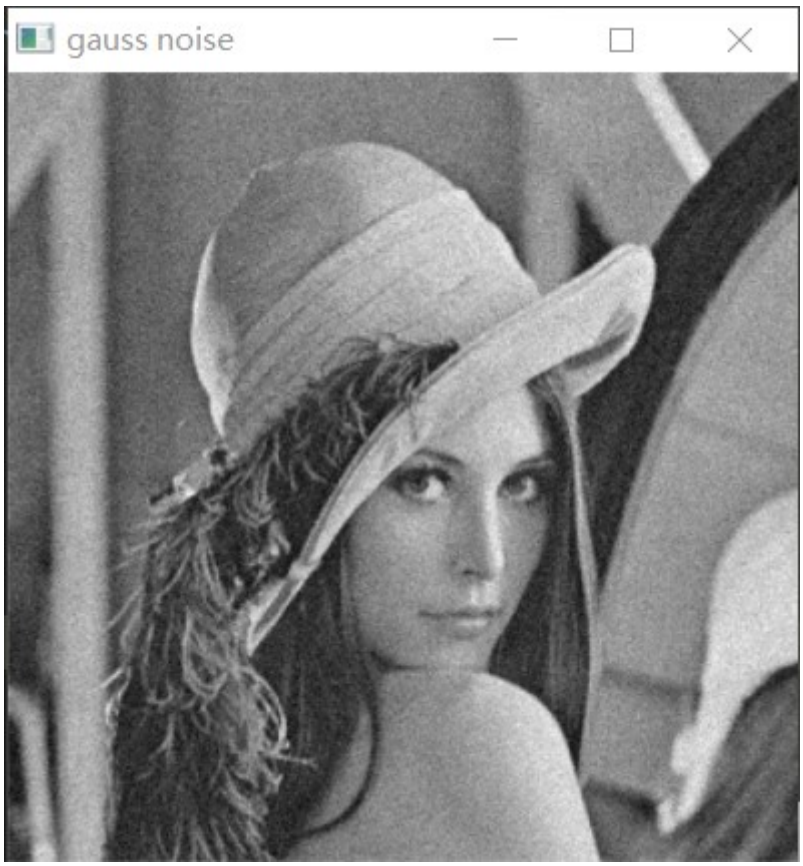
被$P_{pepper}=0.025, P_{salt}=0.025$的椒盐噪声污染的图像



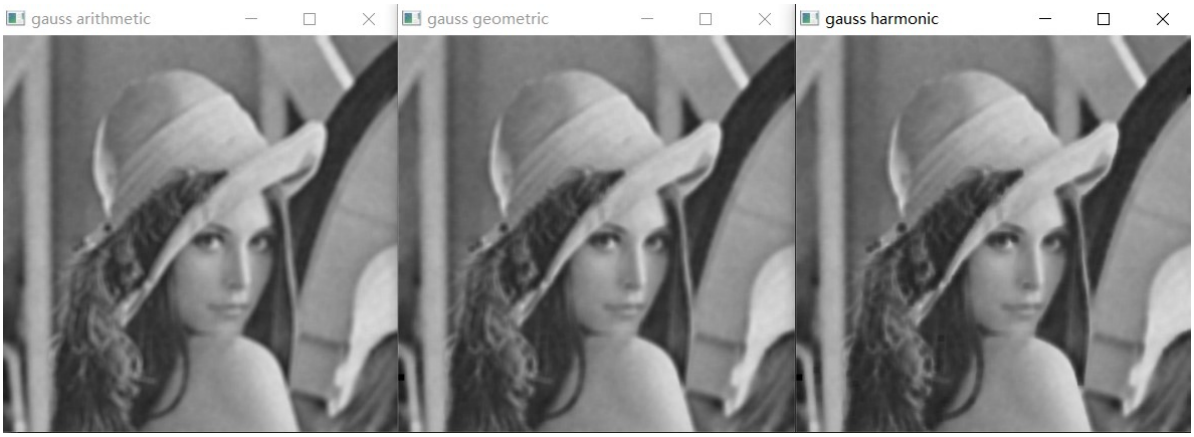
> 使用算术平均滤波的结果



> 被均值为0，方差为100的加性高斯噪声污染的图像

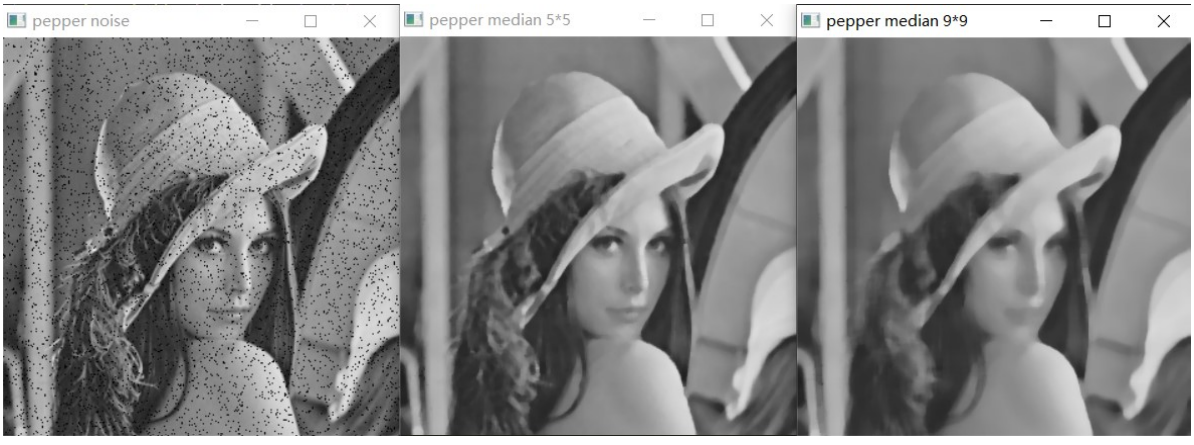


> 左图是算术平均滤波的结果；中间图像是几何平均滤波的结果；右图是谐波平均滤波的结果

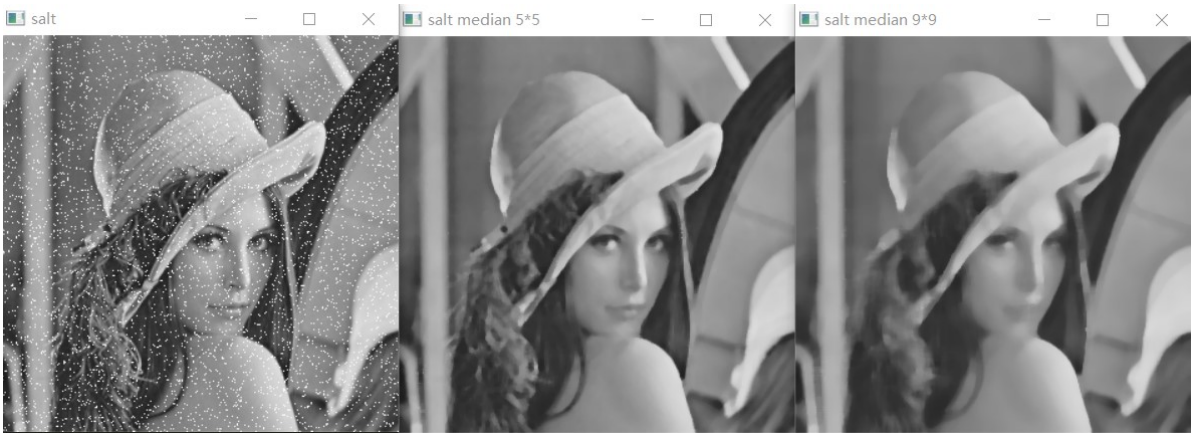


- [x] 中值滤波

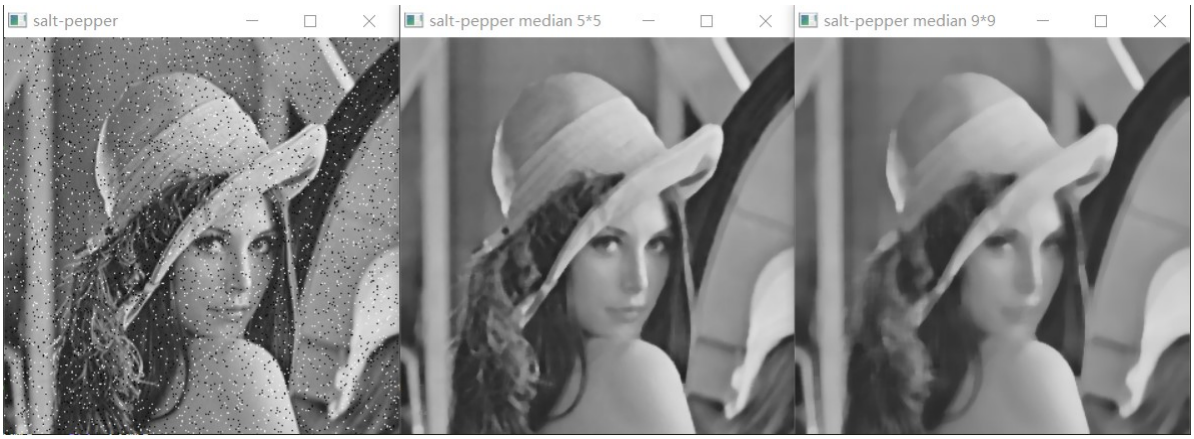
> 左图是被$P_{pepper} = 0.05$的胡椒噪声污染的图像；中间是用5×5尺寸的中值滤波器滤波的结果；右图是用9×9尺寸的中值滤波器滤波的结果



> 左图是被$P_{salt} = 0.05$的盐噪声污染的图像；中间是用5×5尺寸的中值滤波器滤波的结果；右图是用9×9尺寸的中值滤波器滤波的结果

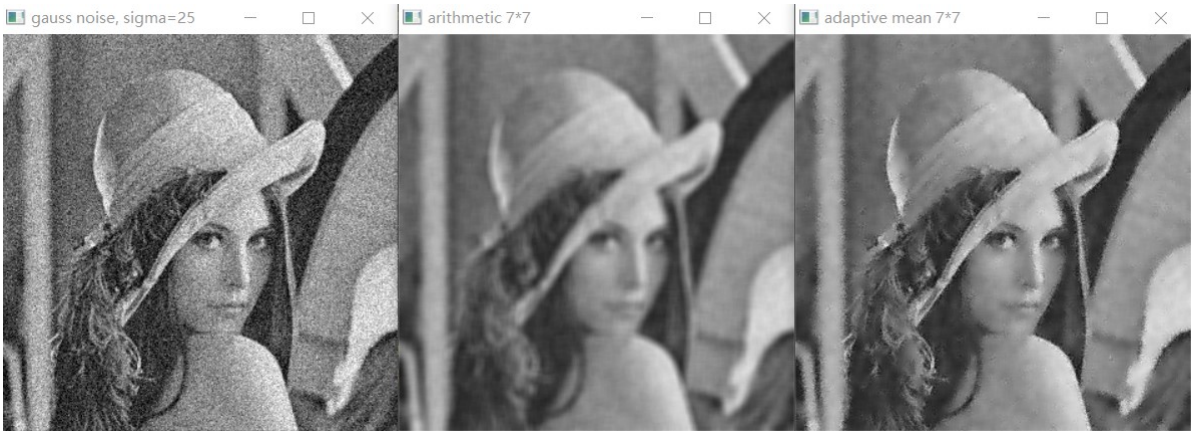


> 左图是被$P_{pepper} = 0.025, P_{salt} = 0.025$的椒盐噪声污染的图像；中间是用$5 \times 5$尺寸的中值滤波器滤波的结果；右图是用$9 \times 9$尺寸的中值滤波器滤波的结果



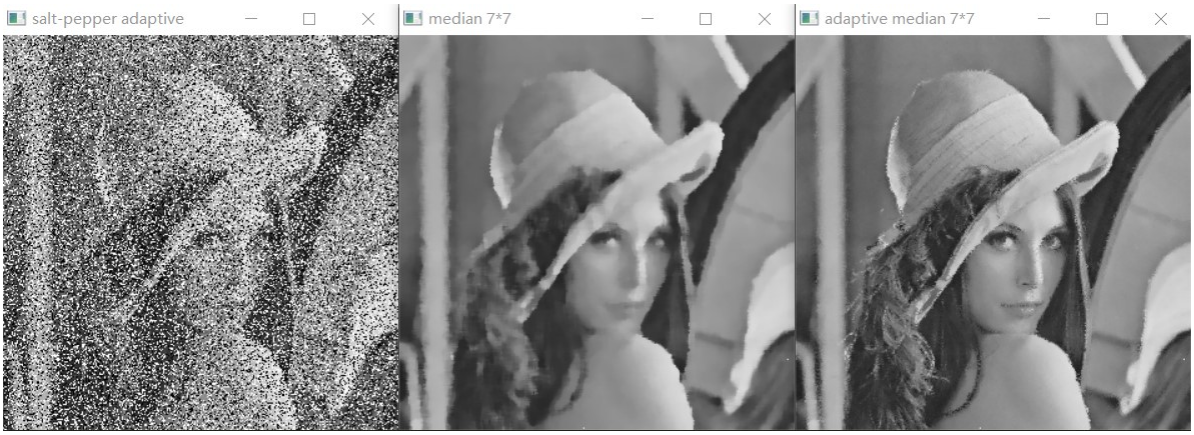
- [x] 自适应均值滤波

> 左图是被均值为0，方差为625的加性高斯噪声污染的图像；中间是使用大小为7×7的算术平均滤波器滤波的结果；右图是使用同样尺寸的自适应均值滤波器滤波的结果。可以看出，自适应均值滤波的效果要明显优于算术均值滤波。



- [x] 自适应中值滤波

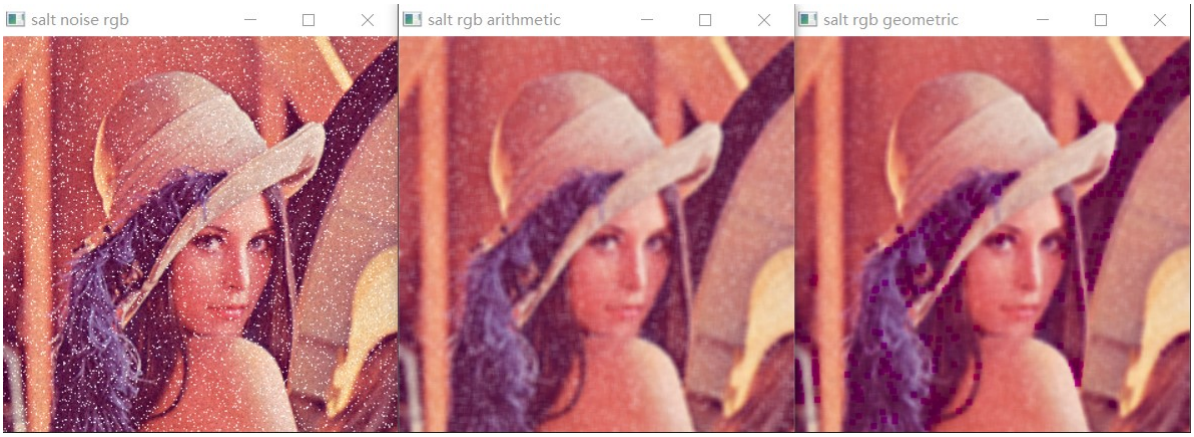
> 左图是被$P_{pepper} = 0.25, P_{salt} = 0.25$的椒盐噪声污染的图像；中间是使用大小为$7 \times 7$的中值滤波器滤波的结果；右图是使用初始大小$S_{xy} = 3 \times 3$，最大允许尺寸$S_{max} = 7 \times 7$的中值滤波器滤波的结果。可以看出，自适应中值滤波在保留清晰度和细节方面做得更好。



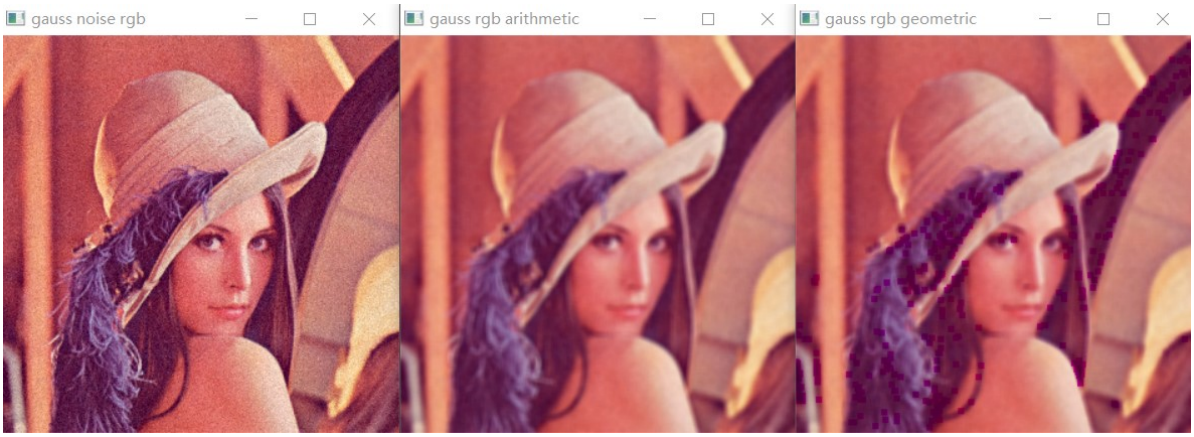
- [x] 彩色图像均值滤波

> 使用的彩色原始图像

> 左图是被$P_{salt} = 0.05$的盐噪声污染的图像；中间是用5×5尺寸的算术均值滤波器滤波的结果；右图是用同样尺寸的几何均值滤波器滤波的结果



> 左图是被均值为0，标准差为10个灰度级的加性高斯噪声污染的图像；中间是用5×5尺寸的算术均值滤波器滤波的结果；右图是用同样尺寸的几何均值滤波器滤波的结果



完整的源代码见附件`lab4.cpp`。