实验一 图像处理与可视化编程

1.1 任务二：交互式程序设计

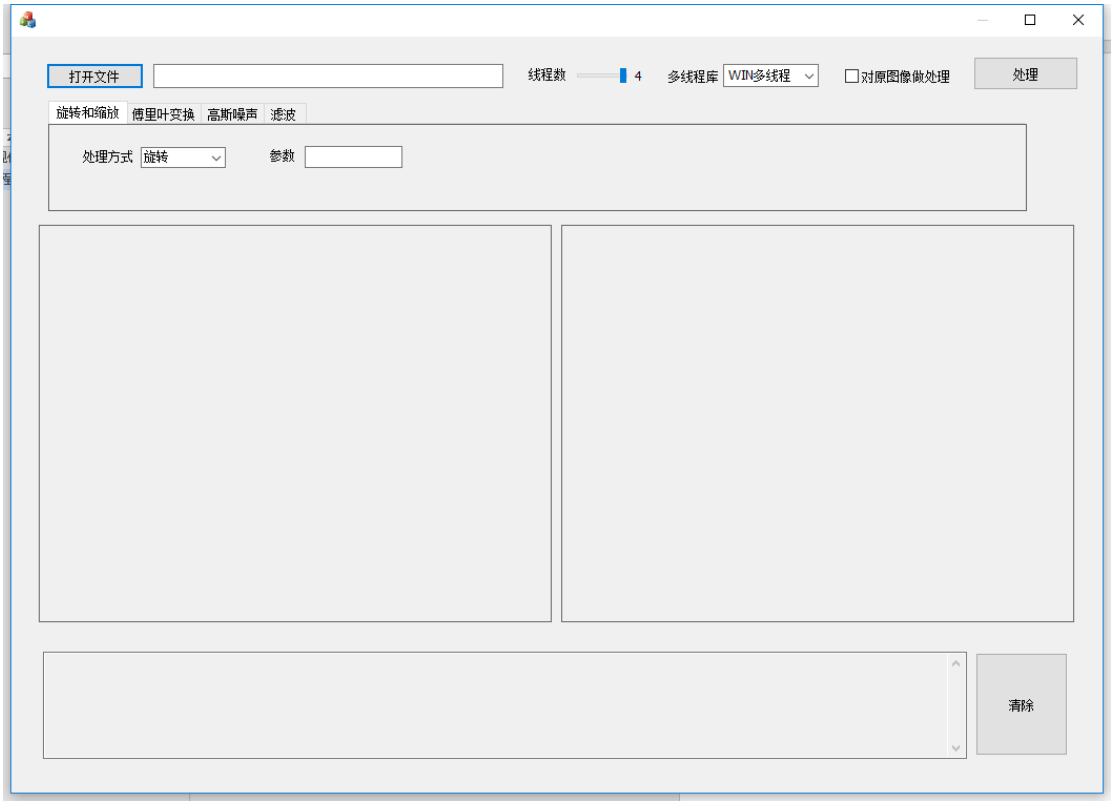


图 1 图形界面

1.1.1 Tab 页

在主窗口中使用一个 CTabControl 组件，然后为每个功能都添加一个新的窗口，然后指定窗口的父级窗口为主窗口下的 CTabControl 组件。

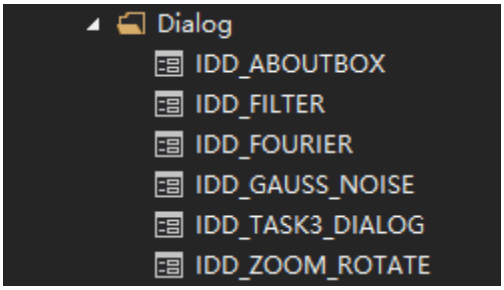


图 2 主窗口（IDD_TASK3_DIALOG）及各个功能窗口

在主窗口类中将各个功能窗口类作为自己的数据成员，包装各个功能窗口类，实现窗口间的数据传递。

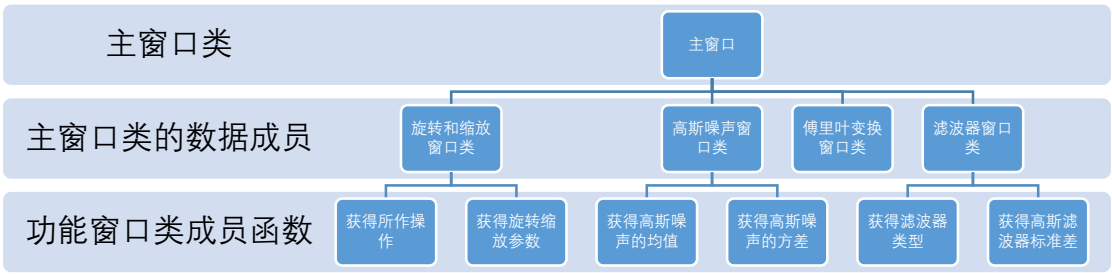


图 3 各个窗口之间的结构

1.1.2 图像显示区域

图像显示区域实现了图像的自适应，在显示图片之前，先将图片的宽高和图像显示区域的宽高进行对比，如果发现，图片能在区域内完全容纳，就直接显示原大小，如果至少有一条边（宽/高）超出了图片显示区的对应宽/高，则将图片缩小至图像显示区能容纳的大小进行显示。

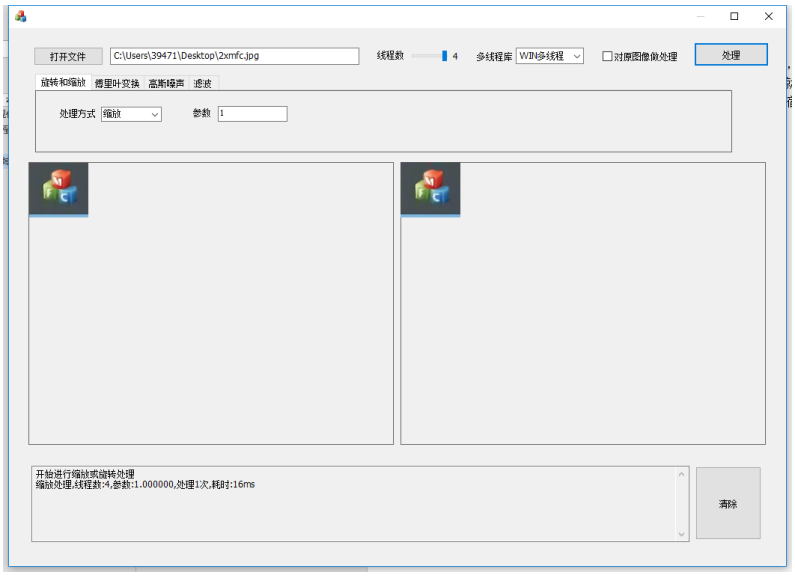


图 4 原始大小的图片

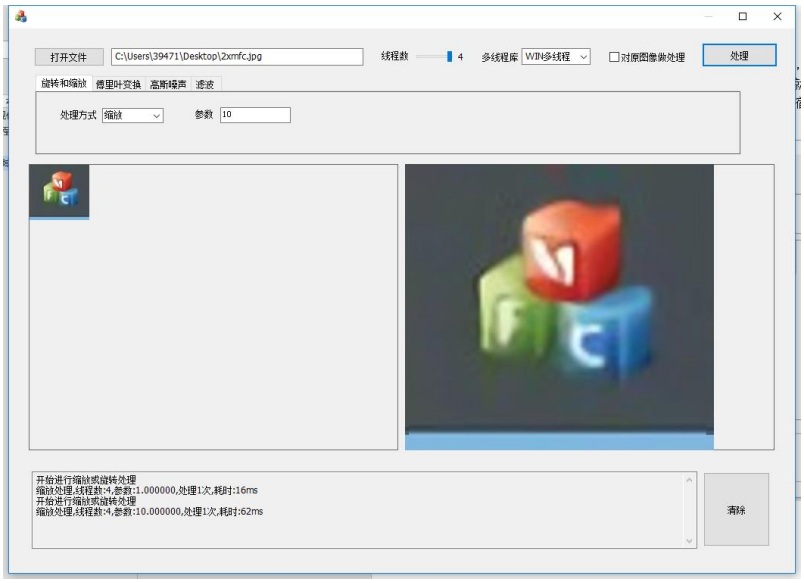


图 5 放大 10 倍后自适应缩放显示的图片

1.1.3 参数输出区域

参数输出区域被设置成透明底色，且为只读区域，可以复制、滚动、选择，右侧提供按钮进行清空。

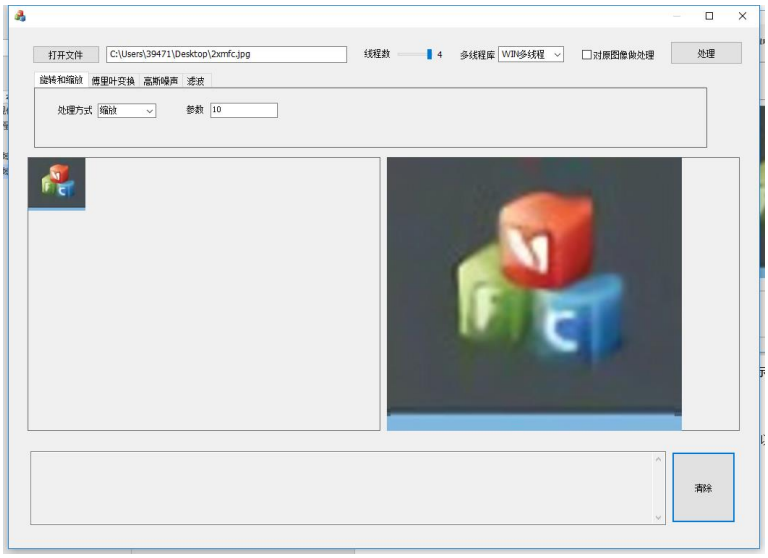


图 6 清空后的参数输出区

1.2 任务三：图像处理功能及其多线程实现

1.2.1 采用双立方插值的图像旋转和缩放

选择的双三次插值的权重公式如下，其中$a=-0.5$，x 为插入点与临近列/行的水平距离或者竖直距离（x>0）

$$\begin{cases} (2+a)*x^3-(3+a)*x^2+1 & x\le 1 \\ a*x^3-5*a*x^2+8*a*x-4*a & 1<x<2 \\ 0 & 2\le x \end{cases}$$

公式 1 双三次插值的权重公式

取得权值矩阵之后，进行加权，得到插入点的像素，写入图像即可完成插值。

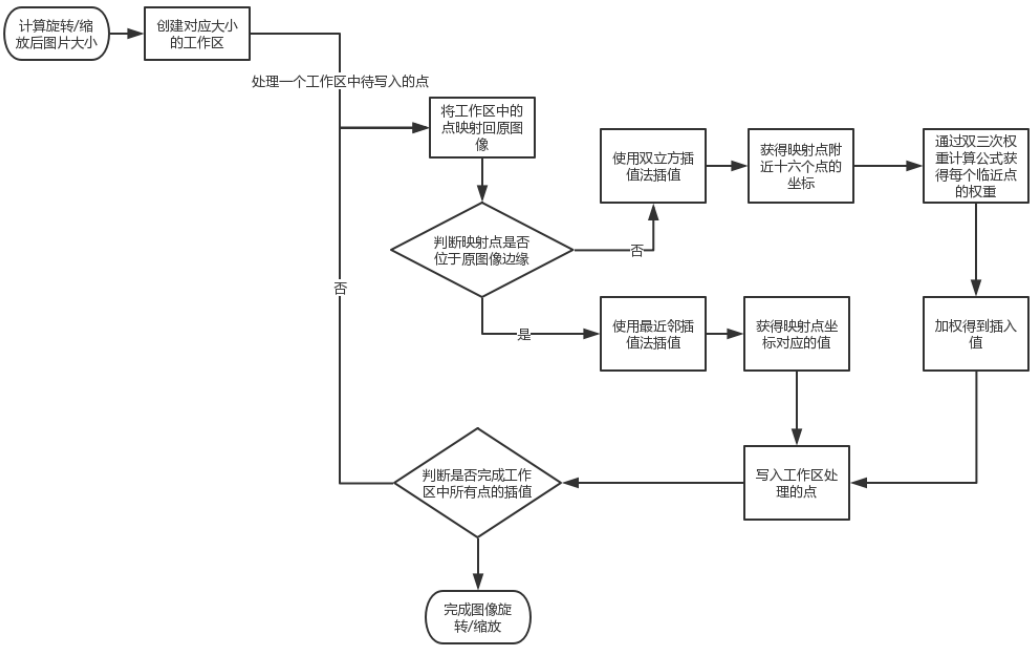


图 7 旋转/缩放程序流程图

(a) 图像缩放

图像缩放所进行的反向映射比较简单，设缩放系数 f（$0 < f < +\infty$）,工作区图像的某一点坐标为(x,y)，则对应原图上的点(x`,y`)满足公式 2：

$$\begin{cases} x` = x/f \\ y` = y/f \end{cases}$$

公式 2

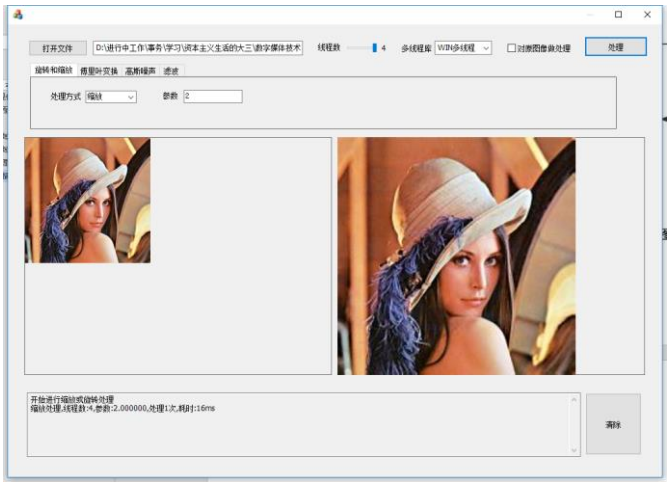


图 8 图像放大两倍的结果

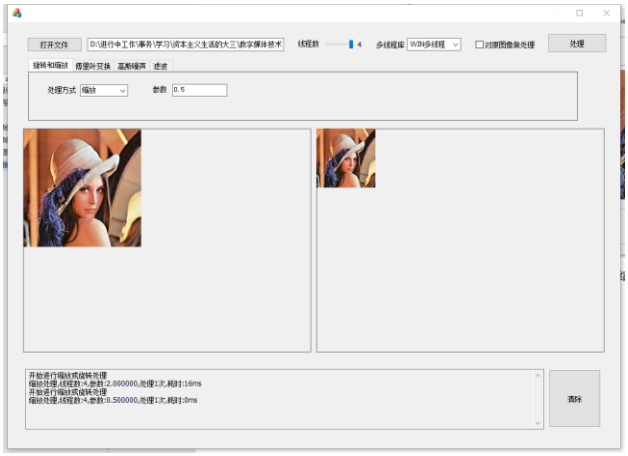


图 9 图像缩小为原来的一半的结果

(b) 图像旋转

由于工作区的图形一定是正的四边形，因此图形经过旋转之后也会改变图像大小，因此也需要对其进行反向映射，设用户选择顺时针旋转的角度为 θ，则图像旋转所使用的反向映射分为两步，

(i) 将工作区域的点 A(x,y)绕工作区图像中心点(xc`,yc`)逆时针旋转 θ，得到点 A 的旋转前位置 A0(x0,y0)，其中 x0，y0 满足公式 3：

$$\begin{cases} x0 = (x - xc`) * \cos\theta + (y - yc`) * \sin\theta \\ y0 = -(x - xc`) * sin\theta + (y - yc`) * cos\theta \end{cases}$$

公式 3

(ii) 得到旋转前位置相对中心点的坐标(x0-xc,y0-yc)，计算得到原图图像中心点(xc`,yc`),在原图中找到映射的点 A`(x`,y`)，其中 x`,y`满足公式 4：

$$\begin{cases} x` = xc` + x0 - xc \\ y` = yc` + y0 + yc \end{cases}$$

公式 4

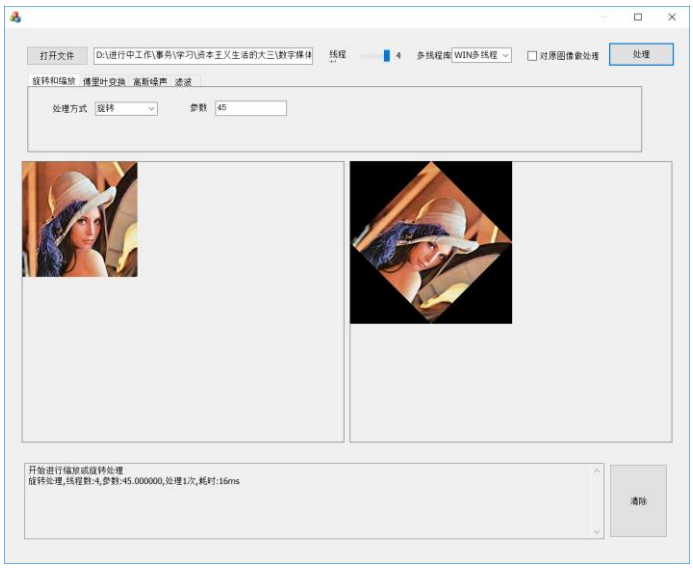


图 10 图像旋转结果

1.2.2 图像傅里叶变换

(i) 图像中心化

为了便于观察，我们需要在开始进行傅里叶变换之前对原始图片进行中心化。

即对处理得到的每个像素点均乘以-1^{x+y},x 和 y 分别是该像素对应的列数和行数。

(ii) 离散二维傅里叶变换公式

利用方便计算机编程的离散二维傅里叶

$$\mathrm{F(u,v)} = \sum_{y=0}^{N-1}\sum_{x=0}^{M-1} f(x,y)e^{i2\pi(\frac{ux}{M}+\frac{vy}{N})}$$

公式 5 离散二维傅里叶变换公式

然后，利用欧拉公式展开，得到方便计算机编程的离散二维傅里叶

$$\mathrm{F(u,v)} = \sum_{y=0}^{N-1}\sum_{x=0}^{M-1} f(x,y) * (\cos(-2\pi\,(\frac{\mathrm{ux}}{\mathrm{M}}+\frac{\mathrm{vy}}{\mathrm{N}})) + \mathrm{isin}(-2\pi(\frac{ux}{M}+\frac{vy}{N})))$$

公式 6 方便计算机编程的离散二维傅里叶

然后，因为实现过程中涉及到了复数的运算，于是自己封装了一个 ComplexNumber 类，重载了各种运算符方便运算。

```
struct ComplexNumber
{
    double realpart;
    double imagecypart;

    ComplexNumber(double realpart, double imagecypart);
    ComplexNumber() { realpart = 0.0; imagecypart = 0.0; };
    void setValue(double realpart, double imagecypart);


    inline ComplexNumber operator +(const ComplexNumber &x){ ... }

    inline ComplexNumber operator -(const ComplexNumber &x){ ... }

    inline ComplexNumber operator *(const ComplexNumber &x){ ... }

    inline ComplexNumber operator *(const int x){ ... }

    inline ComplexNumber operator *(const double x){ ... }

    inline ComplexNumber operator /(const ComplexNumber &x){ ... }

    inline double getNorm(){ ... }
};
```

图 11 自己封装的复数类

(iii) 对数变换

同样，为了便于观察，我们对中心化后的图像进行对数变化，用于增强图像细节。

即对处理得到的每个像素点均乘以$1 + 15\log|F(u,v)|$,u 和 v 分别是频谱图中的列数和行数。

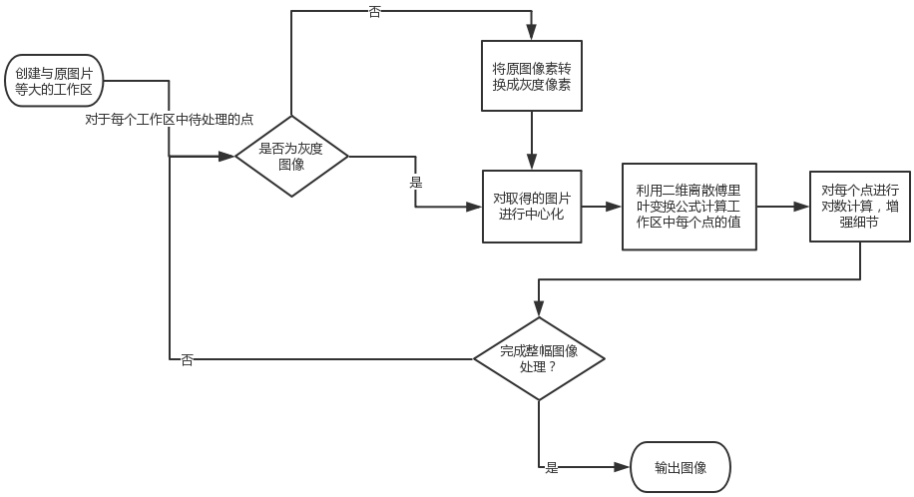


图 12 傅里叶变换程序流程图

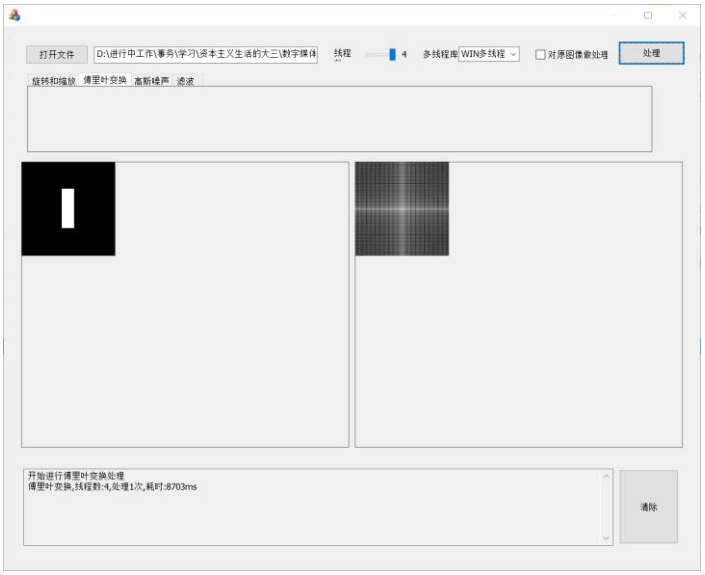


图 13 傅里叶变换结果

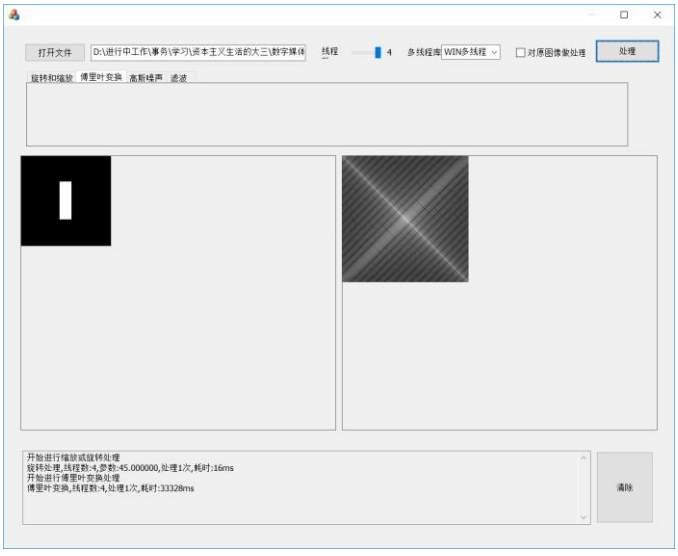


图 14 与旋转功能联动，得到的傅里叶变换结果

1.2.3 高斯噪声添加

(i) Box-Muller 公式生成基本噪声

使用 rand()/RAND_MAX 产生两个 0~1 的随机数，然后利用 Box-Muller 公式获得均值为 0，方差为 σ 的高斯分布的一个噪声值。

$$Z_0 = \sigma\sqrt{-2\ln U_1}\cos(2\pi U_2)$$
$$Z_0 = \sigma\sqrt{-2\ln U_1}\sin(2\pi U_2)$$

公式 7 Box-Muller 公式

(ii) 噪声合成

获得基本噪声后，我们将用户指定的均值 μ 加上，得到最后的高斯噪声值。

(iii) 噪声添加

高斯噪声是一种加性噪声，则原图上对应点的像素值加上噪声值即可生成高斯噪声。

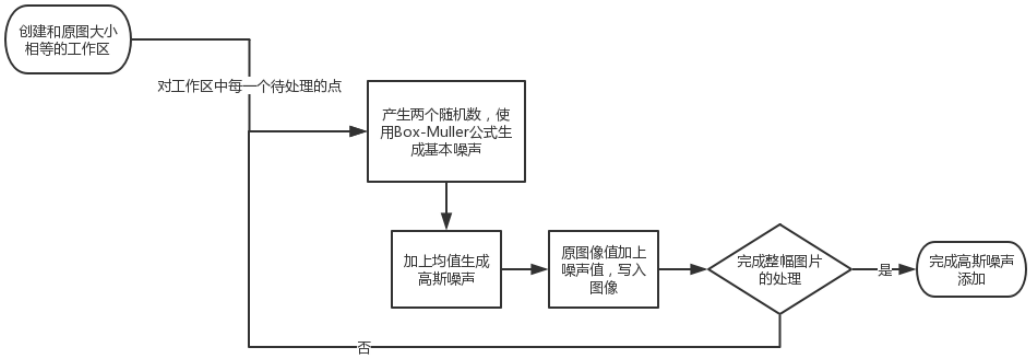


图 15 高斯噪声程序流程图

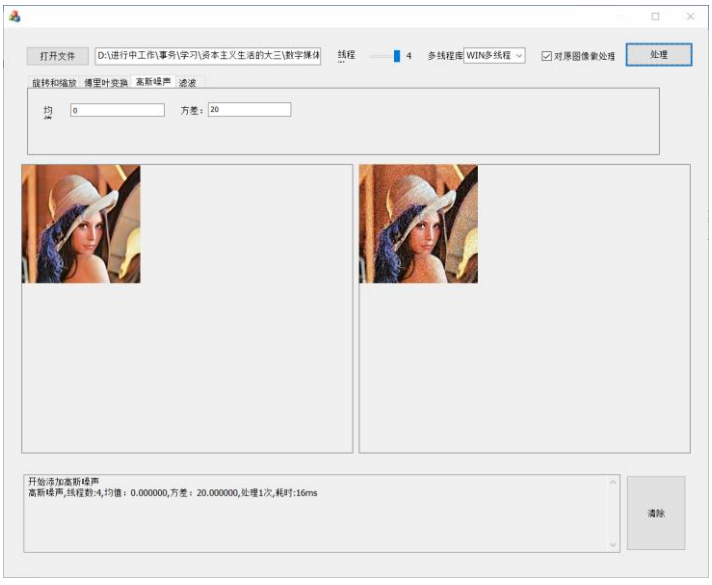


图 16 高斯噪声添加结果

1.2.4 滤波器

图 17 均值为 0，方差为 20 的高斯噪声处理后的图片

(a) 算术均值滤波器

算术均值滤波器的实现原理比较简单，就是取一个固定大小的矩形区域，将里面所有像素点取平均，这样可以去掉噪声的影响

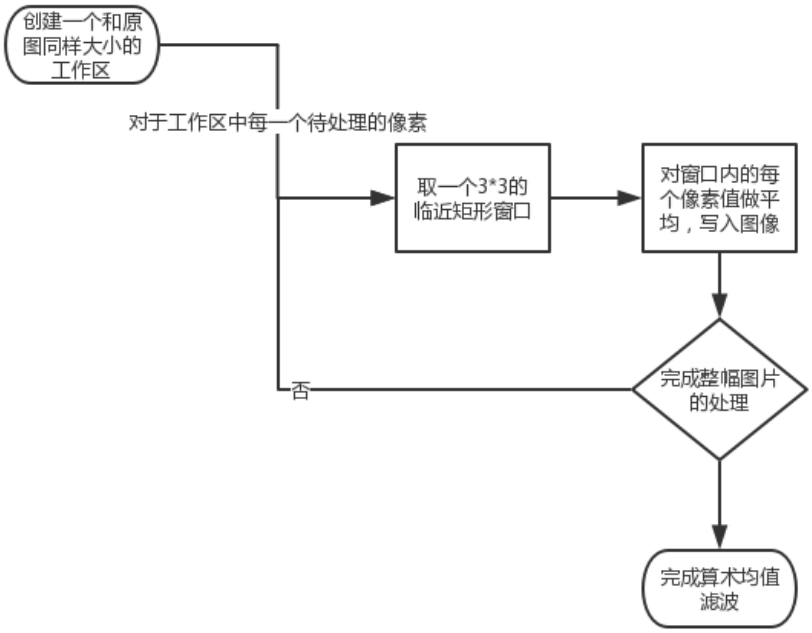


图 18 算术均值滤波器程序流程图

图 19 使用算术均值滤波后的图像

(b) 高斯滤波器

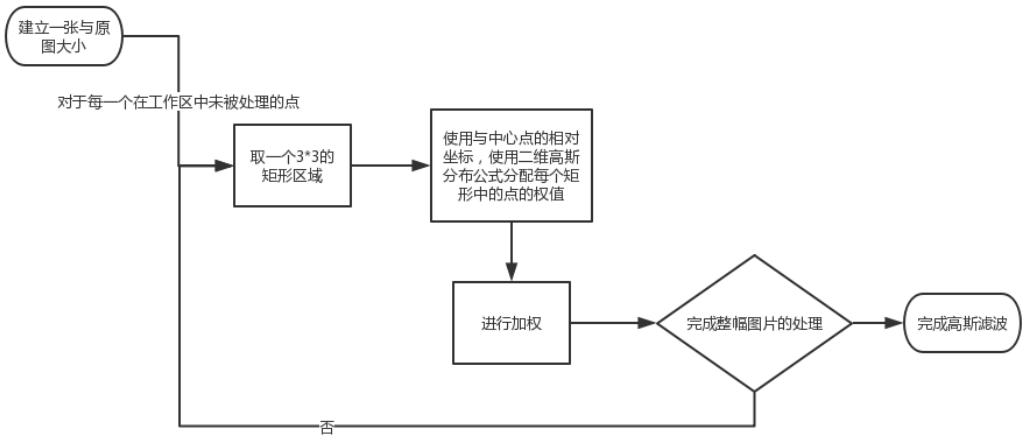
(i) 二维高斯分布函数

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

公式 8 二维高斯分布函数

利用高斯分布在所选择的矩形区域生成满足对应方差的，均值为 0 的高斯分布权值

(ii) 加权处理，写入图像



公式 9 高斯滤波器程序流程图

图 20 使用方差为 20 的高斯滤波后的图像

(c) 维纳滤波器

(i) 图片总体方差计算

因为是自适应的维纳滤波，因此先要遍历原图像的每个点，求出整体图像的方差用于，后续公式的计算。

(ii) 维纳滤波公式

$$dst(x,y) = \mu + \frac{\max(0, \sigma^2 - v^2)}{\max(\sigma^2, v^2)}(src(x,y) - \mu)$$

公式 10 维纳滤波公式

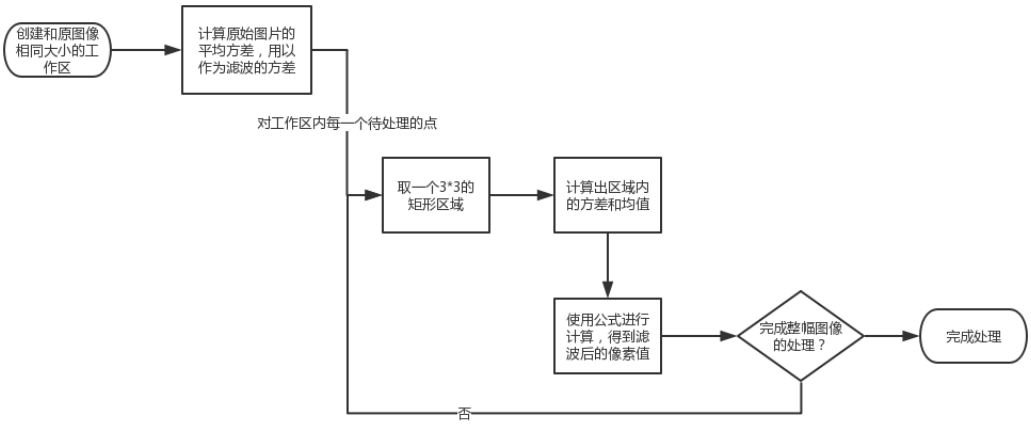


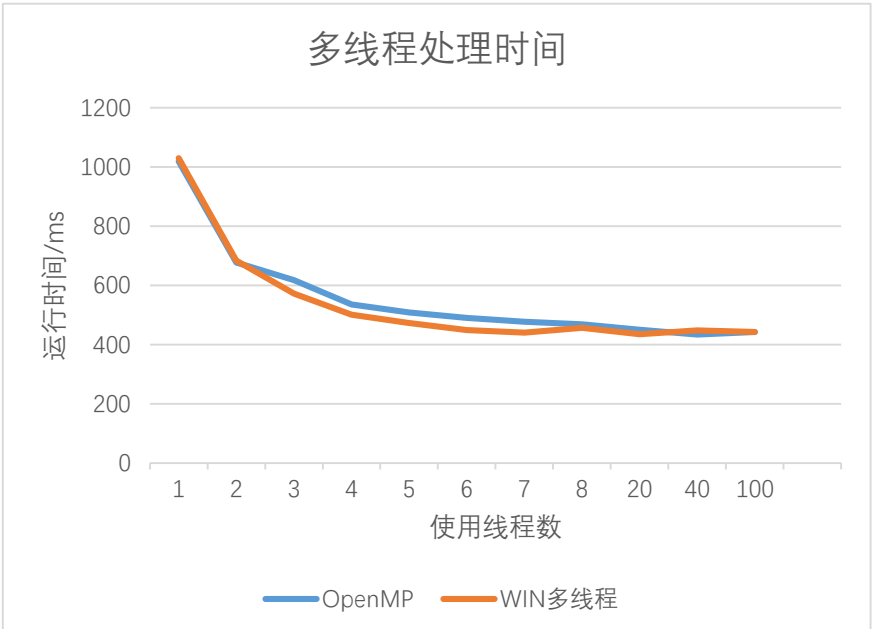
图 21 自适应维纳滤波器程序流程图

图 22 自适应维纳滤波之后的图像

综合上述对比，可以看到，维纳滤波和高斯滤波的效果较好，算术均值滤波效果一般。

1.3 多线程和单线程效率提高分析

由于本次实现的功能都是线程之间没有依赖的，独立的，因此对于测试多线程和单线程



效率这个问题与使用什么功能来测试无关，因此，本人希望对一个功能做更多的测试，来分析多线程的效率问题

	OpenMP(ms)	WINAFX(ms)

1	1019	1029
2	677	683
3	618	572
4	535	501
5	508	473
6	490	449
7	477	440
8	468	456
20	450	435
40	434	448
100	443	442

表格 1 图像旋转中各个参数的效率对比（每一个数据都是处理 15 次以上去除极端值取平均的结果）

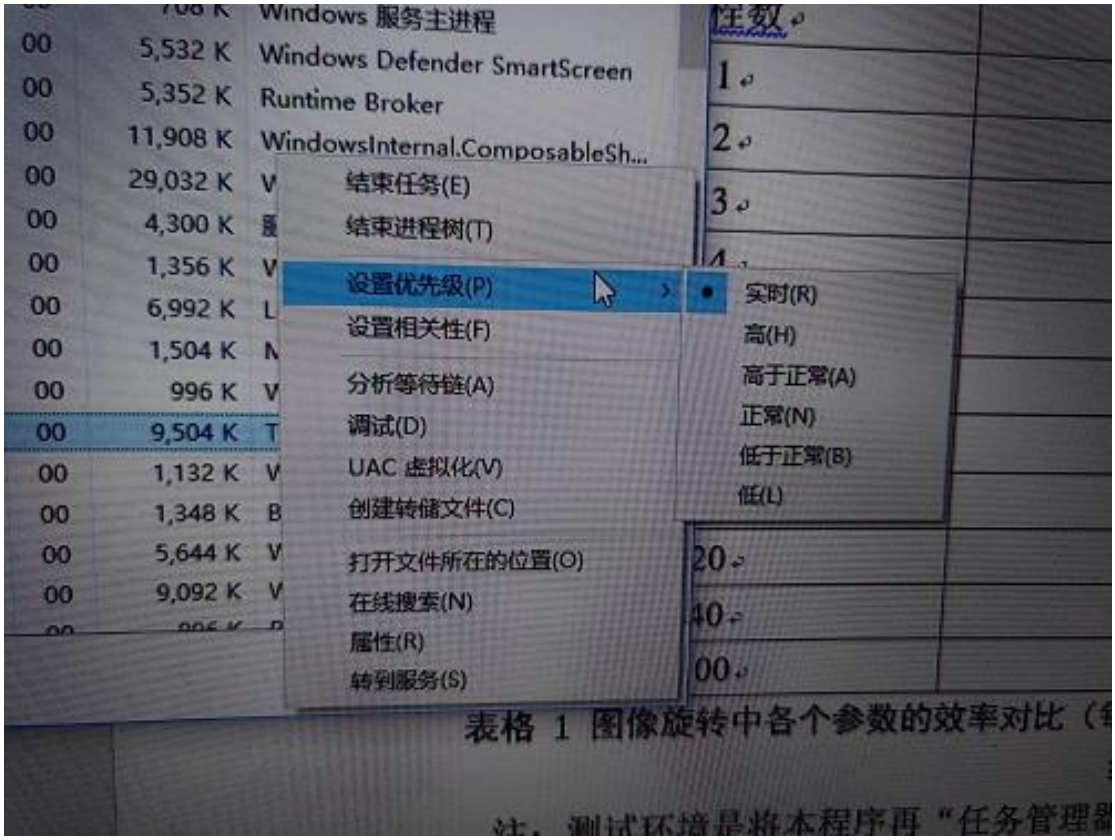


图 23 测试环境是将本程序再“任务管理器”中设置成为最高优先级

从表格 1 中我们可以看到，OpenMP 和 WINAFX 的性能相差不远（在考虑计时器计时误差的情况下）

本机的硬件环境是 2 核心 4 个逻辑处理器的 CPU，可以看到在线程数量是 1 变成 2 的时候，时间几乎是原来的一半，这是非常理想的时间缩小。线程数从 2 变成 4 的时候，有将近 20%的效率提升，这是因为 4 个逻辑处理器都得到了很好的运用，也是很理想的。在那之后的效率提升就非常的小了，从 4 变换到 8，即两倍逻辑处理器数量的线程的时候，效率提升了 10%。然后，后面的线程数即使增加很大幅度，效率的提升也不是很大，大概只有 5%之后，应该是因为逼近了极限速度。由此看来，分配的线程数越多，的确是最后的效率比较高。