重 庆 交 通 大 学

学生实验报告

课 程 名 称：数 字 图 像 处 理 .

开 课实验室：软 件 实 验 中 心 .

学 院：信息学院 2021 年级物联网工程专业 2 班

学 生 姓 名：李骏飞 学 号 632109160602

指 导 教 师：蓝 章 礼 .

开 课 时 间：2023 至 2024 学年第 二 学期

成 绩	
教师签名	

实验项目名称		实验一：图像的去噪			
姓名	李骏飞	学号	**632109160602**	实验日期	**2024.4.2**
教师评阅： 1:实验目的明确□A□B□C□D 2:内容与原理□A□B□C□D 3:实验报告规范□A□B□C□D 4:实验主要代码与效果展示□A□B□C□D 5:实验分析总结全面□A□B□C□D					
实验记录					

一、实验目的

完成图像的加噪（包括随机噪声、黑白噪声），然后设计相应的算法对噪声进行去除或减弱操作，包括但不限于邻域平均、中值滤波等。

二、实验主要内容及原理

✧ 图像噪声

数码相机拍摄的任何图像中总会有一些噪音，噪声通常在图像中显示为随机斑点。它的产生，一般是图像在获取或是传输过程中收到随机信号干扰，妨碍人们对图像理解及分析处理的信号。很多时候，将图像噪声看作多维随机过程。图像噪声的产生来自图像获取中的环境条件和传感元器件自身的质量，常见的两种图像噪声包括：**椒盐噪声**和**高斯噪声**

（1）椒盐噪声：

椒盐噪声也称为脉冲噪声，是图像中经常见到的一种噪声，它是一种随机出现的自点（盐点）或者黑点（胡椒点），可能是亮的区域有黑色像素或是在暗的区域有自色像素（或是两者皆有）。椒盐噪声的成因可能是影像讯号受到突如其来的强烈干扰而产生、类比数位转换器或位元传输错误等。例如：失效的感应器导致像索值为最小值，饱和的感应器导致像素值为最大值。通过随机获取像素点并设置为高亮度点和低灰度点，可以实现向图像模拟添加椒盐噪声。（双极）脉冲噪声的 Probability Density Function PDF）

$$p(z)=\begin{cases} p_a & z=a \\ p_b & z=b \\ 0 & 其他 \end{cases}$$

如果 b>a，灰度值 b 在图像中将显示为一个亮点，a 的值将显示为一个暗点。若 P 或 P 为零，则脉冲噪声称为单极脉冲。如果 P 和 P 均不可能为零，尤其是它们近似相等时，脉冲噪声值将类似于随机分布在图像上的胡椒和盐粉微粒。由于这个原因，双极脉冲噪声也称为盐噪声。同时，它们有时也称为散粒和尖峰噪声。噪声脉冲可以是正的，也可以是负的。在一幅图像中，脉冲噪声总是数字化为最小值或最大值（纯黑或纯白）。负脉冲以一个黑点（胡椒点）出现在图像中。由于相同的原因，正脉冲以白

点（盐点）出现在图像中。

（2）高斯噪声：

高斯噪声是指高绿密度函数服从高斯分布的一类噪声。特别的，假设一个噪声，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，则称这个噪声为高斯白噪声。高斯白噪声的二阶矩不相关，一阶矩为常数，是指先后信号在时间上的相关性。高斯噪声是与光强没有关系的噪声，无论像素值是多少，噪声的平均水平（一般是 0）不变。高斯随机变量 z 的 Probability Density Function(PDF)由下式给出：

$$p(z) = \frac{1}{\sqrt{2\Pi}\sigma} e^{\frac{-(z-\mu)^2}{2\sigma^2}}$$

其中，z表示灰度值，μ表示z的平均值或期望值，σ表示 z 的标准差。标准差的平方σ^2称为 z 的方差

（3）均匀噪声

均匀噪声是一种具有均匀分布的噪声，也被称为平坦噪声或白噪声。其特点是其幅度在一定范围内均匀分布，没有明显的偏向性。在图像中，均匀噪声会导致像素值的随机波动，使图像看起来更加粗糙或杂乱。

在均匀噪声中，每个像素的噪声值是由一个均匀分布的随机数生成的，该随机数的范围通常是 [-A, A]，其中 A 是噪声强度的一半。通过将噪声值加到原始图像的像素值中，可以在图像中引入均匀噪声。

✧ 滤波器：

处理图像噪声的主要手段就是滤波器，图像的实质可以被理解为一种二维信号，而滤波本身是信号处理中的一个重要概念。在图像处理中，滤波是一常见的技术，滤波器分类包括：

线性滤波：对邻域中的像素的计算为线性运算时，如利用窗口函数进行平滑加权求和的运算，或者某种卷积运算，都可以称为线性滤波。常见的线性滤波有：**方框滤波、均值滤波、高斯滤波、拉普拉斯滤波**等等，通常线性滤波器之间只是模版的系数不同。

非线性滤波：非线性滤波利用原始图像跟模板之间的一种逻辑关系得到结果，如：**最值滤波器，中值滤波器，双边滤波器**

（1）均值滤波器

均值滤波是一种典型的线性滤波算法，主要是利用像素点邻域的像素值来计算像素点的值。其具体方法是首先给出一个滤波模板（卷积核）。该卷积核将覆盖像素点周围的其他邻域像素点，将像素点与其邻域像素点相加，然后取平均值，即为该像素点的新的像素值，这就是均值滤波的本质。即：在图片中一个方块区域 N*M 内（大部分情况下 N=M），中心点的像素为全部点像素值的平均值。均值滤波就是通过这个方块区域在整张图片上各个像素的滑动，对全部像素进行以上操作　该操作过程，实际上就是卷积的基本原理。

（2）高斯滤波

高斯滤波是一种低通滤波 ，其过滤掉图像高频成分(图像细节部分)，保留图像低频成分（图像平滑区域），所以图像会变得模糊。作为一种线性平滑滤波，适用于消除高斯噪声，广泛应用于图像处理的减噪过程。

通俗而言，高斯滤波就是对整幅图像进行加权平均的过程，每一个像素点的值，都由其本身和邻域内的其他像素值经过加权平均后得到。高斯滤波的具体操作是：用一个模板（或称卷积、掩模）扫描图像中的每一个像素，用模板确定的邻域内像素的加权平均灰度值去替代模板中心像素点的值。

（3）中值滤波

中值滤波是一种非线性滤波器，它将每个像素的值替换为邻域像素的中值。中值滤波对于去除椒盐噪声等概率噪声效果很好，因为它能够保持边缘细节。

（4）自适应中值滤波

自适应中值滤波是一种非线性图像滤波方法，用于去除图像中的噪声。与传统的中值滤波相比，自适应中值滤波具有更强的自适应性，能够根据局部区域内像素的特征来调整滤波器的大小。

滤波过程如下：

1.对于图像中的每个像素，定义一个初始的滤波器窗口大小；

2.在当前像素的滤波器窗口内，计算邻域内像素的最小值、最大值和中值。

3. 计算中值和当前像素值之间的差值，如果差值小于一定的阈值，则认为当前像素不受噪声干扰，保持原始像素值；否则，执行步骤 4。

4. 增加滤波器窗口的大小，重新计算邻域内像素的最小值、最大值和中值。

5. 重复步骤 3 和步骤 4，直到满足条件或达到最大滤波器窗口的大小。

6. 将经过滤波处理的像素值作为输出。

（5）领域平均滤波

邻域平均滤波的基本原理是利用周围像素的信息来平滑图像中的噪声。通过计算邻域内像素的平均值，噪声的影响可以被分散和减小，从而达到平滑图像的效果。该滤波器对于均匀分布的噪声有较好的去除效果，但在去除噪声的同时也可能导致图像细节的模糊。同时比较适用于轻度噪声的去除，对于强烈的噪声或图像细节保留的要求较高的情况，可能不够理想。

三、实验环境

Windows11

Visual Studio2021

C#语言

四、实验主要代码与效果展示

图像添加椒盐噪声

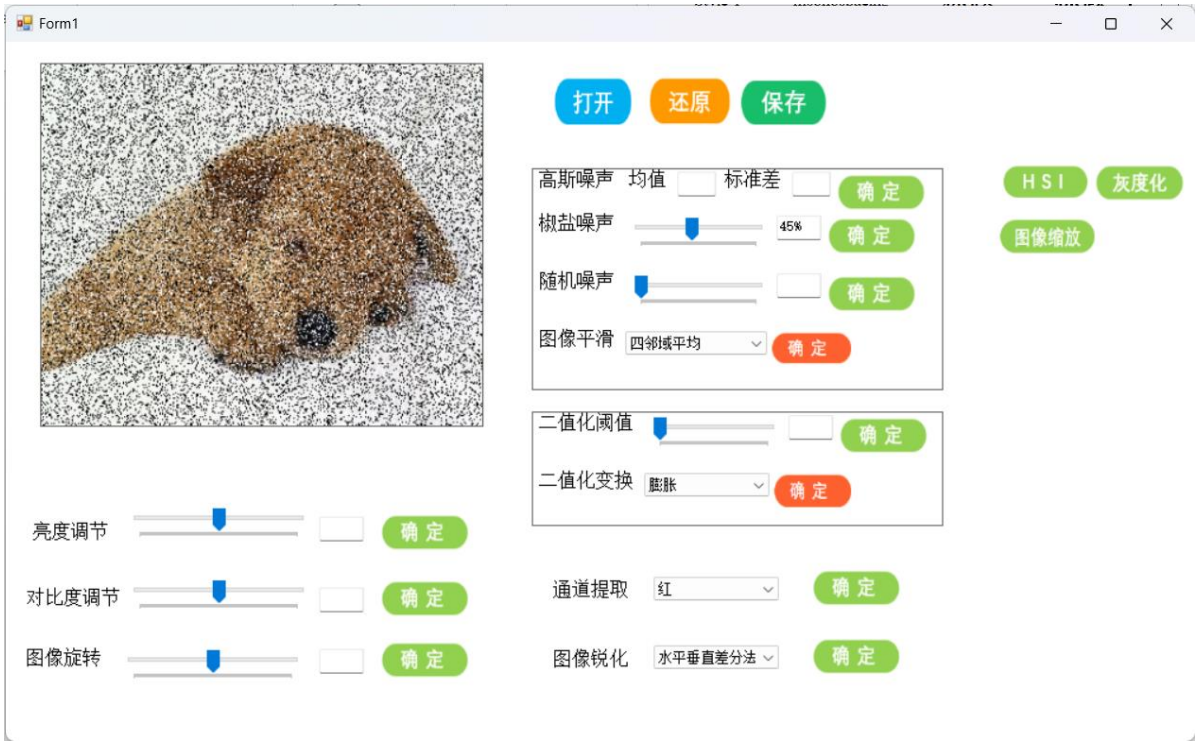
➢ 算法描述：

根据输入图像的宽度和高度，计算图像的总像素数 totalPixels，然后根据噪声比例参数 noiseRatio 即用户设置的噪声概率，计算需要添加椒盐噪声的像素数量 noisePixels。之后用 Random 类生成随机数，来确定添加椒盐噪声的像素位置。然后循环遍历每个噪声像素。在每次迭代中，随机生成像素的横坐标 x 和纵坐标 y。对于黑白像素点，我们可以使用 Color 类创建一个随机的黑色或白色像素，通过判断随机数的奇偶性来决定是黑色还是白色，如此就得到了随机的椒盐噪声点。

```
1.  //添加椒盐噪声的确定按
```

```csharp
钮? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?

private void btnImpulseadjust_Click(object sender, EventArgs e)
{
    if (!judge_pb_img_exit()) return;
    Bitmap bt2 = new Bitmap(original_image.Image);

    // 添加椒盐噪声
    //double noiseProbability = 0.05; // 噪声概率，控制噪声密度
    double noiseProbability = 1.0 * trackBar_impulseNoise.Value / 100; ; // 噪声概率，控制噪声密度
    //Bitmap noisyImage = AddSaltAndPepperNoise(bt2, noiseProbability);
    Bitmap noisyImage = ImageNoiseGenerator.AddSaltAndPepperNoise(bt2, noiseProbability);
    ////刷新图片框
    pictureBox_show.Refresh();
    pictureBox_show.Image = noisyImage;

}
//添加椒盐噪声
public static Bitmap AddSaltAndPepperNoise(Bitmap image, double noiseRatio)
{
    Bitmap noisyImage = new Bitmap(image);

    int width = noisyImage.Width;
    int height = noisyImage.Height;
    int totalPixels = width * height;
    int noisePixels = (int)(noiseRatio * totalPixels);

    Random random = new Random();

    for (int i = 0; i < noisePixels; i++)
    {
        int x = random.Next(width);
        int y = random.Next(height);

        Color randomColor = random.Next(2) == 0 ? Color.Black : Color.White;
        noisyImage.SetPixel(x, y, randomColor);
    }

    return noisyImage;
}

```

➢ **演示效果：**



图像添加高斯噪声

➢ 算法描述：

一般默认高斯噪声的均值为 0，然后根据用户设置的标准差，来生成

高斯分布随机数。但是这里我定义了两个文本框来获取用户输入的**均值**和**方差参数**，并将其解析为 double 类型的浮点数。如果解析成功，就可以使用这两个参数进行后续操作。如果解析失败，就弹出一个错误提示框。

然后定义了一个名为 NextGaussian 的方法来生成满足高斯分布的随机数。该方法接受一个 Random 对象、均值和标准差作为参数，它使用 Box-Muller 转换来生成服从标准正态分布的随机数，然后通过乘以方差和加上均值的方式，得到服从指定均值和方差的随机数。并返回一个满足高斯分布的随机数。

在 AddGaussianNoise 方法中，创建一个 Random 对象，生成随机数，再使用嵌套的循环遍历原始图像的每个像素，对于每个像素，获取其颜色值，并调用方法生成高斯随机数，接着将生成的随机数与像素的 R、G、B 通道值相加，并将结果限制在 0 到 255 之间，确保像素值的有效性，最后创建一个新的 Color 对象 noisyPixel，使用修正后的颜色通道值和原始像素的 Alpha 值。

此就得到了随机的椒盐噪声点。

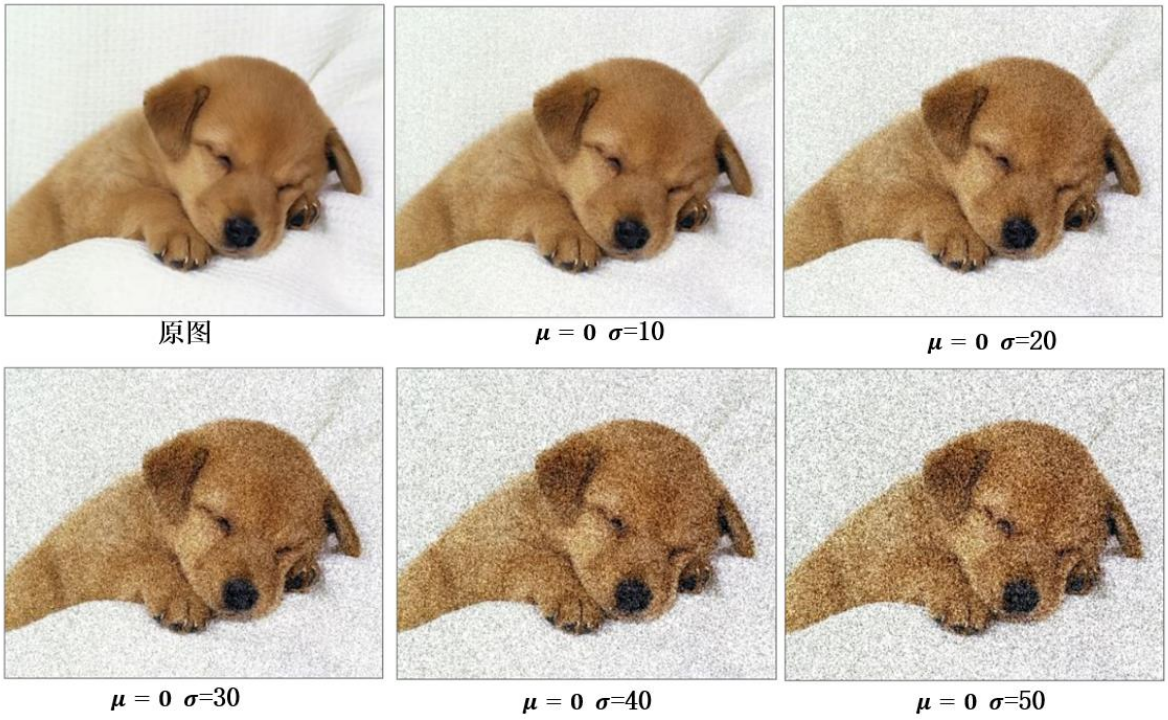
```
1.    //添加高斯噪声的确定按钮
2.  private void btnGaussianadjust_Click(object sender, EventArgs e)
3.  {
4.      if (!judge_pb_img_exit()) return;
5.      Bitmap bt1 = new Bitmap(original_image.Image);
6.
7.      double mean;// 均值
8.      double standardDeviation; // 标准差，控制噪声强度
9.      if (double.TryParse(textBox_gaussianAvg.Text, out mean) &&
10.         double.TryParse(textBox_gaussianVariance.Text, out standardDeviation))
11.     {
12.         // 转换成功，可以使用 mean 变量和 standardDeviation 变量进行后续操作
13.         mean = double.Parse(textBox_gaussianAvg.Text); // 均值
14.         standardDeviation = double.Parse(textBox_gaussianVariance.Text);
15.         Bitmap noisyImage = AddGaussianNoise(bt1, mean, standardDeviation);
16.         pictureBox_show.Refresh();//刷新图片框
17.         pictureBox_show.Image = noisyImage;
18.     }
19.     else
20.     {
21.         // 转换失败，弹窗提示报错
22.         MessageBox.Show("输入的文本不是有效的数字", "错误
    ", MessageBoxButtons.OK, MessageBoxIcon.Error);
23.     }
24. }
25. public static double NextGaussian(Random random, double mean, double standardDevia
    tion)
26. {
27.     double u1 = 1.0 - random.NextDouble();
```

```
    double u2 = 1.0 - random.NextDouble();
    double randStdNormal = Math.Sqrt(-2.0 * Math.Log(u1)) * Math.Sin(2.0 * Math.PI * u2);
    double randNormal = mean + standardDeviation * randStdNormal;
    return randNormal;
}
public static Bitmap AddGaussianNoise(Bitmap image, double mean, double standardDeviation)
{
    Random random = new Random();
    Bitmap result = new Bitmap(image.Width, image.Height);

    for (int x = 0; x < image.Width; x++)
    {
        for (int y = 0; y < image.Height; y++)
        {
            Color pixel = image.GetPixel(x, y);
            double gaussian = NextGaussian(random, mean, standardDeviation);

            int red = (int)(pixel.R + gaussian);
            int green = (int)(pixel.G + gaussian);
            int blue = (int)(pixel.B + gaussian);

            red = Math.Max(0, Math.Min(255, red));
            green = Math.Max(0, Math.Min(255, green));
            blue = Math.Max(0, Math.Min(255, blue));

            Color noisyPixel = Color.FromArgb(pixel.A, red, green, blue);
            result.SetPixel(x, y, noisyPixel);
        }
    }

    return result;

}
```

- **演示效果：**

添加高斯噪声的可视化界面设计：

保持均值不变，改变方差，由于噪声服从高斯分布,所以方差越大,数据越分散,噪声也就越多。



保持方差不变，改变均值，均值决定着整个图像的明亮程度,均值大于0,表示图像加上一个使自己变亮的噪声,小 于 0,表示图像加上一个使自己变暗的噪声。

原图

$\mu = -20\ \sigma=30$

$\mu = -10\ \sigma=30$

$\mu = 10\ \sigma=30$

$\mu = 20\ \sigma=30$

$\mu = 30\ \sigma=30$

图像添加均匀噪声

- 算法描述:

获取滑动条 trackBar_RandomNoise 的值 noiseProbability，该值表示随机噪声的强度，对于每个像素，获取其原始颜色值 pixel，使用 random 类的方法生成一个介于 -noiseProbability 和 noiseProbability 之间的随机噪声值，将噪声值与原始像素的 RGB 通道值相加，并使用 Clamp 方法将结果限定在 0 和 255 之间，确保颜色值的合法范围。

Clamp 方法用于将值限定在指定的范围内。如果值小于最小值，则返回最小值；如果值大于最大值，则返回最大值；否则返回原始值.

```
//添加随机噪声的确定按钮? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
private void btnRandomadjust_Click(object sender, EventArgs e)
{
    if (!judge_pb_img_exit()) return;
    Bitmap bt2 = new Bitmap(original_image.Image);
    int noiseProbability = trackBar_RandomNoise.Value;//噪声概率
    Random random = new Random();

    for (int i = 0; i < bt2.Width; i++)
    {
        for (int j = 0; j < bt2.Height; j++)
        {
            Color pixel = bt2.GetPixel(i, j);

```

```
            int noise = random.Next(-noiseProbability, noiseProbability + 1);

            int red = Clamp(pixel.R + noise, 0, 255);
            int green = Clamp(pixel.G + noise, 0, 255);
            int blue = Clamp(pixel.B + noise, 0, 255);

            Color newPixel = Color.FromArgb(red, green, blue);
            bt2.SetPixel(i, j, newPixel);
        }
    }


    pictureBox_show.Refresh();
    pictureBox_show.Image = bt2;

}
private int Clamp(int value, int min, int max)
{
    if (value < min)
        return min;
    if (value > max)
        return max;
    return value;
}
```

- 实现效果：

原图

原图

随机噪声范围（-17，18）

随机噪声范围（-61，62）

随机噪声范围（—116，117）

使用高斯滤波：

➢ 算法描述：

GenerateGaussianKernel 函数用于生成高斯核。它接受两个参数：标准差 sigma 和滤波器的大小 size。该函数使用高斯函数的公式来计算每个位置上的权重，并将这些权重存储在一个二维数组（即高斯核）中。

高斯核的大小是一个奇数，以便确保核有一个中心元素。sigma 控制了权重分布的广度。高斯核的计算基于高斯函数的定义，即 exp(-(x^2 + y^2) / (2 * sigma^2)) / (2 * pi * sigma^2)。

在计算高斯核时，首先计算每个位置 (x, y) 处的权重，然后将权重进行标准化，以确保总和为 1。这样做是为了保持滤波后图像的亮度不变。

```
1.  private static double[,] GenerateGaussianKernel(double sigma, int size)
2.  {
3.      double[,] kernel = new double[size, size];
4.      double sum = 0;
5.
6.      int radius = size / 2;
7.
```

```
    for (int y = -radius; y <= radius; y++)
    {
        for (int x = -radius; x <= radius; x++)
        {
            double exponent = -(x * x + y * y) / (2 * sigma * sigma);
            double weight = Math.Exp(exponent) / (2 * Math.PI * sigma * sigma);

            kernel[x + radius, y + radius] = weight;
            sum += weight;
        }
    }

    // 标准化核函数
    for (int y = 0; y < size; y++)
    {
        for (int x = 0; x < size; x++)
        {
            kernel[x, y] /= sum;
        }
    }
    return kernel;
}

```

Apply 函数，接受三个参数：输入图像 image、标准差 sigma 和滤波器大小 kernelSize，用于将高斯滤波应用于输入图像。

对于图像中的每个像素 (x, y)，使用一个二重循环来遍历邻域，计算出以该像素为中心的邻域内像素的加权平均值，对于每个邻域像素，根据其与中心像素的相对位置，从预先生成的高斯核中获取对应位置的权重。乘以该像素的颜色分量（红、绿、蓝）并将其累加到对应的和中，同时累加权重和。然后将每个颜色分量的加权和除以权重和，得到平均值。使用 Math.Round 方法将平均值四舍五入为最接近的整数，并确保它在 0 到 255 的范围内。

根据计算得到的红、绿、蓝值创建一个新的 Color 对象，并将其设置为结果图像 result 中对应像素的颜色，然后返回结果图像即可。

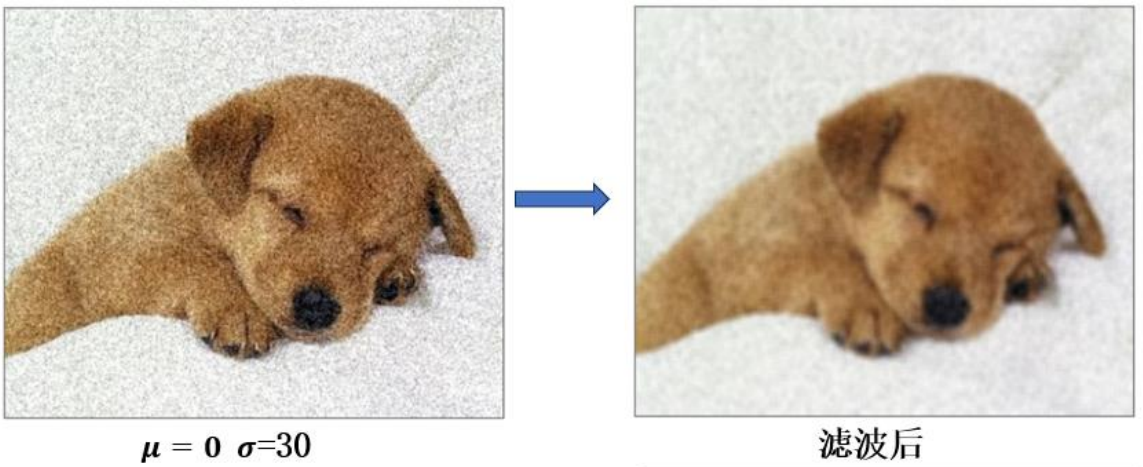
```
//返回高斯滤波之后的图像? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
public static Bitmap Apply(Bitmap image, double sigma, int kernelSize)
{
    Bitmap result = new Bitmap(image.Width, image.Height);

    int radius = kernelSize / 2;
    double[,] kernel = GenerateGaussianKernel(sigma, kernelSize);

    for (int y = 0; y < image.Height; y++)
    {
        for (int x = 0; x < image.Width; x++)
        {
```

```
            double redSum = 0;
            double greenSum = 0;
            double blueSum = 0;
            double weightSum = 0;

            for (int j = -radius; j <= radius; j++)
            {
                for (int i = -radius; i <= radius; i++)
                {
                    int posX = x + i;
                    int posY = y + j;

                    if (posX >= 0 && posX < image.Width && posY >= 0 && posY < ima
ge.Height)
                    {
                        Color pixel = image.GetPixel(posX, posY);
                        double weight = kernel[i + radius, j + radius];

                        redSum += pixel.R * weight;
                        greenSum += pixel.G * weight;
                        blueSum += pixel.B * weight;
                        weightSum += weight;
                    }
                }
            }

            int red = (int)Math.Round(redSum / weightSum);
            int green = (int)Math.Round(greenSum / weightSum);
            int blue = (int)Math.Round(blueSum / weightSum);

            red = Math.Max(0, Math.Min(255, red));
            green = Math.Max(0, Math.Min(255, green));
            blue = Math.Max(0, Math.Min(255, blue));

            Color filteredColor = Color.FromArgb(red, green, blue);
            result.SetPixel(x, y, filteredColor);
        }
    }

    return result;
}
```

➢ 实现效果：

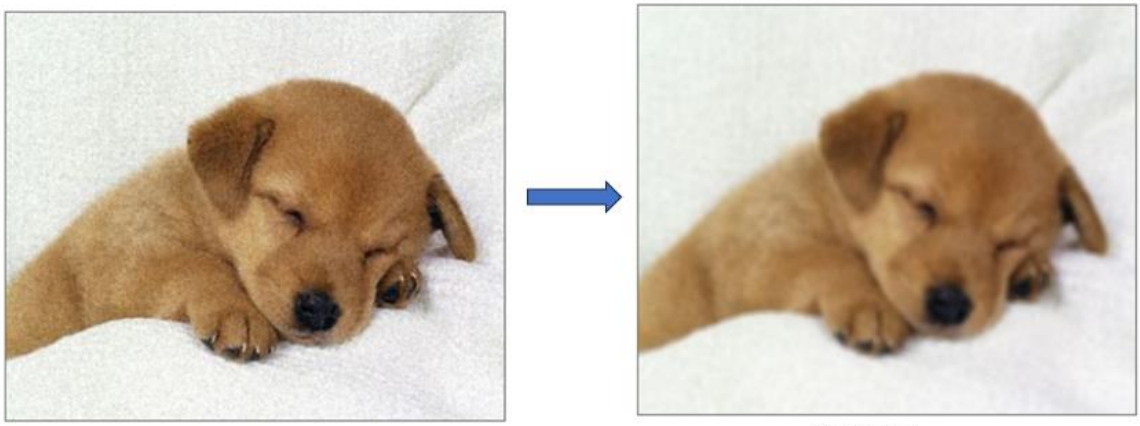
保留了图像的整体特征。由标准差和滤波器大小控制生成不同的高斯核，实现不同程度的平滑效果，但是存在一定的细节模糊现象。

滤波对比：



$\mu = 0$ σ=30

滤波后



$\mu = 0$ σ=10

滤波后

使用中值滤波

➢ 算法描述：

遍历当前像素点周围的 3×3 邻域。在每个邻域内，获取像素的 RGB 值，并将其红色通道的值存入数组 dt 中。在获取完邻域内所有像素的红色通道值后，通过两个嵌套的 for 循环对数组 dt 进行排序，将值从小到大排列。通过取数组 dt 中位于中间位置的值作为新的像素值 rr，实现中值滤波的效果。

```
        //把 bt1 位图对象赋值给图像框
private void image_smooth_中值滤波(object sender, EventArgs e)
{
    Color c = new Color();
    Bitmap bt1 = new Bitmap(pictureBox_show.Image);
    Bitmap bt2 = new Bitmap(pictureBox_show.Image);
    int rr, r1, dm, m;
    //设置一个数组用于储存 3×3 像素快的 r 分量值
    int[] dt = new int[20];
    for (int i = 1; i < bt1.Width - 1; i++)
    {
        for (int j = 1; j < bt1.Height - 1; j++)
        {
            rr = 0; m = 0;
            for (int k = -1; k < 2; k++)
            {
                for (int n = -1; n < 2; n++)
                {
                    //获取该坐标的像素并存入数组 dt[]中
                    c = bt1.GetPixel(i + k, j + n);
                    r1 = c.R;
                    dt[m++] = r1;
                }
            }
            for (int p = 0; p < m - 1; p++)
            {
                for (int q = p + 1; q < m; q++)
                {
                    //对存与数组里的数据进行从大到小的排序
                    if (dt[p] > dt[q])
                    {
                        dm = dt[p];
                        dt[p] = dt[q];
                        dt[q] = dm;
                    }
                }
            }
            //获取数值所有存储数据的中间值
            rr = dt[(int)(m / 2)];
            bt2.SetPixel(i, j, Color.FromArgb(rr, rr, rr));
        }
        pictureBox_show.Refresh();
        pictureBox_show.Image = bt2;
```

```
44.     }
45.
46. }
```

- 实现效果：

原图

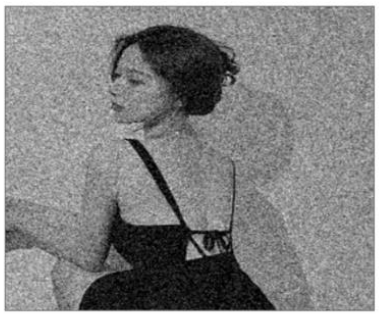
加入17%的椒盐噪声

采用中值滤波降噪

加入32%的椒盐噪声

采用中值滤波降噪

原图



μ=0， σ=30的高斯噪声

采用中值滤波降噪

μ=0， σ=50的高斯噪声

采用中值滤波降噪

中值滤波处理椒盐噪声较低的图像时效果较好，处理高斯噪声的效果一般

使用自适应中值滤波

- 算法描述：

这里我指定中值滤波器的最大窗口大小为 7，然后定义 selfAdaptMedianFilter 静态方法方法，用于实现自适应中值滤波算法。该方法接受一个原始图像 image 和最大窗口大小 maxWindowSize 作为参数，并返回处理后的图像。

在该方法中，使用双重循环遍历图像的每个像素。在每个像素位置，定义一个初始滤波器窗口大小为 3x3。

然后定义一个循环(**算法的核心部分**)，用于自适应地调整滤波器窗口的大小，直到滤波器窗口大小达到最大值，它通过遍历滤波器窗口内的像素来确定当前像素是否受到噪声干扰，并根据判断结果进行处理。循环将获取的像素值存储在数组 pixels 的相应位置，然后计算数组 pixels 中间位置的像素值，即中值。同时获取当前像素 (x, y) 的 RGB 值，比较当前像素值 currentPixel 与数组 pixels 的最小值和最大值。如果当前像素值介于最小值和最大值之间，说明当前像素不受噪声干扰，保持原始像素值。如果当前像素值不在最小值和最大值范围内，说明当前像素受到噪声干扰，需要扩大窗口。如果窗口大小超过了最大窗口大小 maxWindowSize，表示达到了最大滤波器窗口的大小仍然没有找到合适的像素值，此时使用中值作为输出像素值

```
private void image_smooth_自适应中值滤波(object sender, EventArgs e)
{
    Bitmap image = new Bitmap(pictureBox_show.Image);

    int maxWindowSize = 7; // 最大窗口大小
    Bitmap filteredImage = selfAdaptMedianFilter(image, maxWindowSize);
    pictureBox_show.Refresh();
    pictureBox_show.Image = filteredImage;
}
//返回自适应中值滤波之后的图
像? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
public static Bitmap selfAdaptMedianFilter(Bitmap image, int maxWindowSize)
{
    Bitmap result = new Bitmap(image.Width, image.Height);

    int width = image.Width;
    int height = image.Height;

    int halfMaxWindowSize = maxWindowSize / 2;

    for (int y = 0; y < height; y++)
    {
```

```
        for (int x = 0; x < width; x++)
        {
            int windowSize = 3; // 初始滤波器窗口大小为 3x3

            while (windowSize <= maxWindowSize)
            {
                int halfWindowSize = windowSize / 2;
                int[] pixels = new int[windowSize * windowSize];

                int index = 0;
                for (int dy = -halfWindowSize; dy <= halfWindowSize; dy++)
                {
                    for (int dx = -halfWindowSize; dx <= halfWindowSize; dx++)
                    {
                        int nx = x + dx;
                        int ny = y + dy;

                        // 边界处理，超出图像范围的像素使用原始像素值
                        if (nx < 0 || nx >= width || ny < 0 || ny >= height)
                        {
                            pixels[index] = image.GetPixel(x, y).ToArgb();
                        }
                        else
                        {
                            pixels[index] = image.GetPixel(nx, ny).ToArgb();
                        }

                        index++;
                    }
                }

                Array.Sort(pixels);

                int median = pixels[pixels.Length / 2];
                int currentPixel = image.GetPixel(x, y).ToArgb();

                if (currentPixel > pixels[0] && currentPixel < pixels[pixels.Lengt
h - 1])
                {
                    // 当前像素不受噪声干扰，保持原始像素值
                    result.SetPixel(x, y, Color.FromArgb(currentPixel));
                    break;
                }
                else
                {
                    // 增加滤波器窗口的大小
                    windowSize += 2;

                    // 达到最大滤波器窗口的大小仍然没有找到合适的像素值，使用中值
                    if (windowSize > maxWindowSize)
                    {
                        result.SetPixel(x, y, Color.FromArgb(median));
                    }
                }
            }
        }
```

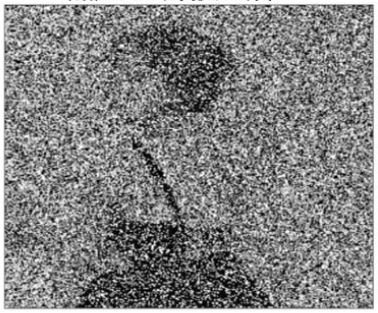
```
    }

    return result;
}
private static int[] GetWindowValues(Bitmap image, int x, int y, int windowSize)
{
    int halfWindowSize = windowSize / 2;
    int[] values = new int[windowSize * windowSize];
    int index = 0;

    for (int j = -halfWindowSize; j <= halfWindowSize; j++)
    {
        for (int i = -halfWindowSize; i <= halfWindowSize; i++)
        {
            int neighborX = x + i;
            int neighborY = y + j;

            if (neighborX < 0 || neighborX >= image.Width || neighborY < 0 || neighborY >= image.Height)
            {
                values[index] = 0; // or any default value for out-of-bounds pixels
            }
            else
            {
                  values[index] = image.GetPixel(neighborX, neighborY).R;
              }

              index++;
          }
      }

      return values;
  }
```

- 实现效果：

原图

加入38%的椒盐噪声

采用自适应中值滤波降噪



加入81%的椒盐噪声

采用自适应中值滤波降噪

原图

μ=0， σ=30的高斯噪声

采用自适应中值滤波降噪

μ=0， σ=50的高斯噪声

采用自适应中值滤波降噪

自适应中值滤波处理椒盐噪声的效果很好，但是处理高斯噪声效果很差。

使用邻域平均滤波（四邻域，八邻域）

➢ 算法描述：

通过两层嵌套的循环遍历图像中的每个像素，对于每个像素，获取其

四个邻域(八个邻域）像素的 RGB 通道值。计算**四个邻域(八个邻域）**像素的 RGB 通道值的平均值，将平均值作为当前像素的新 RGB 值，赋给 bt2 中对应的像素。

```
1.  private void image_smooth_四邻域平均(object sender, EventArgs e)
2.  {
3.      Bitmap bt1 = new Bitmap(pictureBox_show.Image);
4.      Bitmap bt2 = new Bitmap(pictureBox_show.Image); //定义两个位图对象
5.      int R1, R2, R3, R4, Red;
6.      int G1, G2, G3, G4, Green;
7.      int B1, B2, B3, B4, Blue;
8.      for (int i = 1; i < bt1.Width - 1; i++)
9.      {
10.         for (int j = 1; j < bt1.Height - 1; j++)
11.         {
12.             R1 = bt1.GetPixel(i, j - 1).R;
13.             R2 = bt1.GetPixel(i, j + 1).R;
14.             R3 = bt1.GetPixel(i - 1, j).R;
15.             R4 = bt1.GetPixel(i + 1, j).R;
16.             Red = (R1 + R2 + R3 + R4) / 5;
17.             G1 = bt1.GetPixel(i, j - 1).G;
18.             G2 = bt1.GetPixel(i, j + 1).G;
19.             G3 = bt1.GetPixel(i - 1, j).G;
20.             G4 = bt1.GetPixel(i + 1, j).G;
21.             Green = (G1 + G2 + G3 + G4) / 5;
22.             B1 = bt1.GetPixel(i, j - 1).B;
23.             B2 = bt1.GetPixel(i, j + 1).B;
24.             B3 = bt1.GetPixel(i - 1, j).B;
25.             B4 = bt1.GetPixel(i + 1, j).B;
26.             Blue = (B1 + B2 + B3 + B4) / 5;
27.             bt2.SetPixel(i, j, Color.FromArgb(Red, Green, Blue));
28.         }
29.         pictureBox_show.Refresh();//刷新图片框
30.         pictureBox_show.Image = bt2;
31.     }
32. }
33. private void image_smooth_八邻域平均(object sender, EventArgs e)
34. {
35.     Bitmap bt1 = new Bitmap(pictureBox_show.Image);
36.     Bitmap bt2 = new Bitmap(pictureBox_show.Image); //定义两个位图对象
37.     int R1, R2, R3, R4, R5, R6, R7, R8, Red;
38.     int G1, G2, G3, G4, G5, G6, G7, G8, Green;
39.     int B1, B2, B3, B4, B5, B6, B7, B8, Blue;
40.     for (int i = 1; i < bt1.Width - 1; i++)
41.     {
42.         for (int j = 1; j < bt1.Height - 1; j++)
43.         {
44.             R1 = bt1.GetPixel(i - 1, j - 1).R;
45.             R2 = bt1.GetPixel(i, j - 1).R;
46.             R3 = bt1.GetPixel(i + 1, j - 1).R;
47.             R4 = bt1.GetPixel(i - 1, j).R;
48.             R5 = bt1.GetPixel(i + 1, j).R;
49.             R6 = bt1.GetPixel(i - 1, j + 1).R;
50.             R7 = bt1.GetPixel(i, j + 1).R;
51.             R8 = bt1.GetPixel(i + 1, j + 1).R;
```

```
52.             Red = (int)(R1 + R2 + R3 + R4 + R5 + R6 + R7 + R8) / 8;
53.             G1 = bt1.GetPixel(i - 1, j - 1).G;
54.             G2 = bt1.GetPixel(i, j - 1).G;
55.             G3 = bt1.GetPixel(i + 1, j - 1).G;
56.             G4 = bt1.GetPixel(i - 1, j).G;
57.             G5 = bt1.GetPixel(i + 1, j).G;
58.             G6 = bt1.GetPixel(i - 1, j + 1).G;
59.             G7 = bt1.GetPixel(i, j + 1).G;
60.             G8 = bt1.GetPixel(i + 1, j + 1).G;
61.             Green = (int)(G1 + G2 + G3 + G4 + G5 + G6 + G7 + G8) / 8;
62.             B1 = bt1.GetPixel(i - 1, j - 1).B;
63.             B2 = bt1.GetPixel(i, j - 1).B;
64.             B3 = bt1.GetPixel(i + 1, j - 1).B;
65.             B4 = bt1.GetPixel(i - 1, j).B;
66.             B5 = bt1.GetPixel(i + 1, j).B;
67.             B6 = bt1.GetPixel(i - 1, j + 1).B;
68.             B7 = bt1.GetPixel(i, j + 1).B;
69.             B8 = bt1.GetPixel(i + 1, j + 1).B;
70.             Blue = (int)(B1 + B2 + B3 + B4 + B5 + B6 + B7 + B8) / 8;
71.             bt2.SetPixel(i, j, Color.FromArgb(Red, Green, Blue));
72.         }
73.         pictureBox_show.Refresh();//刷新图片框
74.         pictureBox_show.Image = bt2;
75.     }
76. }
```

- 实现效果：

原图

随机噪声范围在（-30，31）

四邻域平均

μ=0， σ=30的高斯噪声

八邻域平均

五、实验结果及分析(包括心得体会，本部分为重点，不能抄袭复制)

- **完成情况:**

完成了实验全部的基本要求和全部的扩展要求，最终的结果基本达到了我的预期

- **实验结果与讨论:**

高斯滤波：高斯滤波是一种基于高斯函数的线性平滑滤波器，能够有效地去除高斯噪，能够平滑图像并保持边缘细节。

中值滤波：中值滤波是一种非线性滤波器，将像素点周围邻域内的像素值排序，并用中值替代当前像素值。中值滤波对椒盐噪声具有较好的去除效果，能够有效消除离群点。

自适应中值滤波：自适应中值滤波是在中值滤波的基础上增加了动态邻域大小的调整机制。该算法能够根据邻域像素值的统计特征自适应地调整邻域大小，对不同强度的图像噪声有较好的去噪效果。

领域均匀滤波：领域均匀滤波是一种基于均匀噪声模型的滤波器，将邻域内像素值的平均值作为当前像素值。该滤波器对随机噪声有一定的去噪效果，但对于其他噪声类型的去除效果较差。

- **实验心得**

在实验中，我发现不同的噪声类型适用于不同的去噪算法，根据噪声的特点选择合适的算法能够取得更好的去噪效果。并且每个去噪算法都有一些参数需要调整，例如滤波器的大小、邻域大小等。通过仔细调整这些参数，可以获得更好的去噪效果。在实验中，我尝试了不同的参数组合，并观察其对去噪效果的影响。另外在图像去噪处理中，细节保留也是一个重要的考虑因素。有时过度的去噪处理可能会导致图像变得模糊或细节丢失。因此，在选择和调整去噪算法时，需要注意在去除噪声的同时尽可能保留图像的细节。并且多种算法也可以组合应用，我尝试了将多种算法进行组合应用的方法。例如，先使用中值滤波处理椒盐噪声，再使用高斯滤波进行平滑处理。这种组合应用能够进一步提高去噪效果，并在一定程度上解决单一算法的局限性。

综上所述，本次实验通过对图像添加不同类型的噪声，并使用高斯滤波、中值滤波、自适应中值滤波和领域均匀滤波等算法进行去噪处理，我对不同算法的效果和特点有了更深入的了解。同时，通过调整参数、平衡

去噪与细节保留的关系以及尝试多种算法的组合应用，我获得了更好的去噪效果。在未来的研究中，可以进一步探索其他图像去噪算法，并结合更多的评价指标，以提高图像去噪的质量和效果。