数字图像处理第二次实验

09021220 **周天逸**

一、

- 实验内容：

  i. 构建一个灰度值全是1的灰度图像，加入一定方差的高斯白噪声（要能看到明显的噪声），判断频谱图是否接近水平；

- 实验原理

  i. 高斯白噪声

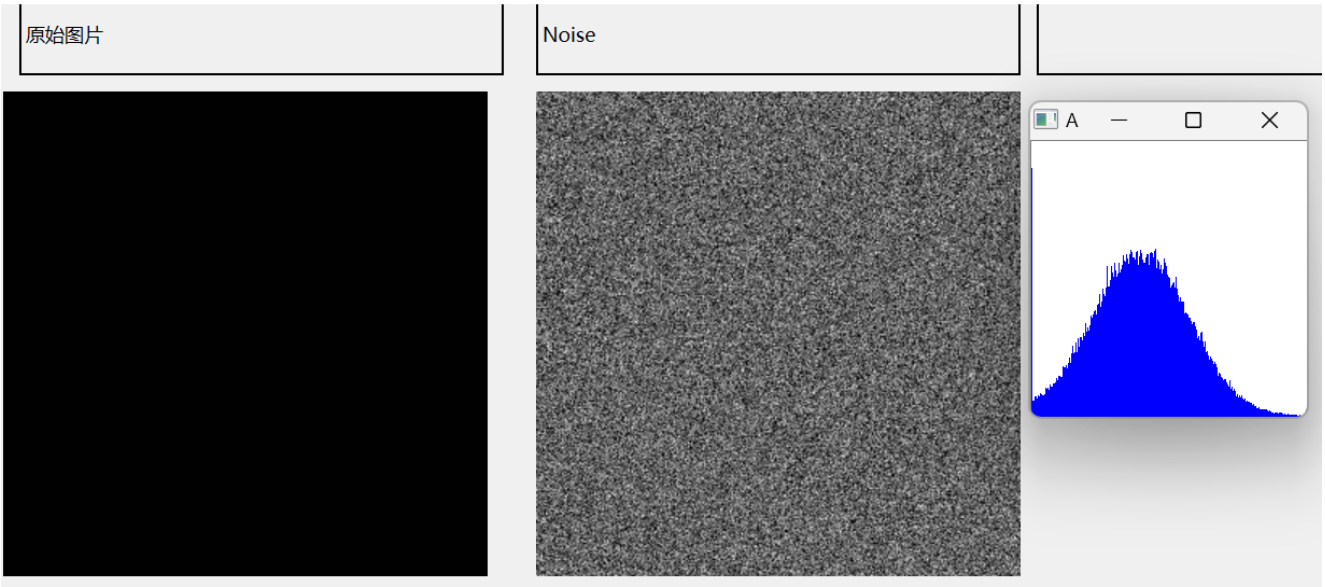
  a. 如果一个噪声，它的瞬时值服从高斯分布，而它的功率谱密度又是均匀分布的，则称它为高斯白噪声。

  b. 加入噪声后频谱图的变换：使得频谱值产生偏差，而偏差的分布近似于正态高斯分布。

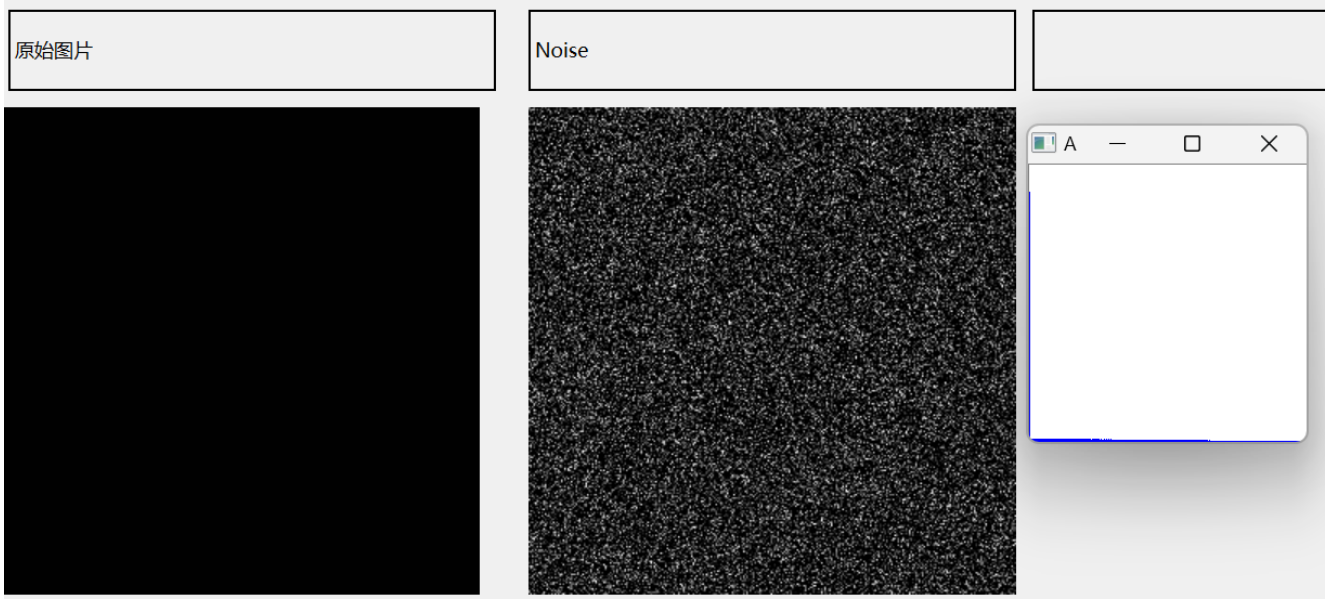
  c. 加入噪声本质就是加入随机数，改变图片的灰度值，让灰度值发生变化。而高斯噪声让灰度值的变化符合正态分布也就是灰度值绝大部分分布在均值点附近而远离均值点越远概率越小，分布的点越少。

- 实验结果

  i. 图片展示：



  ii. 左侧原始图片为构建的灰度值全是1的图片，显示是全黑的，然后加入高斯噪声，设置一定的方差和均值，不是原本的灰度值1为均值，新选择了均值。所以结果的图片不是水平的直线，而是类似正态分布。



iii. 当将均值设置为原来的灰度值1时，频谱图可以发现是水平的直线，大部分都聚集再1的附近而远离1的灰度值分布的点数就很少。所以呈现水平

- 代码展示

i. 图片加入高斯噪声。

```
void GaussianNoiseSingleChannel(const Mat &src, Mat &dst,double u, double v)
{

Mat noiseImg(src.size(), CV_64FC1);           //存放噪声图像
RNG rng((unsigned)time(NULL));                //生成随机数 （均值，高斯）
rng.fill(noiseImg, RNG::NORMAL, u, v);        //随机高斯填充矩阵
Mat yccImg;                                   //用来进行对原图像的转换
cvtColor(src, yccImg, COLOR_BGR2YCrCb);       //色彩空间转换；
Mat sigImg[3];                                //用来存储单通道图像
split(yccImg, sigImg);                        //将图像分解到单通道,一幅灰度图像
sigImg[0].convertTo(sigImg[0], CV_64FC1);     //将uchar转为double
sigImg[0] = sigImg[0] + noiseImg;             //添加高斯噪声（）
sigImg[0].convertTo(sigImg[0], CV_8UC1);      //Y通道加高斯噪声后图像,自动截断小于零和大于
255的值
Mat gaussianImg(src.size(), CV_8UC3);         //添加高斯噪声的图像；
merge(sigImg, 3, gaussianImg);                //和并三个通道
cvtColor(gaussianImg, dst, COLOR_YCrCb2BGR); //色彩空间转换
}
```

ii. 高斯噪声的生产首先构建随机数填充高斯矩阵，将输入的图片通道分离开，然后再一个维度上进行操作。再原图上添加高斯噪声，将灰度值大于255的分布在255，小于0的分布再0.

二、

- 实验内容

i. 对lena灰度图像加入高斯噪声，分别对其进行中值处理、均值处理、自适应中值处理、Nonlocal means处理并进行图像对比

ii. 对lena灰度图像加入一定密度的椒盐噪声（要能看到明显的噪声），分别对其进行中值处理、均值处理、自适应中值处理、Nonlocal means处理并进行图像对比

- 实验原理---不同滤波器的原理

  i. 均值滤波器

     a. 原理：其原理是基于局部像素值的平均来替代每个像素点的灰度值，从而降低图像中的高频噪声。

     b. 步骤

        a. 通过定义一个滤波窗口：均值滤波器需要一个固定大小的窗口，通常是一个正方形或矩形的区域，来滑动遍历整个图像

        b. 将窗口中的像素值取平均：对于窗口中的每个像素，将其灰度值与周围像素的灰度值一起取平均。这个平均值将被分配给窗口中心的像素，以平滑图像。

     c. 优点

        a. 除高频噪声

     d. 缺点：

        a. 导致图像细节的丧失，因为它会模糊图像。

  ii. 中值滤波器

     a. 原理：基于一组像素值的中间值（即中位数）来替代每个像素点的灰度值，从而有效地减少异常值对图像的影响。有利于处理椒盐噪声。

     b. 步骤

        a. 对于窗口中的每个像素，将其灰度值记录下来以形成一个像素值集合。

        b. 计算中值：对像素值集合进行排序，并选择中间位置的像素值作为中值。如果像素值的数量为奇数，中值就是中间值；如果像素值的数量为偶数，中值通常是中间两个值的平均值。

     c. 优点：

        a. 去除椒盐噪声和其他突发噪声效果显著，因为它使用中值而不受异常值的影响。

  iii. 自适应中值处理

     a. 原理：自适应中值滤波器可以根据每个像素的邻域条件自动调整滤波窗口的大小，以更好地适应图像的特性和噪声水平。这使得它在不同区域具有不同噪声水平的图像中表现出色。

     b. 步骤：

        a. 先计算最小窗口中的中值，如果中值fmed,如果$fmed<fmax,fmed>fmin$进入第二步，否则扩大窗口大小。如果到达最大窗口大小后还是没有进入第二步返回$f_{i,j}$

        b. 判断 $f_{i,j}>fmin,f_{i,j}<fmax$则返回$f_{i,j}$否则返回fmed

     c. 优点：

        a. 根据图像的局部特性自动调整窗口大小，从而更好地处理不同区域的图像噪声。它在去除椒盐噪声和其他类型噪声方面效果良好，同时保留了图像的细节

        b. 尽量保护原来图像的信息，而舍去噪声信息。

     d. 代码实现

```
uchar adaptiveMedianFilter(cv::Mat &img, int row, int col, int kernelSize, int
maxSize)
{
    std::vector<uchar> pixels;
    for (int y = -kernelSize / 2;y <= kernelSize / 2;y++)
    {
        for (int x = -kernelSize / 2;x <= kernelSize / 2;x++)
        {
            pixels.push_back(img.at<uchar>(row + y, col + x));
        }
    }

    sort(pixels.begin(), pixels.end());  // 排序找中值

    auto min = pixels[0];
    auto max = pixels[kernelSize*kernelSize - 1];
    auto med = pixels[kernelSize*kernelSize / 2];
    auto zxy = img.at<uchar>(row, col);
    if (med > min && med < max)  // 实现比较的过程
    {
        // to B
        if (zxy > min && zxy < max)
            return zxy;
        else
            return med;
    }
    else
    {
        kernelSize += 2;
        if (kernelSize <= maxSize)
            return adaptiveMedianFilter(img, row, col, kernelSize, maxSize);// 增大
窗口尺寸，继续A过程。
        else
            return med;
    }
}
```

iv. Nonlocal means

a. 原理

a. 非局部均值滤波则使用当前滤波点的邻域块与矩形窗口内其它点的邻域块的相似度来计算权重，相似度越大则权重越大。

b. 非局部均值滤波算法就是使用MSE来计算两个邻域块的相似度。邻域块的行列相同，假设均为m行n列，那么MSE的计算如下式，其中A(i,j)为点A邻域块中的点(i,j)的像素值，B(i,j)为点B邻域块中的相同位置点的像素值：$MES(A,B)=\frac{1}{mn} \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(A(i,j)-B(i,j)^2$

c. 点A的滤波值由搜索窗口内所有点的像素值加权平均得到：$NL(A)=\sum w(A,B)*I(B)$其中w(Am,B)是高斯权重，I(B)是B点的灰度值。NLmeans采用的类似于高斯权重的形式

d. 计算$w(A,B),w(A,B) = \frac{1}{sum} \sum e^{-\frac{MES(A,B)}{h^2}}$, 而比例sum为$sum =\sum e^{-\frac{MES(A,B)}{h^2}} $,以实现归一化的作用。

b. 朴素代码

```
for (int j = boardSize; j < boardSize + rows; j++)
    {
        uchar *dst_p = dst.ptr<uchar>(j - boardSize);
        for (int i = boardSize; i < boardSize + cols; i++)
        {
            Mat patchA = boardSrc(Range(j - halfKernelSize, j + halfKernelSize),
Range(i - halfKernelSize, i + halfKernelSize));
            double w = 0;
            double p = 0;
            double sumw = 0;

            for (int sr = -halfSearchSize; sr <= halfSearchSize; sr++)   //在搜索框
内滑动
            {
                uchar *boardSrc_p = boardSrc.ptr<uchar>(j + sr);
                for (int sc = -halfSearchSize; sc <= halfSearchSize; sc++)
                {
                    Mat patchB = boardSrc(Range(j + sr - halfKernelSize, j + sr +
halfKernelSize), Range(i + sc - halfKernelSize, i + sc + halfKernelSize));
                    float d2 = MSE_block(patchA, patchB);

                    w = exp(-d2 / h2);
                    p += boardSrc_p[i + sc] * w;
                    sumw += w;
                }
            }

            dst_p[i - boardSize] = saturate_cast<uchar>(p / sumw);

        }
    }
    return dst;
```

朴素代码的时间负责度为$O(n^4)$基本运行lena的图片要20s以上，效率可以说非常低。

c. 优化--基于积分图的优化方式

a. 积分图是一种用于加速图像处理的数据结构，它可以在常数时间内计算出给定区域内像素值的和。类似二维前缀和的思路计算积分图：对原始图像进行积分图计算，得到每个像素位置的累积和。这可以通过迭代计算得到，从左上角到右下角遍历图像，并在每个像素位置计算该位置及其左上角相邻位置的累积和。

b. 对于每个像素，定义一个固定大小的窗口，然后使用积分图快速计算窗口内像素值的和，以代替传统的遍历计算，计算窗口内像素值的均值。

c. 代码实现积分图

```
  void integralImgSqDiff(Mat src, Mat &dst, int Ds, int t1, int t2, int m1, int
n1)
  {
      //计算图像A与图像B的差值图C
      Mat Dist2 = src(Range(Ds, src.rows - Ds), Range(Ds, src.cols - Ds)) -
src(Range(Ds + t1, src.rows - Ds + t1), Range(Ds + t2, src.cols - Ds + t2));
      float *Dist2_data;
      for (int i = 0; i < m1; i++)
      {
          Dist2_data = Dist2.ptr<float>(i);
          for (int j = 0; j < n1; j++)
          {
              Dist2_data[j] *= Dist2_data[j];  //计算图像C的平方图D
          }
      }
      integral(Dist2, dst, CV_32F); //计算图像D的积分图
  }
```

d. 后续的计算MSE以及给点的综合权重的过程与朴素做法类似。运行上述代码同样对496*472的Lena图像去噪，结果如下图所示，耗时由原来的20秒左右减少为1.6秒左右，所以加速效果还是非常显著的。

v. 四种滤波器的对比

a. 均值滤波器：
   a. 缺点:可能会模糊图像细节，对于椒盐噪声等异常噪声效果差。
b. 中值滤波器
   a. 缺点:可能减少图像细节，计算复杂度较高
c. 自适应中值滤波器
   a. 缺点：计算复杂度较高，可能需要更多的参数调整。
d. NLM
e. 缺点：计算复杂度高，特别是对于大尺寸图像和大窗口尺寸的情况。需要高效的相似性度量方法。

- 高斯实验结果展示

i.

ii. 分析：
   a. 中值滤波器和均值滤波器对于高斯噪声的处理可以处理到大部分的噪声，但是会使图片过度虚化。并且两个图片还是存在噪声，背景噪声还是很大，只是将噪声细小的噪声，模糊化。
   b. 自适应中值滤波器表现的效果最差，因为高斯噪声，并不是和椒盐噪声是最大或者最小的噪声，所以再使用自适应中值滤波器时，并不能消除噪声。所以处理的图像和原图像基本差不多
   c. 非局部均值滤波器 (NLM)：效果最好，因为它充分考虑像素之间的相似性度量来降低高斯噪声。因为它能够有效去除高斯噪声而又保留图像的细节。

- 椒盐实验结果展示

i.

ii. 分析：
   a. 加入很多的椒盐噪声。
   b. 代码

```
 void saltNoise(cv::Mat img, int n)
{
    int x, y;
    for (int i = 0;i < n / 2;i++)
    {
        x = std::rand() % img.cols;
        y = std::rand() % img.rows;
        if (img.type() == CV_8UC1)
        {
            img.at<uchar>(y, x) = 255;
        }
        else if (img.type() == CV_8UC3)
        {
            img.at<cv::Vec3b>(y, x)[0] = 255;
            img.at<cv::Vec3b>(y, x)[1] = 255;
            img.at<cv::Vec3b>(y, x)[2] = 255;
        }
    }
}

//椒噪声
void pepperNoise(cv::Mat img, int n)
{
    int x, y;
    for (int i = 0;i < n / 2;i++)
    {
        x = std::rand() % img.cols;
        y = std::rand() % img.rows;
        if (img.type() == CV_8UC1)
        {
            img.at<uchar>(y, x) = 0;
        }
        else if (img.type() == CV_8UC3)
        {
            img.at<cv::Vec3b>(y, x)[0] = 0;
            img.at<cv::Vec3b>(y, x)[1] = 0;
            img.at<cv::Vec3b>(y, x)[2] = 0;
        }
    }
}
```

椒盐噪声我是用的是加入0和255两个极端灰度值，分别表示椒噪声，盐噪声。如图2，会让图片产生很多的噪声干扰，图片已经模糊不清了。

iii. 中值滤波器，自适应中值滤波器

a. 中值滤波器：每一次都取中值，有利于避开值为0或者255的椒盐噪声，但是，由于噪声太多了，即使使用中值也会避免不了取到中值，所以在图像上也会有黑色的点表示噪声。

b. 自适应中值滤波器是所有滤波器中表现最好的一个，自适应中值滤波器可以根据每个像素的邻域条件自动调整滤波窗口的大小，以更好地适应图像的特性和噪声水平。这使得它在不同区域具有不同噪声水平的图像中表现出色。并且对于边缘的处理也有较好的特性。虽然在边缘出有很多的噪声因为边缘会有很多只为0的补全不分，就导致边缘的噪声不能被删除。

iv. 均值滤波器

a. 均值滤波器是将是将所有窗口内的值取平均，对于椒盐噪声的处理和高斯噪声类似。但是会使图片过度虚化。并且两个图片还是存在噪声，背景噪声还是很大，只是将噪声细小的噪声，模糊化。

b. NLmeans滤波器，依旧能让图片有较高的不失真性。但还是存在少量的噪声误差，不能做到和ADP一样的完美程度。