CS219 Project 3 : BMP Image Processing

目录

1. **项目介绍**
    - 1.1 基本信息
    - 1.2 使用说明
2. **BMP读入**
    - 2.1 BMP文件格式
    - 2.2 BMP文件的C语言读写实现
    - 2.3 32位BMP文件的兼容读入
    - 2.4 8位BMP文件的兼容读入
3. **用户输入管理**
    - 3.1 输入解析和操作结构体
    - 3.2 内存管理和鲁棒性
4. **图片处理操作实现**
    - 4.1 亮度和HSV调整
        - 4.1.1 RGB, HSV, HSL
        - 4.1.2 HSV和RGB的转换
        - 4.1.3 亮度实现：alpha遮罩
        - 4.1.4 常数时间优化：函数指针分离HSV操作
        - 4.1.5 SIMD优化：定值亮度调节
    - 4.2 裁剪
    - 4.3 水平翻转
    - 4.4 旋转
        - 4.4.1 难点和思路
        - 4.4.2 简单旋转实现
        - 4.4.3 双线性插值进行清晰度优化
    - 4.5 缩放
        - 4.5.1 纯放大：双线性插值
        - 4.5.2 纯缩小：区域采样
        - 4.5.3 任意比例放缩
    - 4.6 混合
        - 4.6.1 基本实现
        - 4.6.2 SIMD加速

    - 4.7 锐化
        - 4.7.1 卷积核
        - 4.7.2 初步实现
        - 4.7.3 复杂度优化：边缘不处理
        - 4.7.4 效果优化：预模糊降噪
    - 4.8 模糊
        - 4.8.1 实现方案
        - 4.8.2 初步实现
        - 4.8.3 复杂度优化：卷积核拆分
    - 4.9 滤镜
        - 4.9.1 矩阵"实现"非线性变换
    - 4.10 二值化
        - 4.10.1 二值化类型
        - 4.10.2 函数指针统一管理
    - 4.11 操作平均运行时间
5. **AI使用**
    - 5.1 代码框架初步实现
    - 5.2 帮助文档和目录生成
6. **总结**
7. **参考**

1 项目介绍

1.1 基本信息

本项目实现了基于C语言且不调用任何外部库的BMP图像处理程序，可以完成BMP图像的读入、修改和写出。

- 在读入部分，程序通过舍弃alpha通道来实现32位BMP图像向24位图像的转换；通过将像素索引映射到调色板并扩展为RGB空间来实现8位BMP图像向24位图像的转换。因而程序可以读入32位/24位/8位的图像，提升了兼容性。
- 在用户输入部分，有清晰易懂的输入规则，可以通过执行 `-help` 进行详细查询。此外对输入越界或未定义操作有全面的检测。
- 在修改部分，程序实现了大部分常见的图像修改功能：
    - HSV：将RGB通道转换成HSV通道并以此修改图片的色温、饱和度和明度
    - 亮度：通过PhotoShop采用的alpha遮罩的亮度算法，优化了亮度变化的视觉效果，减少了纯RGB值增加导致的饱和度变化。此外也允许仅使用定值增加的方式来调整亮度，并通过SIMD加速计算
    - 缩放：实现了基于双线性插值的图幅纯放大和基于加权采样的图幅纯缩小，并通过“先缩小后放大”的方式间接实现所有可能的图幅变换，减少因图幅放缩导致的数据丢失问题
    - 常用：实现了基本的裁剪、旋转和镜像翻转功能，可以指定裁剪的具体位置和大小以及旋转的角度

- 混合：通过采用已实现的图片放缩，可以实现任意两张不同尺寸的图片的混合，融合的占比可通过参数调整，并在均等混合的条件下通过SIMD优化速度
- 锐化：基于预先定义的多种卷积核，实现图片锐化，可以通过调整迭代次数来提高锐化程度
- 模糊：基于优化后的横竖一维高斯模糊实现图像的模糊，可以通过参数调整高斯核的大小和模糊的迭代次数
- 滤镜：基于预先定义的多种变换矩阵，实现一些常见的滤镜：复古、灰白和反色。
- 二值化：实现了类似OpenCV的二值化，包括五种：二进制阈值化、反二进制阈值化、截断二值化、反阈值化为0和阈值化为0

1.2 使用说明

编译好文件后，执行 `<PATH>/bmpdit -help` 来查看以下内容：

```
BMP图像处理工具
支持8/24/32位无压缩BMP文件格式

用法: ./bmpedit1 -i 输入文件.bmp -o 输出文件.bmp -op <操作> [参数]
选项:
  -i <文件名>                    输入BMP文件路径 (必填)
  -o <文件名>                    输出BMP文件路径 (必填)
  -op <操作> <参数>              要执行的操作及参数
                                   可通过多次输入-op进行多种操作,并按输入顺序执行

可用操作:
  addL <值>                      调整亮度 (范围: -100到100)
  addH <值>                      调整色调 (范围: 任意整数)
  addS <值>                      调整饱和度 (范围: -100到100)
  addV <值>                      调整明度 (范围: -100到100)
  blend <路径> <比例>           与另一图像混合 (比例: 0.0-1.0)
  rescale <宽> <高>              缩放图像到新尺寸 (宽高为正整数)
  cut <x偏移> <y偏移> <新宽> <新高>
                                 裁剪图像 (以左下角为原点)
  rotate <角度>                  旋转图像 (角度范围: 任意浮点数)
  flip                            水平翻转图像
  sharp <类型> <迭代次数>      锐化图像 (类型: 0-3, 迭代次数: 正整数)
                                      类型说明:
                                      0-无效果
                                      1-拉普拉斯锐化
                                      2-索贝尔锐化
                                      3-掩模锐化
  blur <比例> <迭代次数>       高斯模糊 (比例: 0.0-1.0,对应高斯核1-41宽度, 迭代次数: 正整数)
  filter <类型>                  应用滤镜 (类型: 0-3)
                                      类型说明:
                                      0-原图
                                      1-复古
                                      2-反色
                                      3-灰度
  binarize <阈值> <类型>       二值化 (阈值: 0-255, 类型: 0-4)
                                      类型说明:
```

```
0-普通二值化
1-反色二值化
2-截断二值化
3-零处理二值化
4-反色零处理
```

```
示例：
  调整亮度并应用复古滤镜：
  ./bmpedit -i in.bmp -o out.bmp -op addL 50 -op filter 1
  混合两张图片：
  ./bmpedit -i in.bmp -o out.bmp -op blend other.bmp 0.5
  旋转并缩放图像：
  ./bmpedit -i in.bmp -o out.bmp -op rotate 90 -op rescale 800 600
  高斯模糊迭代三次：
  ./bmpedit -i in.bmp -o out.bmp -op blur 0.5 3
  得到竖直镜像：
  ./bmpedit -i in.bmp -o out.bmp -op flip -op rotate 180
```

2 BMP读入

2.1 BMP文件格式

由于BMP文件对我而言并不常用，唯一一次使用是利用软件将生成的字模转化成16进制导入BMP文件，之后直接将文件的内容转化为二维数组实现像素级别的VGA数字和字母显示。因此，BMP在我的首印象中是可以很方便地直接根据文件的16进制内容直接转化为对应像素的颜色值的一种图片格式。

事实也的确如此，BMP是Bitmap(位图)的缩写，是一种独立于显示器的位图数字图像文件格式。图像通常保存的颜色深度有1位(0或1，仅黑白色)、4位(预定义的16种颜色)、8位(预定义256种颜色)、16位(初步使用RGB通道，分别占用5位)、24位(RGB通道分别占用8位)和32位(RGBA四个通道，分别占用8位，A为alpha，代表图像对应像素点的透明度)，本次项目主要专注在24位BMP图像的处理上。

不论BMP文件的颜色深度是多少，其存储格式都是相似的：

1. 前14字节为位图文件头(BMP header)：合法的BMP文件中，前两字节一定是`42 4D`,对应ASCII码中的`BM`，表明这是一个BMP文件。再往后4字节是整个文件的大小，单位为字节，因此最大可以是2^32字节=4GB大小。再之后4字节是保留位，通常为0。最后4字节是位图数据的实际存储地址，以相对于文件开头的字节偏移量格式存储，在实际操作时很常用。
2. 后面是DIB头，包含图像的详细信息，在屏幕上显示图像将会使用这些信息。现在计算机最通用的DIB头是`BITMAPINFOHEADER`类，占用40字节。可以通过检测色深来确定是24位还是32位或者其他位的图片；而存储信息中最重要和常用的是位图宽度和位图高度，分别用于对图片进行像素行和列的操作时确定对应字节在文件中的位置。完整解释见下图，来自Wikipedia的BMP介绍。

偏移量	大小（字节）	用途
0Eh	4	该头结构的大小（40字节）
12h	4	位图宽度，单位为像素（有符号整数）
16h	4	位图高度，单位为像素（有符号整数）
1Ah	2	色彩平面数；只有1为有效值
1Ch	2	每个像素所占位数，即图像的色深。典型值为1、4、8、16、24和32
1Eh	4	所使用的压缩方法，可取值见下表。
22h	4	图像大小。指原始位图数据的大小（详见后文），与文件大小不是同一个概念。
26h	4	图像的横向分辨率，单位为像素每米（有符号整数）
2Ah	4	图像的纵向分辨率，单位为像素每米（有符号整数）
2Eh	4	调色板的颜色数，为0时表示颜色数为默认的$2^{色深}$个
32h	4	重要颜色数，为0时表示所有颜色都是重要的；通常不使用本项

3. 再之后就全部是图片的像素数据了，位图文件头中最后四位的偏移量指向的就是这个位置，在24位图中，每一行的数据字节数实际是`位图宽度 * 3`，因为每个像素要从左到右分别存储BGR的三个字节(不是R、G、B顺序)，而32位中则是`位图宽度 * 4`，对应每个像素的BGRA四个字节。对于24位的来讲，最后像素数据部分的实际字节数是`位图高度 * 位图宽度 * 3`。需要注意的是，**图片的实际存储顺序是从左下角开始，逐行向上存储的**。

实际上我们可以利用指令打开一个bmp文件的来看看它的前54位内容：

```
od -Ax -tx1 -N 54 LenaRGB.bmp
```

```
0000000    42  4d  38  00  0c  00  00  00  00  00  36  00  00  00  28  00
0000010    00  00  00  02  00  00  00  02  00  00  01  00  18  00  00  00
0000020    00  00  00  00  00  00  81  b8  00  00  81  b8  00  00  00  00
0000030    00  00  00  00  00  00
0000036
```

可以看到以下信息：位图文件头中前2字节为`42 4D`，说明这是一个合法BMP文件，之后四个字节是`38 00 0c 00`，由于是小端序存储，所以实际是`0x000c038`个字节，即786,488字节，查看文件大小后发现是786KB，数据符合。位图14字节的最后4字节是0x00000036，代表偏移量是54字节，正好是BMPheader+DIBheader的大小。DIB header的第5到13字节为`00020000 00020000`表明文件是`512x512`分辨率大小。偏移两个字节之后的两个字节为`00 18`表明这是24位色深的图片。

2.2 BMP文件的C语言读写实现

了解了以上信息，我们就可以实现基于C语言的BMP文件读写了。由于一开始不好下手，所以让Deepseek生成了一份对应的读写框架。让我们来分析一下整个结构和读写过程：

```c
#pragma pack(push, 1) // Ensure no padding in structs
typedef struct {
    char signature[2];  //must be "BM"
    int file_size;      //size of the file
    short reserved1;    //0
    short reserved2;    //0
    int data_offset;    //the offset of the real data comparing to the header
} BMPHeader;
```

```
typedef struct {
    int header_size;
    int width;
    int height;
    short planes;
    short bits_per_pixel;
    int compression;
    int image_size;
    int x_pixels_per_meter;
    int y_pixels_per_meter;
    int colors_used;
    int important_colors;
} DIBHeader;
#pragma pack(pop)

typedef struct {
    BMPHeader file_header;
    DIBHeader dib_header;
    unsigned char* pixel_data;
} BMPImage;
```

首先，我们可以看到3个struct结构，分别是BMPImage，包含位图文件头(`file_header`)、DIB文件头(`dib_header`)和像素数据(`pixel_data`)三个成员，这和我们先前对BMP文件格式的分析是一致的；其中，两个文件头又分别是一个结构体，对应的成员也是实际上这两部分所包含的信息。

然而，我们可以看到一个特别的部分，那就是在两个header前后存在`#pragma pack(push, 1)`和`#pragma pack(pop)`。根据Deepseek的解释：“这是用于控制结构体（struct）内存对齐的编译器指令。在 BMP 文件解析的上下文中，它们的核心作用是确保结构体成员紧密排列，消除编译器自动填充的额外字节。”

也就是说，根据我们所学，C/C++中的结构体所占用的内存不一定是所有成员的数据类型的大小的总和，而是会为了对齐而在部分成员变量后面增加一些空间保证整体的4字节对齐，比如struct{int a; char b;} 是5个字节，而为了对齐，整个struct实际会占用8个字节。但是由于BMP文件内部文件头的字节是紧密排列的，不会出现小于4字节的信息之后自动填充4字节。比如BMP header的前两个字节是字符BM，之后直接就是一个4字节的`file size`，整体是6字节；如果直接读入，可能会出现编译器自动扩展到2+2+4=8字节的情况。

不做紧密排列的结果是什么，我们还要看读写是具体怎么实现的：

```
FILE* file = fopen(filename, "rb");
CHECK_FILE(file, filename, "Could not open file");

BMPImage* image = (BMPImage*)malloc(sizeof(BMPImage));
CHECK_NULL(image, "Memory allocation failed");

// Read headers
fread(&image->file_header, sizeof(BMPHeader), 1, file);
fread(&image->dib_header, sizeof(DIBHeader), 1, file);

// Verify BMP format
if (image->file_header.signature[0] != 'B' || image->file_header.signature[1] != 'M') {
  fprintf(stderr, "Error: Not a valid BMP file\n");
  fclose(file);
```

```
  free(image);
  exit(EXIT_FAILURE);
}
// Compute actual row size
int row_size = image->dib_header.width * 3;
int padding = (4 - (row_size % 4)) % 4;
int actual_row_size = row_size + padding;

image->pixel_data = (unsigned char*)malloc(actual_row_size * image->dib_header.height);
CHECK_NULL(image->pixel_data, "Memory allocation for pixel data failed");

fseek(file, image->file_header.data_offset, SEEK_SET);
fread(image->pixel_data, actual_row_size * image->dib_header.height, 1, file);
fclose(file);
return image;
```

可以看到，read headers部分，实际读入的字节数是根据`sizeof(BMPHeader)`和`sizeof(DIBHeader)`来完成的，也就是说，如果我们不做紧密排列，读入的字节数量就会出现误差，导致读入出错。

但是，和文件头部分的紧密排列不一样，在像素数据部分，文件是会对每一行的数据做整体的字节填充的。在Compute actual row size 部分，出现了一个`padding`的计算，这一部分表明，每一行的字节数一定要是4的倍数，如果不足4，那么BMP会填充几个字节进行对齐。最终要给`pixel_data`分配的大小就是实际填充后的行的字节数乘以图像高度。

写出的部分相似，就是将读入的文件读入内存操作逆转成从内存写进文件，不再赘述。

2.3 32位BMP文件的兼容读入

了解了读入的实现原理后，我本想找一个BMP文件做测试，然而却发现是32位色深的，所以我由此想到是否可以通过去除alpha通道的方式来兼容读入32位的BMP文件呢?

```
int src_row_size = image->dib_header.width * 4; // for 32bit
int src_padding = (4 - (src_row_size % 4)) % 4;
int src_actual_row_size = src_row_size + src_padding;

int dst_row_size = image->dib_header.width * 3; // for 24bit
int dst_padding = (4 - (dst_row_size % 4)) % 4;
int dst_actual_row_size = dst_row_size + dst_padding;

for (int y = 0; y < image->dib_header.height; y++) {
    fread(src_row, src_actual_row_size, 1, file);
        for (int x = 0; x < image->dib_header.width; x++) {
        int src_pos = x * 4;
        int dst_pos = y * dst_actual_row_size + x * 3;

        image->pixel_data[dst_pos]     = src_row[src_pos];     // B
        image->pixel_data[dst_pos + 1] = src_row[src_pos + 1]; // G
        image->pixel_data[dst_pos + 2] = src_row[src_pos + 2]; // R
    }
}
```

以上是实现后的关键循环，由于不论是多少位色深的图片，读入前后的分辨率不应该改变，所以x和y的for循环终止条件分别为分辨率的宽高；对于x的部分，由于32位和24位色深每个像素的字节数不同，所以我们分别计算得到原图的字节位置和新图的字节位置，然后对弈每一个像素，仅取前三个字节(BGR)进行复制，忽略第四个字节(alpha通道)。就可以实现文件色深的转化。

当然还有一些细节文件头信息需要更改，不然会导致图片损坏无法打开：

```
image->dib_header.bits_per_pixel = 24;//原为32
image->dib_header.header_size = 40;
image->dib_header.compression = 0;//有的压缩信息不为0
image->dib_header.image_size = dst_actual_row_size * image->dib_header.height;
image->file_header.file_size = sizeof(BMPHeader) + sizeof(DIBHeader) + image-
>dib_header.image_size;
image->file_header.data_offset = sizeof(BMPHeader) + sizeof(DIBHeader);
```

2.4 8位BMP文件的兼容读入

此外，下载更多经典常用图片时，发现更多的其实是8位色深的图片。8位图片和32/24位色深图片的最大不同之处在于，在文件头的54字节结束之后，并没有立即存储像素的值，而是通过1024字节存储了一个调色板，为了字节对齐，每4个字节存储一种RGB颜色"BGR0"，总共256种颜色，而像素数据部分，每个像素存储的8位数字并不是RGB颜色，而是对应调色板的索引。因此，我们的实现思路就在于，使用三个指针分别指向：原图调色板起始位置、原图像素数据起始位置和新图像素数据起始位置

```
unsigned char* palette = (unsigned char*)malloc(1024);
unsigned char* src_row = (unsigned char*)malloc(width);
image->pixel_data = (unsigned char*)malloc(dst_actual_row_size * height);
```

随后，遍历新图的每一个像素，根据原图索引找到调色板的颜色，赋值到新图的RGB通道：

```
//in for cycle
int dst_pos = y * dst_actual_row_size + x * 3;
unsigned char idx = src_row[x] * 4;

image->pixel_data[dst_pos]     = palette[idx];    // B
image->pixel_data[dst_pos + 1] = palette[idx+1];  // G
image->pixel_data[dst_pos + 2] = palette[idx+2];  // R
```

这样就完成了8位色深BMP图的兼容读入。

3 用户输入管理

3.1 输入解析和操作结构体

用户通过执行文件时附带的各项参数进行图片的处理，由此，程序通过主函数的`int argc, char* argv[]`对输入内容进行管理。利用宏定义`COMPARE(argv[i],"name",parameter number)`来比较输入是否合法确保读入范围不超出限制。

```
#define COMPARE(NAME1,NAME2,NUM) strcmp(NAME1, NAME2) == 0 && i + NUM < argc
```

```
for (int i = 1; i < argc; i++)
{
    if (COMPARE(argv[i],"-i",1))
        input_path = argv[++i];
    else if (COMPARE(argv[i],"-o",1))
        output_path = argv[++i];
    else if (COMPARE(argv[i],"-op",1))
    {
        char * op_name = argv[++i];
        Operation op = {0};

        if (COMPARE(op_name,"addL",1))
        {
            op.type = OP_ADD_L;
            op.int1 = atoi(argv[++i]); // Lightness
        }
        ......
    }
}
```

操作部分，用两个结构体，分别存储一个操作的内容，`OperationType`统一对操作类型进行管理，`Operation`包含整个操作的全部内容，包括操作类型和操作参数。这里的操作参数实际使用方式类似于寄存器，根据输入参数数量和类型的需要在对应的“寄存器”中存储，并在函数实现中取出。

```
typedef enum {
    OP_ADD_L,       // 亮度
    OP_ADD_H,       // 色调
    OP_ADD_S,       // 饱和度
    OP_ADD_V,       // 明度
    OP_BLEND,       // 混合
    OP_RESCALE,     // 缩放
    OP_CUT,         // 裁剪
    OP_ROTATE,      // 旋转
    OP_FLIP,        // 镜像
    OP_SHARP,       // 锐化
    OP_BLUR,        // 模糊
    OP_FILTER,      // 滤镜
    OP_BINARIZE     // 二值化
} OperationType;

typedef struct {
    OperationType type;
    int int1; // parameters fucntions like registers
    int int2;
    float float1;
    int int3;
    int int4;
    char* blend_image_path;
} Operation;

// in main
```

```
Operation *operations = (Operation*)malloc(argc * sizeof(Operation));
```

最终，通过一个operation数组实现队列，进行统一管理。在for循环中对读入的操作顺序执行。

```
for (int i = 0; i < op_count; i++) {
    Operation op = operations[i];
    switch (op.type) {
        case OP_ADD_L:
          ...
          break;
        ......
        default:
            ...
     }
}
```

3.2 鲁棒性

在操作内容读入后进行合法性检查，包括输入输出路径是否存在、是否进行操作、读入失败后的报错提醒和完整及时的内存释放。

```
#define CHECK_NULL(ptr, msg) if ((ptr) == NULL) { fprintf(stderr, "Error: %s\n", msg);
exit(EXIT_FAILURE); }
#define CHECK_FILE(file, filename, msg) if ((file) == NULL) { fprintf(stderr, "Error: %s
'%s'\n", msg, filename); exit(EXIT_FAILURE); }

// in main
Operation *operations = (Operation*)malloc(argc * sizeof(Operation));
CHECK_NULL(operations, "Memory reallocation for operation queue failed");

if (!input_path || !output_path || op_count == 0) {
    fprintf(stderr, "Error: Missing required arguments\n");
    print_help();
    free(operations);
    return EXIT_FAILURE;
}

BMPImage *image = read_bmp(input_path);
if (!image) {
    fprintf(stderr, "Error: Failed to read input image\n");
    free(operations);
    return EXIT_FAILURE;
}
```

在读入部分虽然没有做参数的范围检查，但是在每个函数内部都有对所有参数的完整检查和错误或者警告提醒，有的函数在警告后不进行任何图片操作，有的越界操作会进行饱和运算

```
//in images_blend()
CHECK_NULL(image1,"Error: NULL image can't be processed")
CHECK_NULL(image2,"Error: NULL image can't be processed")
if(ratio < 0) ratio = 0;
if(ratio > 1) ratio = 1;
//in images_sharpen
if(type>= sizeof(KERNALS) / sizeof(KERNALS[0]) || type < 0)type = 0;//KERNALS 0为无操作核
```

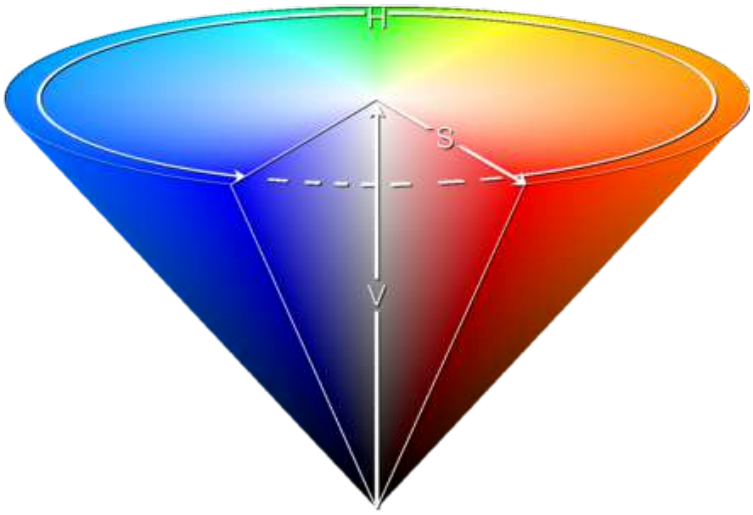
4 图片处理操作实现

以下内容为各个功能的介绍，包括原理、代码实现、时间/清晰度优化。时间优化后的测试统一在 **4.11** 部分进行比较。

4.1 亮度和HSV调整

4.1.1 RGB, HSV, HSL

对于24位BMP文件，像素默认以RGB三通道的方式进行存储。虽然RGB以三原色光的形式通过强度的线性组合形成各种颜色，但是在具体图像调整的时候却难以直接通过调整三个通道的大小的方式实现适合于人眼的常见调节，换句话说，这三种颜色的取值与所生成的颜色之间的联系并不直观。因此，我们还可以采用HSV或HSL两种颜色模型。这两种表示法是RGB模型的一种非线性变换，试图做到比基于笛卡尔坐标系几何结构的RGB模型更加直观。这两种模型中，H均表示色相(Hue)，S均表示饱和度(Saturation)，V表示明度(Value)，L表示亮度(Lightness)，这几个参数最终将颜色描述为圆柱坐标系内的点。



我们可以用圆锥体来视觉化这个坐标系。以HSV模型为例，色相H最终的范围是0～360度，绕着锥体的外圆周变化，每120度经过RGB三种颜色的一个区间，这个数值越接近0或者360(轮转一圈回来)越趋近于红色，越接近180越趋近于青色。而饱和度S最终范围是0～1，表示为圆锥表面到中心轴的距离，这个距离越大色彩的强度/纯度越大，越小色彩的强度/纯度越小。明度V的范围也是0～1，表示圆锥锥顶到垂直正下方底面圆心的距离，这个距离越大颜色越趋近于白色，越小越趋近于黑色。

然而，由于饱和度的定义略有争议，所以仅明度V改变时虽然饱和度S没有变，但究竟有没有改变真正的饱和度仍存在分歧，由此又有了HSL模型。不过我在实际实现时，最终采用的亮度变化算法并没有采用这两种模型的V和L，详见4.1.3

4.1.2 HSV和RGB的转换

且不论亮度变化的问题，仅HSV可以实现色温和饱和度的变化这一特征，我们就可以采用这个模型进行图片的修改。我们可以通过简单的计算公式实现从RGB到HSV模型的转换：

$$s = \begin{cases} 0, & \text{if } max = 0 \\ \frac{max-min}{max} = 1 - \frac{min}{max}, & \text{otherwise} \end{cases}$$

$$v = max$$

$$h = \begin{cases} 0^\circ & \text{if } max = min \\ 60^\circ \times \frac{g-b}{max-min} + 0^\circ, & \text{if } max = r \text{ and } g \geq b \\ 60^\circ \times \frac{g-b}{max-min} + 360^\circ, & \text{if } max = r \text{ and } g < b \\ 60^\circ \times \frac{b-r}{max-min} + 120^\circ, & \text{if } max = g \\ 60^\circ \times \frac{r-g}{max-min} + 240^\circ, & \text{if } max = b \end{cases}$$

以及HSV到RGB的逆变换：

$$h_i = \left\lfloor \frac{h}{60} \right\rfloor$$

$$f = \frac{h}{60} - h_i$$

$$p = v \times (1 - s)$$

$$q = v \times (1 - f \times s)$$

$$t = v \times (1 - (1 - f) \times s)$$

对于每个颜色向量 (r, g, b),

$$(r, g, b) = \begin{cases} (v, t, p), & \text{if } h_i = 0 \\ (q, v, p), & \text{if } h_i = 1 \\ (p, v, t), & \text{if } h_i = 2 \\ (p, q, v), & \text{if } h_i = 3 \\ (t, p, v), & \text{if } h_i = 4 \\ (v, p, q), & \text{if } h_i = 5 \end{cases}$$

代码实现基本类似，详见代码部分的`cvtRGB_HSV` 和 `cvtHSV_RGB`函数。最终在函数内，遍历每个像素进行操作：

```
AdjustFunc func = get_adjust_func(type);

for (int y = 0; y < col_size; y++) {
    unsigned char* cur_row = image->pixel_data + y * actual_row_size;
    for (int x = 0; x < row_size; x++) {
        func(&cur_row[x * 3],param1,param2);
    }
}
```

对于HSVL，均输入循环并统一操作：

```
float H = 0,S = 0,V = 0;
int R = 0,G = 0,B = 0;

/* adjust brightness */
B = cur_row[x]   * delt + offset;
G = cur_row[x+1] * delt + offset;
R = cur_row[x+2] * delt + offset;

/* adjust HSV */
cvtRGB_HSV(&R,&G,&B,&H,&S,&V);
H += H_v;
S += S_v;
V += V_v;
cvtHSV_RGB(&H,&S,&V,&R,&G,&B);

cur_row[x]   = (unsigned char)B;
cur_row[x+1] = (unsigned char)G;
cur_row[x+2] = (unsigned char)R;
```

先将RGB转化为HSV，再根据传入的参数修改HSV的值，最后再转化回RGB颜色。为了用户输入的友好性，读入时S和V默认传入-100～100的整数，在函数内部再除以100，这样还避免了小数位过多的参数传入，提高性能。

由于在转化函数内部已经完成了诸如`*R = (int)fmaxf(0, fminf(255, *R));`的操作进行范围上下限的规范，所以最后赋值的时候没有做多余的范围检测。最终实现效果如下：

对于青椒部分：左上角是原图，显示了一些红色和绿色的辣椒，右上角是Hue降低30的结果，左下角是Hue增加30的结果，右下角是Hue增加180度，也就是取反色的效果。可以看到这其中原本红色的辣椒色彩变化最明显，原本在0度附近，H增加30后趋近橙色，减少30后趋近粉紫色，取反色后接近青色，这和Hue色彩轮盘的结果是相合的。

对于猴子部分：左上角是原图，右上角是饱和度S降低40的结果，左下角是饱和度增加40的结果，右下角是饱和度归零的结果。大体是符合人眼视觉效果的。

4.1.3 亮度实现：alpha遮罩

亮度实现部分，我试图用HSV的V进行亮度变化，得到如下图片中的第一列结果。然而实际效果上，以人的肉眼来看似乎饱和度也发生了变化，可以看到，在V增加时，图片中的颜色明显变得更鲜艳了，但是根据实际经验，亮度增加并不会导致颜色变得更加鲜艳或者暗淡(比如肉眼观察手机屏幕并加减亮度，最终屏幕的显示不会发生鲜艳程度的变化)。

顺便，原图如下，是阳台山接近山顶的地方拍的：

于是，我们采用最直接的对RGB通道增加固定值的方法，得到第二列结果。第二列结果看起来效果更好，但是存在一个关键问题，那就是当亮度增加或者减少值较大时，原本有颜色差异的部分就会变得无法区分。举个例子，比如两个像素点X ={110,105,195} 和Y ={195,107,110}，如果增加或减少亮度规范化到-100～100，那么，增加60%的亮度相当于对每个通道都增加60 * 255 / 100 = 153 的值，截断多余的部分后生成的新点为 _X={255,255,255} _Y={255,255,255} ，也就是均为纯白色。这样算出来就会出现第二列最下面图片的情况，天空整体偏蓝偏白，三个通道的值都比较大，在亮度增加后，部分区域就会变得难以区分细节。

到了这里，我想到可以借助一些实现了亮度变化的工具，看看他们的亮度实现效果，最后我打开PhotoShop 2021版，对图片打开HSV的图层调整，得到第三列结果。可以看到，这个结果就肉眼的效果而言更符合预期，虽然PhotoShop标记的是HSV调整，但是明度的变化却和我生成的HSV的明度变化效果不同。在网上一番搜索和尝试之后，我找到了一个结果上接近PS效果的算法：给定参数alpha，范围在 [-1,1]，根据参数具体值，有：

- 如果alpha大于0，相当于利用一个白色遮罩层合成 RGB = RGB * （1 - alpha） + 255 * alpha；
- 如果alpha小于0，相当于利用一个黑色遮罩层合成 RGB=RGB * （1+alpha） + 0 * alpha；

最终效果为第四列：

上图中，每一组实验都生成了四张图片，分别是按百分比减少60%亮度、减少30%亮度、增加30%亮度和增加60%亮度。

4.1.4 常数时间优化：函数指针分离HSV操作

由于所有HSV和L的操作均可以通过相似的代码实现，所以为了避免函数内做实际操作类型的判断，并且对没有修改的参数也进行多余计算，可以优化时间，选择统一用函数指针进行管理。对于每一个操作设置一个对应的可内联函数，如addV：

```
inline void adjust_H(unsigned char* pixel, float H_v, float none) {
    float H, S, V;
    int R = pixel[2], G = pixel[1], B = pixel[0];

    cvtRGB_HSV(&R, &G, &B, &H, &S, &V);
    H += H_v;
    cvtHSV_RGB(&H, &S, &V, &R, &G, &B);

    pixel[0] = (unsigned char)B;
    pixel[1] = (unsigned char)G;
    pixel[2] = (unsigned char)R;
}
```

内循环直接调用函数指针：

```
for (int x = 0; x < row_size; x++) {
    func(&cur_row[x * 3],param1,param2);
}
```

4.1.5 SIMD优化：定值亮度调节

不过，我们可以发现，在亮度并没有调整到极大或者极小的情况下，仍然可以用对各通道的纯定值加减法来实现亮度改变，因此，我也同样预留了一个对应的函数，通过宏定义`_SIMPLE_LIGHT_ADD_`决定是否开启

```
case OP_ADD_L:
#ifdef _SIMPLE_LIGHT_ADD_
    image_simp_light_adjust(result,op.int1);
#else
    image_color_adjust(result,op.int1,0);
#endif
    break;
```

函数内部，由于定值加法的运算较为简单，无需复杂的类型转换，而且内存连续，因此可以尝试使用SIMD进行算术加速：如果`_NEON_`宏定义已启用，则会在每一行一次性加载16个字节进行计算，如果没有，则会直接采用普通的逐字节运算。通过同一个`int x`管理地址，即达成了启用SIMD情况下图像横向字节数不与16对齐情况下的剩余标量运算，也可以在不启用SIMD情况下直接作为普通运算使用。

```
L_val = (L_val < -255 ? -255 : (L_val > 255 ? 255 : L_val));
uint8_t abs_L = (uint8_t)(L_val < 0 ? -L_val : L_val);
LightFunc Func = get_light_func(L_val >= 0 ? 0 : 1); //根据不同的L_val的值决定使用加法还是减法

#pragma omp parallel for schedule(guided)
for (int y = 0; y < height_size; y++)
{
    unsigned char* cur_row = image->pixel_data + y * actual_row_size;
    int x = 0;
#ifdef _NEON_
        uint8x16_t v_offset = vdupq_n_u8(abs_L);
        for (; x + 15 < row_size; x += 16) {
            uint8x16_t v_pixels = vld1q_u8(cur_row + x);
            Func(&v_pixels,v_offset);//函数指针调用加法或者减法器
            vst1q_u8(cur_row + x, v_pixels);
        }
#endif
    for (; x < row_size; x++) {
        int new_val = cur_row[x] + L_val;
        cur_row[x] = (unsigned char)(new_val < 0 ? 0 : (new_val > 255 ? 255 : new_val));

    }
}
```

然而，对于alpha遮罩方法和HSV计算，却不能用SIMD加速。因为BMP文件的内容是以uint8数据结构存储，所以与float类型进行混合运算时，由于数据结构的位宽不同，转换较为繁琐，最终效果测试后发现不如直接运算。

```
pixel[i] = (unsigned char)(pixel[i] * delt + offset);
```

4.2 裁剪

4.2.1 代码实现和参数检查

裁剪功能实现较为简单，只需要找到新图片的位置和大小，再进行数据复制即可。

裁剪功能共有四个参数：

1. 相对于图片左下角的横向偏移量
2. 相对于图片左下角的竖直偏移量
3. 新图片的宽度
4. 新图片的高度

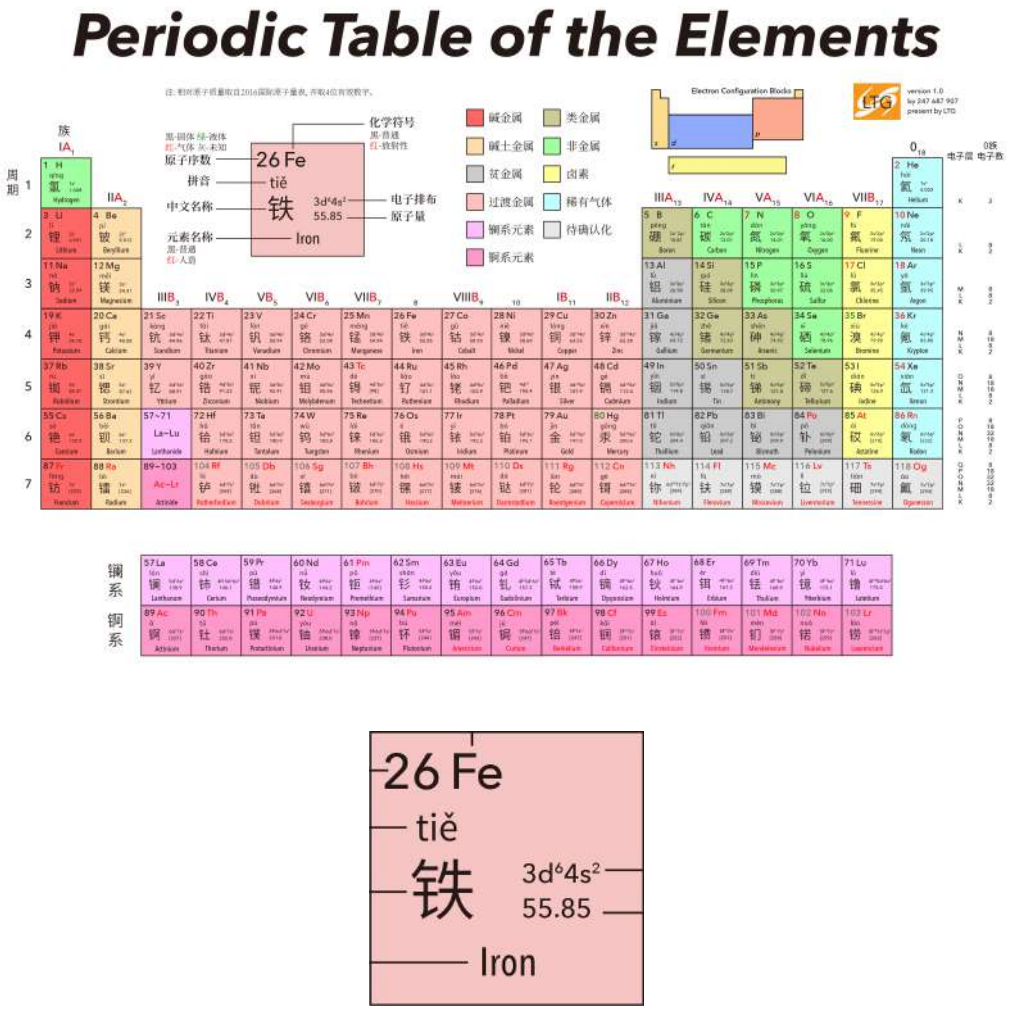
进行了全面的参数检查：

```
//保证图像存在，偏移量大于等于0，新宽高大于0
if (!image || offset_x < 0 || offset_y < 0 || new_width <= 0 || new_height <= 0) {
        printf("Invalid parameters for image cutting\n");
        return NULL;
    }
//保证偏移量不超过原图尺寸
if(offset_x >= old_width || offset_y >= height_size){
    printf("Offset too big for image cutting\n");
    return NULL;
}
//保证新的宽高范围仍处在原图范围内
if(new_width > old_width - offset_x){
    new_width = old_width - offset_x;
}
if(new_height > height_size - offset_y){
    new_height =  height_size - offset_y;
}
```

使用`memcpy`加速内存复制，减少逐元素赋值拖慢速度：

```
unsigned char* src_data = image->pixel_data + offset_y * actual_row_size + offset_x * 3;
unsigned char* dst_data = new_image->pixel_data;
for (int y = 0; y < new_height; y++) {
    memcpy(dst_data + y * new_actual_row_size, src_data + y *
actual_row_size,new_row_size);
}
```

示例：将元素周期表图片(4961×3508)上的一部分切割下来：




```
./build/bmpedit1y -i pic_in/R-C.bmp -o report/ETableIron.bmp -op cut 1274 2140 536 536
```

相当于以左下角为原点，以坐标系（1274，2140）点为新图左下角，向右上方向切割536×536大小的区域出来。得到标识铁元素的方框

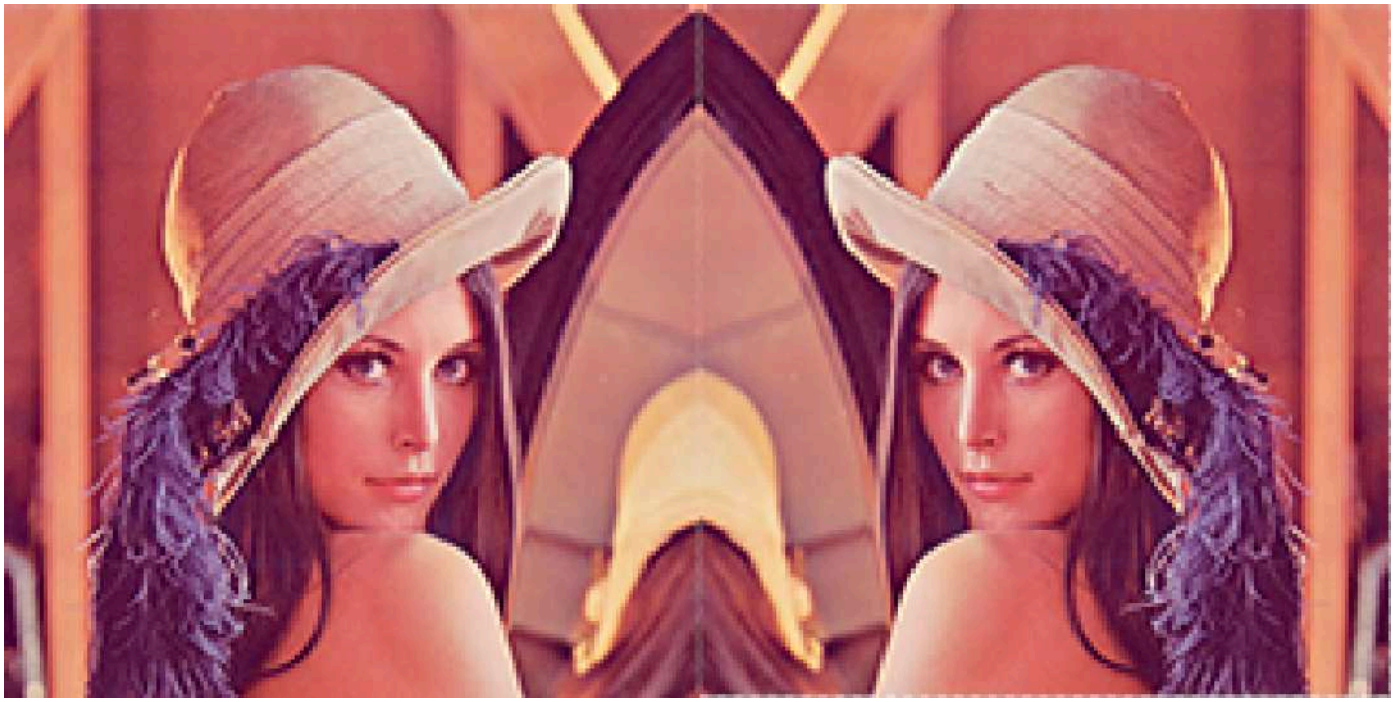
4.3 水平翻转

水平翻转无任何参数，仅将图片进行左右翻转

```
for (int y = 0; y < height_size; y++) {
    unsigned char* cur_row = image->pixel_data + y * actual_row_size;
    for (int x = 0,_x = row_size-3; x < row_size/2; x+=3,_x-=3) {
        int x1 = cur_row[x];
        int x2 = cur_row[x+1];
        int x3 = cur_row[x+2];
        cur_row[x] = cur_row[_x];
        cur_row[x+1] = cur_row[_x+1];
        cur_row[x+2] = cur_row[_x+2];
        cur_row[_x] = x1;
        cur_row[_x+1] = x2;
        cur_row[_x+2] = x3;
    }
}
```

实现较为简单，即以行为单位，交换左右对称的像素值。以下为水平翻转实现，左侧是原图

```
./build/bmpedit1y -i pic_in/LenaRGB.bmp -o pic_out/Lena_flip.bmp -op flip
```



4.4 旋转

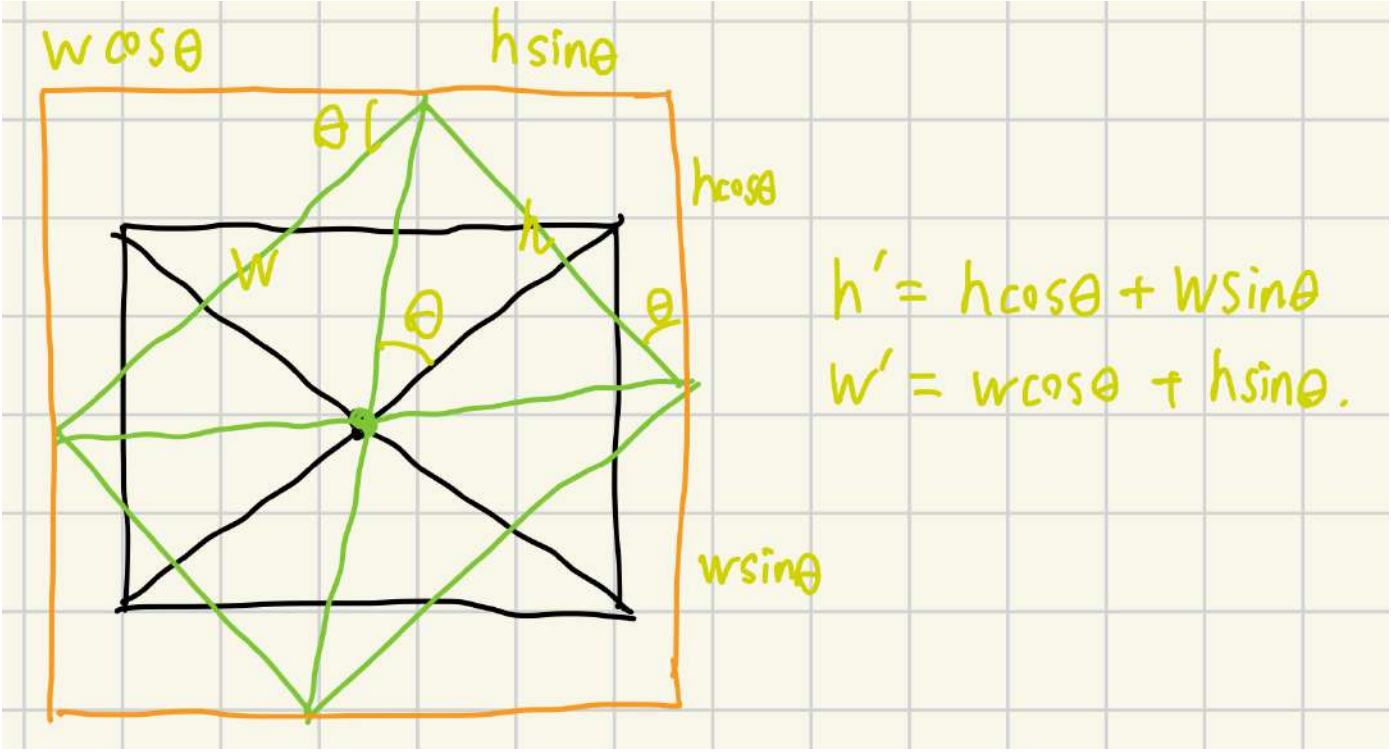
4.4.1 难点和思路

在考虑旋转实现的时候有一个问题，如果旋转后想要保留图片的完整内容(仅指内容，不包括清晰度)，在角度对90度取模不为0的旋转下，一定会出现角落区域是无图片信息的，如果是32位图片，我们可以通过alpha通道实现透明化，但是我们采用的24位BMP图像没有alpha通道，所以最终旋转呈现出的角落部分默认设置为全黑。

旋转的基本原理很简单，那就是将原图片的各个像素点转换成坐标，之后利用旋转矩阵得到旋转后新的坐标点，将旧像素的信息复制到映射后对应的位置即可。然而实际实现存在两点问题：

- 首先，旋转后的图幅是否要发生变化？如果不发生，那么整张图片的分辨率就要放大，如果保持分辨率不变，那么映射的计算就会变得复杂，不但要计算旋转，还要进行一定的缩放。为了保证速度，我们选择图幅不变，分辨率放大。
- 其次，首次转换成的坐标是相对于图片整体左下角的，而左下角却不是旋转中心，所以无法通过矩阵线性变换到对应位置。因此，我采用原点变换的方法，将旋转中心置于图片中心后，再计算新的相对坐标，旋转后再转化回原始坐标。

根据数学公式，当旋转角度为θ，原宽高为W和H时，可以得到新的高为：H' = Hcosθ + Wsinθ，新的宽为 W' = Wcosθ + Hsinθ



4.4.2 简单旋转实现

执行旋转的步骤如下：

1. 以新宽高为新的图片分辨率，定义两张图的中心点C1和C2相对于左下角的坐标分别为对应宽高的一半。

```
float center_x_old = old_width / 2.0f;
float center_y_old = old_height / 2.0f;
float center_x_new = new_width / 2.0f;
float center_y_new = new_height / 2.0f;
```

2. 进入循环后，首先根据当前的坐标减去中心点坐标，得到相对于旋转中心坐标的新坐标。根据逆旋转矩阵，算出旋转前对应的旧坐标

```
//in outer loop
unsigned char* new_row = new_image->pixel_data + y * new_actual_row_size;
unsigned char* old_row = image->pixel_data;

//in inner loop
float x_new = x - center_x_new;
float y_new = y - center_y_new;
// 在循环外预计算cos和sin的值，这里直接用cos_val和sin_val代替
float x_old = x_new * cos_val + y_new * cos_val + center_x_old;
float y_old = -x_new * sin_val + y_new * cos_val + center_y_old;
```

3. 由于实际坐标值是整数，所以我们通过加0.5进行四舍五入得到整数的旧坐标，并检查是否越界，如果对应的旧坐标在范围之外，那么新坐标的实际位置就是出在角落的黑色区域内，直接默认为全0，保持黑色

```
int x0 = (int)(x_old + 0.5f);
int y0 = (int)(y_old + 0.5f);
if (x0 < 0 || x0 >= old_width || y0 < 0 || y0 >= old_height) {
    continue;
}
```

4. 最后，由于我们计算的x是分辨率的位置，但实际上每个像素都有3个字节，所以我们要挨个对应赋值

```
int src_pos = y0 * old_actual_row_size + x0 * 3;
new_row[x * 3]     = old_row[src_pos];       // B
new_row[x * 3 + 1] = old_row[src_pos + 1];   // G
new_row[x * 3 + 2] = old_row[src_pos + 2];   // R
```

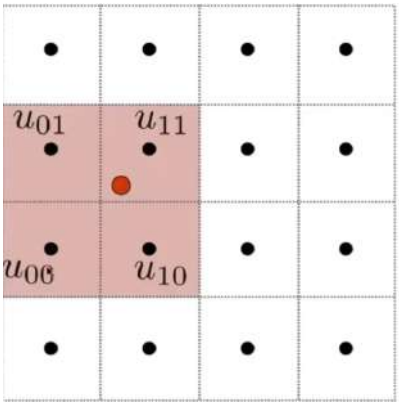
至此，图片旋转就完整实现了。

4.4.3 双线性插值进行清晰度优化

在旋转前后，程序实现的原图像在新图像上的分辨率没有发生改变，但是每个像素对应的位置确实发生了变化。因此，由于旋转矩阵算出的新坐标大部分情况下都不一定是整数，所以可想而知，实际上当我们做像素的旋转移动时，对应到的新位置并不是和新的像素位置严丝合缝的，然而在计算过程中我们就直接对小数部分进行截断，而这个过程一定会产生清晰度损失。因此，我们是否可以有一个优化清晰度的算法呢?

答案是存在。原本我们是截断了新坐标的小数部分取得了x0和y0作为最临近点，但这样显然表明我们会丢失一部分信息，也就是实际算出的点和实际采用的点之间存在一个一定的距离。如果我们不忽视这段距离，而是尝试利用起来，就可以一定程度上提高清晰度。

如图所示，如果旋转变换后的点为红点，那么我们原先截断小数后可能直接就采用了u01点，然而实际上这个点可能离u11点更接近。因此我们使用被称作**双线性插值**的方法来混合周边最近的点信息，最后得到新的点信息。



Take 4 nearest sample locations, with texture values as labeled.

从原步骤的第三步开始，原本只计算了x0和y0，现在计算对角线上的x1和y1，并组合成四个临近点。此外，也要进行边界条件判断

```
int x0 = (int)x_old;
int y0 = (int)y_old;
int x1 = x0 + 1;
int y1 = y0 + 1;

if (x0 < 0 || x1 >= old_width || y0 < 0 || y1 >= old_height) {
    continue;// stay black for the corner
}
```

随后，计算先前提到的实际点和采用点间的水平竖直距离`dx`和`dy`，由于采用点是通过截断小数部分得到的，所以可以保证`dx`和`dy`均小于1。由这一段距离可以算出新点混合周边四个点要采用的对应权重`w00`,`w01`,`w10`,`w11`。

```
float dx = x_old - x0;
float dy = y_old - y0;
float w00 = (1 - dx) * (1 - dy);
float w01 = (1 - dx) * dy;
float w10 = dx * (1 - dy);
float w11 = dx * dy;
```

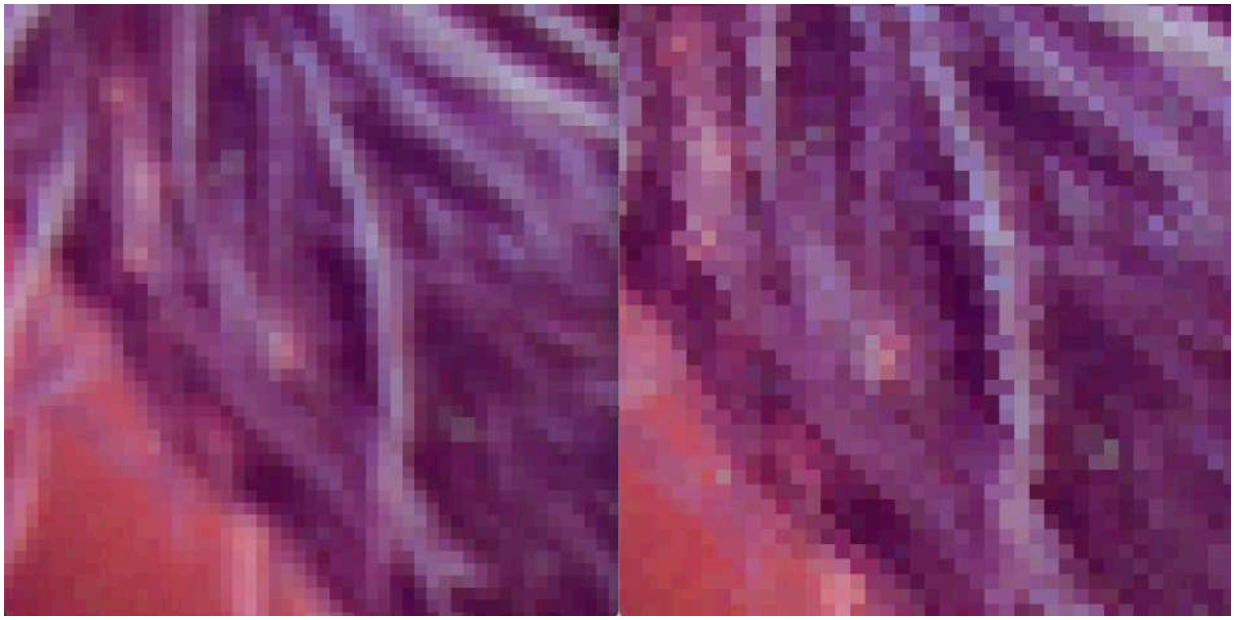
最后，算出四个点对应的位置坐标`p00`,`p01`,`p10`,`p11`，并计算加权后的RGB值，赋值给新的点

```
int p00 = y0 * old_actual_row_size + x0 * 3;
int p01 = y1 * old_actual_row_size + x0 * 3;
int p10 = y0 * old_actual_row_size + x1 * 3;
int p11 = y1 * old_actual_row_size + x1 * 3;

float val;
val = w00 * old_row[p00] + w01 * old_row[p01] + w10 * old_row[p10] + w11 * old_row[p11];
new_row[x * 3] = (unsigned char)fmin(255, fmax(0, val));

val = w00 * old_row[p00+1] + w01 * old_row[p01+1] + w10 * old_row[p10+1] + w11 *
old_row[p11+1];
new_row[x * 3+1] = (unsigned char)fmin(255, fmax(0, val));

val = w00 * old_row[p00+2] + w01 * old_row[p01+2] + w10 * old_row[p10+2] + w11 *
old_row[p11+2];
new_row[x * 3+2] = (unsigned char)fmin(255, fmax(0, val));
```

由此，我们可以对比一下两种算法生成图片的清晰度，将Lena图旋转45度，得到新图，左侧是双线性插值算法，右侧是简单算法

```
./build/bmpedit1y -i pic_in/LenaRGB.bmp -o pic_out/LenaRotateComplex.bmp -op rotate 45
```

大图看起来没有什么区别，放大发丝部分观察：



可以看到双线性插值(左)得到的图像少了很多锯齿，整体边缘轮廓过度更为柔和，效果提升明显。

4.5 缩放

图片的缩放指的是将原分辨率的长和宽进行放大或者缩小，形成新的图片。这个过程中，涉及到放大的部分，就要考虑原图片的各个点位要如何计算放大后到赋值到新图片的各个点位，以及新图片相比于原图片多出的像素点要怎么安排具体的值；涉及到缩小的部分，就要考虑新图片的各个点位要如何计算放大后到对应的原图片的各个点位，以及原图片相比于新图片多出的的像素点要怎么利用好这一部分数据进行清晰度优化。

4.5.1 纯放大

4.5.1.1 简单纯放大实现

纯放大的基础实现原理较为简单，为了保证每个像素都有对应的值，我们需要遍历放大后的图片的每一个像素，并根据缩放的比例计算这个新图中的像素点缩小后对应的原图片上的哪个点，随后复制原图对应的像素信息到新图的点

```
for (int y = 0; y < new_height_size; y++) {
    int _y = (y * height_size) / new_height_size;
    unsigned char* cur_ref_row = image->pixel_data + _y * actual_row_size;//old row
position
    unsigned char* cur_new_row = new_image->pixel_data + y * new_actual_row_size;//new row
position
    for (int x = 0; x < new_width; x++) {
        int _x = (x * old_width) / new_width;

        cur_new_row[x * 3] = cur_ref_row[_x * 3];
        cur_new_row[x * 3 + 1] = cur_ref_row[_x * 3 + 1];
        cur_new_row[x * 3 + 2] = cur_ref_row[_x * 3 + 2];
    }
}
```

然而，和在旋转部分遭遇的问题一样，在对新图片的点进行缩小时，实际产生的精确位置是浮点数，但是像素点的坐标都是整数，所以缩小后实际采用的点是丢失了一定信息的。因此我们需要用到双线性插值来提高放大的清晰度

4.5.1.2 双线性插值优化纯放大

和旋转部分的双线性插值相比，纯放大的双线性插值使用意义并不完全相同。因为纯放大的线性插值本质上在做的事情是在已知像素点之间尽量平滑地插入新值；而旋转由于没有图幅的变换，所以本质上虽然也是从新图的每个点进行反向映射，但是主要是为了尽量平滑过渡，因为像素可以理解为一个个小正方形，所以水平和垂直的边天然就比斜边更加平滑，双线性插值是为了减少旋转过程中极易出现的锯齿。

虽然理论意义有区别，不过实际上代码实现的原理和旋转部分的双线性插值区别不大，将新图片的点对应回原图采用临近四个点计算权重，最后采用四个点加权后的值进行新点的真正值，具体实现如下：

```
for (int y = 0; y < new_height_size; y++) {
    float _y = y * ratio_y;
    int y1 = (int)_y; // y1 promised > 0 and < edge
    int y2 = y1+1;    // so y2 > 0 but not promised < edge
    if(y2 >= height_size) y2 = height_size-1;
    unsigned char* cur_y1_row = image->pixel_data + y1 * actual_row_size;//old row1
position
    unsigned char* cur_y2_row = image->pixel_data + y2 * actual_row_size;//old row2
position
    unsigned char* cur_new_row = new_image->pixel_data + y * new_actual_row_size;//new row
position
    for (int x = 0; x < new_width; x++) {
        float _x = x * ratio_x;
        int x1 = (int)_x;
        int x2 = x1+1;
        if(x2 >= old_width) x2 = old_width-1;

        //(x1,y1) (x2,y1) (x1,y2) (x2,y2)
        float offset_x = _x - x1;
        float offset_y = _y - y1;
        int _x1 = x1 * 3;
        int _x2 = x2 * 3;
```

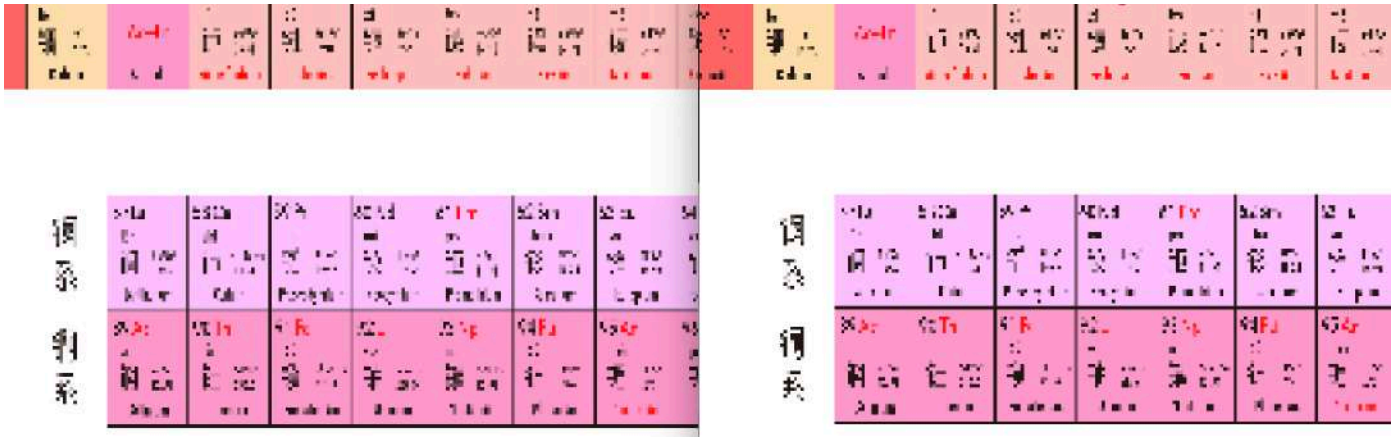
```
        //水平上方两个点插值
        int B1 = (float)cur_y1_row[_x1] + offset_x * (cur_y1_row[_x2] - cur_y1_row[_x1]);
        int G1 = (float)cur_y1_row[_x1+1] + offset_x * (cur_y1_row[_x2+1] -
cur_y1_row[_x1+1]);
        int R1 = (float)cur_y1_row[_x1+2] + offset_x * (cur_y1_row[_x2+2] -
cur_y1_row[_x1+2]);
        //水平下方两个点插值
        int B2 = (float)cur_y2_row[_x1] + offset_x * (cur_y2_row[_x2] - cur_y2_row[_x1]);
        int G2 = (float)cur_y2_row[_x1+1] + offset_x * (cur_y2_row[_x2+1] -
cur_y2_row[_x1+1]);
        int R2 = (float)cur_y2_row[_x1+2] + offset_x * (cur_y2_row[_x2+2] -
cur_y2_row[_x1+2]);
        //水平点在竖直方向上插值
        int B = B1 + offset_y * (B2 - B1);
        int R = R1 + offset_y * (R2 - R1);
        int G = G1 + offset_y * (G2 - G1);
        //溢出检测
        cur_new_row[x * 3] = (unsigned char)(B < 0 ? 0 : (B > 255 ? 255 : B));
        cur_new_row[x * 3 + 1] = (unsigned char)(G < 0 ? 0 : (G > 255 ? 255 : G));
        cur_new_row[x * 3 + 2] = (unsigned char)(R < 0 ? 0 : (R > 255 ? 255 : R));

    }
}
```

对比两种算法，得到结果如图，左边是未优化的纯放大，右边是用了双线性插值的纯放大。很明显右边的清晰度更高，过渡更平滑。

4.5.2 纯缩小

4.5.3.1 为什么不能用线性加权

纯缩小的算法在简单实现上和纯放大完全一样，完全可以调用同一个函数。此外乍一看似乎也可以采用双线性插值来优化算法，也就是把整个过程反过来：纯放大是遍历放大的图的每一个新点，并通过乘以放缩比例来找到原图位置的周边四个点进行线性加权，那么纯缩小也可以遍历缩小的图的每一个新点，乘以放缩比例找到较大原图的位置，对周围四个点进行线性加权。一开始我正是这样实现的：将4961x3508大小的元素周期表缩小到500x500。然而可以看到结果如下，左图是双线性插值，右图是纯缩小：

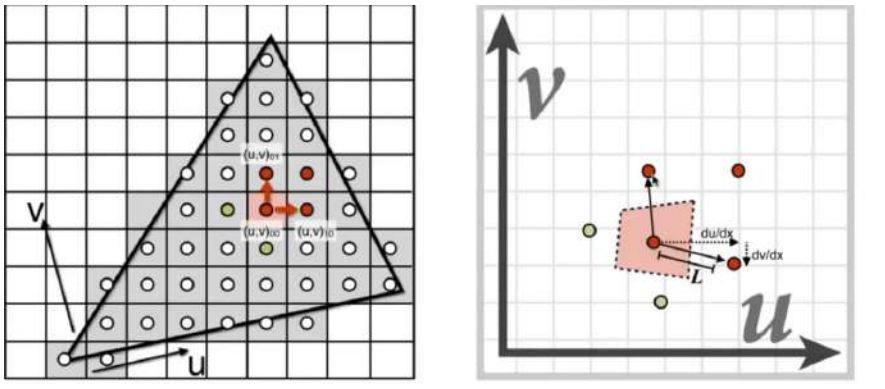


发现双线性插值产生的图片虽然和纯缩小不同，但是清晰度没有什么改观。

仔细分析原因，会发现放大和缩小并不是完全对称的。放大是将新坐标缩小回原图，对应原图的一片区域；而缩小是将新坐标放大到新图，对应新图的一片区域。

这其中的区别在于，**放大之所以采用双线性插值来平滑图像，是因为在新点缩小放回原图的过程中，新图的像素大小逐渐小于原图的像素大小，而原图的像素内部又可以视为完全一致，所以再怎么缩小，我们都没有办法找到更多的采样点来优化新图像素点的显示表现，所以不得已只能用双线性插值找临近四个点来加权平均，因为这从一开始就已经是最小的采样点了。**

但缩小不一样，**缩小后的图对应的像素点在放回原图的过程中像素大小逐渐大于原图的像素大小，而且缩小程度越大，放回原图后对应的覆盖区域就越大，能够使用的采样点就越多**，极端情况下，如果我们将1000x1000的图片缩小到1x1，那么这张1x1的图片唯一一个像素对应的就是原图的全部内容。而这时候，如果我们还用双线性插值，那么这唯一一个像素就只会采用原图的4个像素做平均，而不是全部。这就极大的浪费了原本可以利用的大量采样点。



$$D = \log_2 L \qquad L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2}\right)$$

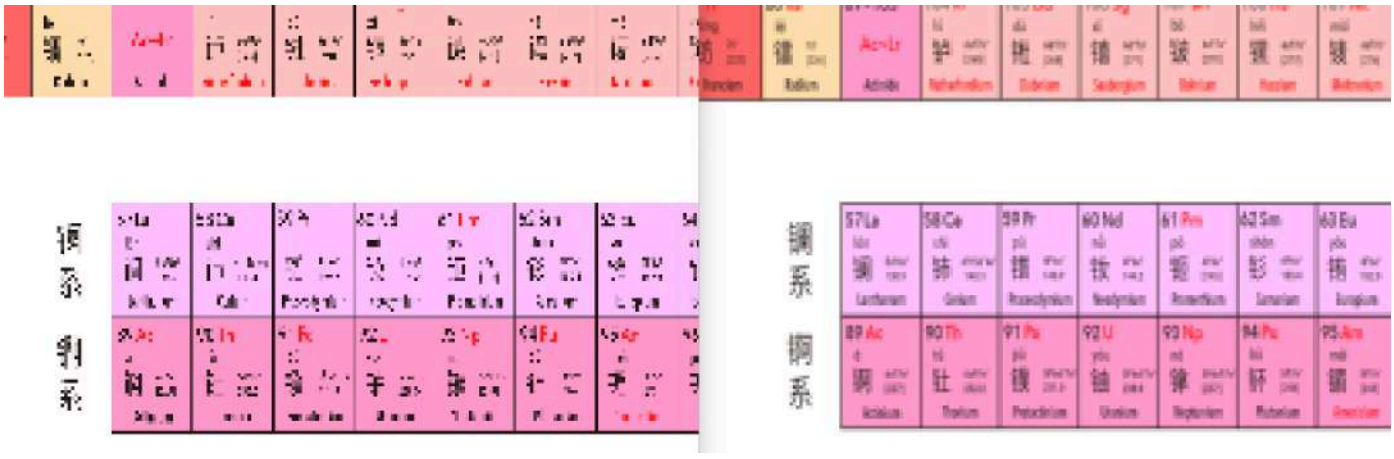
以上是图形学相关的一张图，用于解释纹理太大时如何用Mipmap优化映射效果，虽然公式与此无关，但是仅看两张图片的内容，可以发现和这里的缩小原理是一样的：新图的1单位像素要对应到原图(纹理)上，会发生放大，覆盖更多的区域。

4.5.3.2 区域平均采样

分析明白了这其中的区别，我们就可以实现基于区域采样的纯缩小图片优化，也就是将新像素点对应原图的区域全部进行平均后再赋值，这个过程本质上相当于一个**低通滤波器**，通过区域平均化来消除高频信息，这部分会在锐化部分再展开说明。

```
for (int y = 0; y < new_height; y++) {
    unsigned char* new_row = new_image->pixel_data + y * new_actual_row_size;
    for (int x = 0; x < new_width; x++) {

        int x1 = (int)(x * scale_x);
        int x2 = (int)((x + 1) * scale_x);
        int y1 = (int)(y * scale_y);
        int y2 = (int)((y + 1) * scale_y);
        x2 = (x2 >= old_width) ? old_width - 1 : x2;
        y2 = (y2 >= old_height) ? old_height - 1 : y2;

        int sum_b = 0, sum_g = 0, sum_r = 0;
        int count = 0;
        for (int _y = y1; _y <= y2; _y++) {
            unsigned char* old_row = image->pixel_data + _y * old_actual_row_size;
            for (int _x = x1; _x <= x2; _x++) {
                sum_b += old_row[_x * 3];
                sum_g += old_row[_x * 3 + 1];
                sum_r += old_row[_x * 3 + 2];
                count++;
            }
        }
        new_row[x * 3]     = (unsigned char)(sum_b / count);
        new_row[x * 3 + 1] = (unsigned char)(sum_g / count);
        new_row[x * 3 + 2] = (unsigned char)(sum_r / count);
    }
}
```

这其中定义了x1,x2,y1,y2，分别对应映射回原图后整片区域的四个角的坐标，在区域内分别计算RGB总和后再计算平均值。最终效果：



左侧是双线性插值，右侧是区域采样，很明显区域采样清晰度要提高很多，连字都能大致看清了。

4.5.3 任意比例放缩

4.5.3.1 实现策略：任务二分

纯放大和纯缩小我们可以根据其特性，分别采用线性插值和区域采样来优化清晰度，那么还有本质上是一种的变化：宽变短、长变更长，和宽变长，长变短。针对这种变化，我们可以通过已经实现的纯放大和纯缩小总共执行两次来实现，以宽变短，长变长为例：

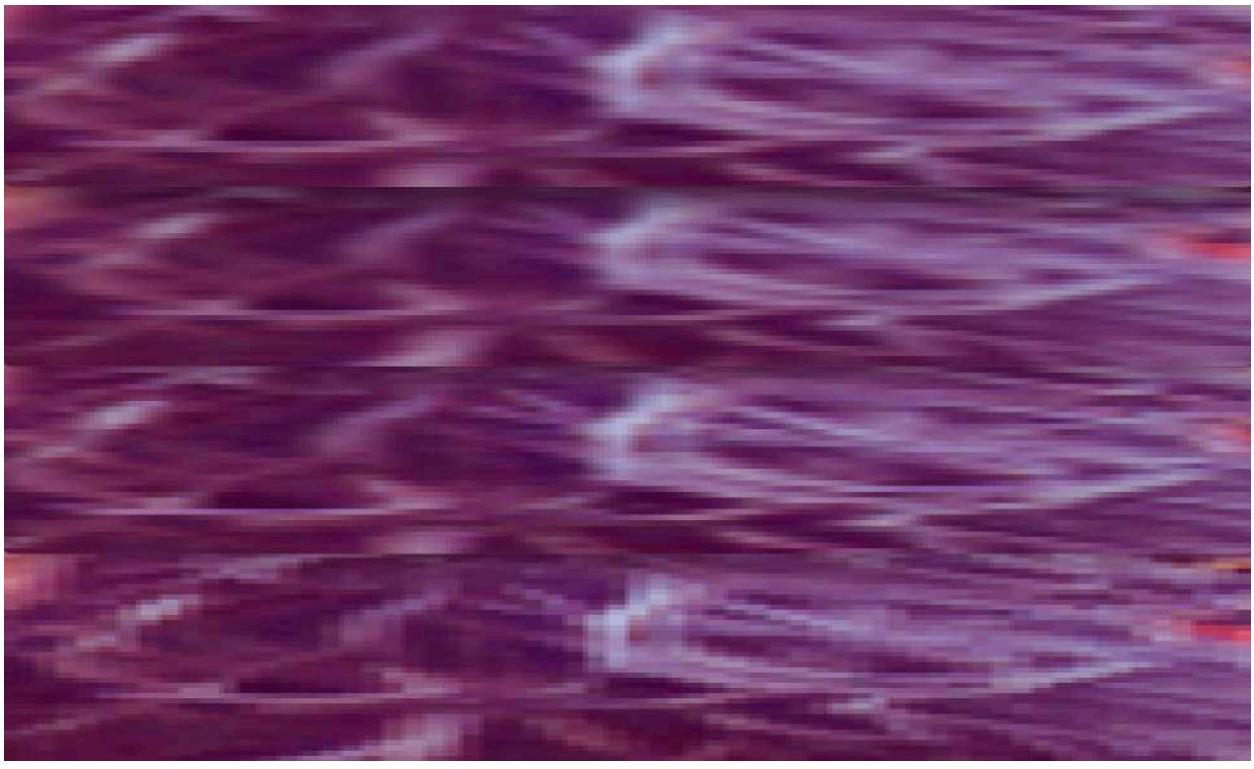
- 先纯放大，将长部分变长，宽不变，之后再纯缩小，将宽缩小，长不变；
- 先纯缩小，将宽部分变短，长不变，之后再纯放大，将长放大，宽不变。

4.5.3.2 为什么要先缩小

现在的问题在于，先缩小后放大，先放大后缩小，这两种哪种更优？根据先前已经观察过的许多图像，我们可以看到，图片清晰度的优化主要集中在如何更密集地采样并且平滑处理边缘，而且不论是图片放大还是缩小，遍历的像素都是新图幅的像素。因此，最后的理论结论是先缩小后放大，主要原因如下：

1. 从清晰度角度：先缩小相当于给图片添加了一个低通滤波器，最终得到的结果是每个新像素为原部分区域的像素平均值，减少了高频信息的比例，使得过渡更加平滑，在后续的放大过程中由于高频信息在缩小时已经被去除并替换为了周围像素的平均，所以最后遍历放大图像的像素时，就降低了某个新像素加权平均后读取到原较小图某个颜色相对周围突变的像素的概率，使得放大后的图像过渡期也更加平滑。
2. 而如果先放大后缩小，一开始将图像放大就会通过插值得到大量作为原像素中间值的新像素，这一部分是生成得到的，本质上信息还是来源于原图像的像素信息，后续缩小过程将这一部分作为区域采样的一部分并不会增加实质上有用的信息。
3. 从时间角度：由于不论是图片放大还是缩小，遍历的像素都是新图幅的像素，所以先缩小，遍历的是新旧图的**交集**部分的像素，是最小的，后放大，遍历的也只不过是新图的全部像素；而如果先放大，遍历的是新旧图的**并集**部分的像素，是最大的，后缩小，遍历的还是新图的全部像素，这样先缩小后放大显然具有更低的时间开销。

由此，我们对比四种方式处理一张512x512的Lena图像，一个边放大一个边缩小到2000x200大小，将得到的图像移动到发丝位置，得到如下四张图：



这四张从上向下依次是：先放大后缩小，先缩小后放大，纯放大函数处理，纯缩小函数处理。可以发现先放大后缩小和先缩小后放大结果较为相似，纯放大处理清晰度下降，纯缩小处理清晰度更低。虽然前两者在这次处理中没有明显的清晰度的区分，不过我后续又测量了时间，对比两种算法处理 512x512 到 200x9000 大小，以下为时间结果：

函数	测量十次的平均时间(ms)
先放大后缩小	14.7
先缩小后放大	6.2

可见先缩小实验结果符合理论，速度比先放大快了一倍不止。

4.5.3.3 代码实现

由于细分起来rescale只有四种可能，所以直接用if else 判断并根据条件判断执行纯放大或纯缩小操作即可

```
if(old_width >= new_width && old_height >= new_height){ //纯缩小
    result = image_scale_smaller(image,new_width,new_height);
}else if(old_width < new_width && old_height < new_height){ //纯放大
    result = image_scale_larger(image,new_width,new_height);
}else if(old_width >= new_width && old_height < new_height){ //宽变大，长变短
    BMPImage* tempt = image_scale_smaller(image,new_width,old_height);
    result = image_scale_larger(tempt,new_width,new_height);
    free(tempt);
}else if(old_width < new_width && old_height >= new_height){ //宽变短 长变长
    BMPImage* tempt = image_scale_smaller(image,old_width,new_height);
    result = image_scale_larger(tempt,new_width,new_height);
    free(tempt);
}
```

4.6 混合

4.6.1 基本实现

有了先前的缩放函数，混合就变得容易很多了。如果按两张图片各自的原图分辨率进行融合，那么如果一张图片比另一张长宽都更小，那么会出现小图片位于大图片的左下角以一定比例混合，如果一张图片的长比另一张大，宽却更小，那么甚至会出现多余的黑色边缘。

因此，为了更规整的进行图片混合，我根据输入两张图片的长和宽取最大值作为混合图的长和宽，并依次将两张图放缩到相同分辨率。

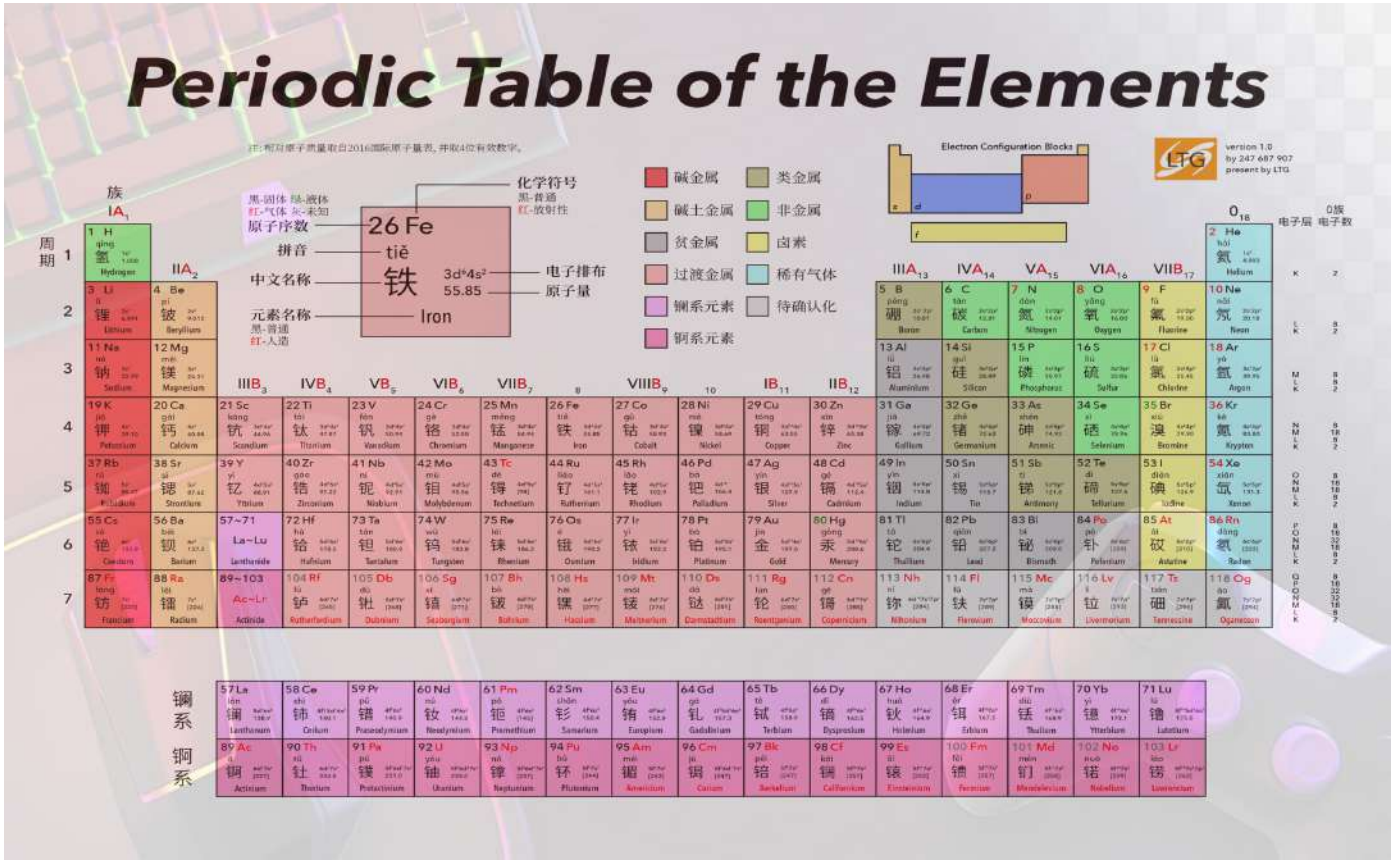
```
int w1 = image1->dib_header.width;
int w2 = image2->dib_header.width;
int h1 = image1->dib_header.height;
int h2 = image2->dib_header.height;
int max_w = w1 > w2 ? w1 : w2;
int max_h = h1 > h2 ? h1 : h2;

BMPImage* image1_r = image_rescale(image1,max_w,max_h);
BMPImage* image2_r = image_rescale(image2,max_w,max_h);
```

随后依照输入的参数按可调节的一定比例进行混合，

```
for (int y = 0; y < height; y++) {
    int offset = y * actual_row_size;
    for (int x = 0; x < width; x++) {
        int idx = offset+x*3;
        row1[idx] = row1[idx] + ratio * (row2[idx] - row1[idx]);
        row1[idx+1] = row1[idx+1] + ratio * (row2[idx+1] - row1[idx+1]);
        row1[idx+2] = row1[idx+2] + ratio * (row2[idx+2] - row1[idx+2]);
    }
}
```

示例：将两张图按2:8的比例混合，其中参数0.2是blend后路径的图片的占比

```
./build/bmpedit -i pic_in/R-C.bmp -o report/blendSTUGAME2.bmp -op blend
pic_in/backGround1.bmp 0.2
```



4.6.2 SIMD加速

由于这里的融合也出现了大面积连续内存访问的循环，所以我们可以试着用SIMD指令批量乘加，加快速度。不过由于我们使用的pixel_data是unsigned char类型，所以如果想要非均衡比例进行混合的话，就需要将unsigned char 全部转化成浮点再进行SIMD加速，原因就在于有一个浮点类型的比例参数。因此，在实现过程中，我们仅对0.5比例混合的图片进行特判，全程使用unsigned char进行计算

```
inline void mix_arrays(uint8_t *p1, const uint8_t *p2, float ratio, size_t num) {
    size_t i = 0;
    float inv_ratio = 1 - ratio;
#if defined(_NEON_)
    if(fabs(ratio-0.5)<FLT_EPSILON){
        for (; i + 15 < num; i += 16) {
            uint8x16_t v_p1 = vld1q_u8(p1 + i);
            uint8x16_t v_p2 = vld1q_u8(p2 + i);
            uint8x16_t v_result = vrhaddq_u8(v_p1, v_p2);

            vst1q_u8(p1 + i, v_result);
        }
    }
#endif
    for (; i < num; i++) {
        p1[i] = p1[i] * inv_ratio + p2[i] * ratio;
    }
}
```

最终结果在不启用多线程，不开O3优化，对两张9000x9000的图片进行平均混合，得到如下结果

操作	采用NEON	不采用NEON
图片混合	234.307 ms	349.183 ms

可见有明显速度提升

4.6.3 OpenMP并行是否更快?

可以看到，在混合实现之前，我们直接创建两张新的图片：

```
BMPImage* image1_r = image_rescale(image1,max_w,max_h);
BMPImage* image2_r = image_rescale(image2,max_w,max_h);
```

然而缩放图片仍会消耗时间，所以我们略微增加一些判断，以尽量减少创建新图片的概率：

```
if(w1==w2 && h1==h2){//两张图片大小一样，不用创建任何图片
    image1_r = image1;
    image2_r = image2;
}else{
    if(max_h == h1 && max_w == w1){//如果第一张图片比第二张大，只用放大第二张图片
        image1_r = image1;
        image2_r = image_rescale(image2,max_w,max_h);
```

```
    }
    else if(max_h == h2 && max_w == w2){//如果第二张图片比第一张大，只用放大第一张图片
        image2_r = image2;
        image1_r = image_rescale(image1,max_w,max_h);
    }else{//剩余情况
        #pragma omp parallel sections
        {
            #pragma omp section
            {
                image1_r = image_rescale(image1, max_w, max_h);
            }
            #pragma omp section
            {
                image2_r = image_rescale(image2, max_w, max_h);
            }
        }
    }
}
```

针对最后的剩余情况：两张图片都需要放缩，但是两张图片的放缩彼此不相干，所以采用多线程并行执行两个图片的放缩，然而，实际使用时却发现，这么做比起单纯的串行执行反而慢了两倍，仔细查看函数发现，rescale中调用了bigger和smaller两个函数，这两个函数内部又在最外层for循环定义了多线程，所以这里并行执行两个图片创建，可能会导致线程数超过核心数，导致性能下降，或者因为线程调度问题导致调度时间具备一定开销。因此，经过测试，对比发现，**外部创建多线程执行，实际速度不如外部串行执行，内部再并行执行**。最终将剩余情况又改回了串行执行

```
else{
    image1_r = image_rescale(image1, max_w, max_h);
    image2_r = image_rescale(image2, max_w, max_h);
}
```

4.7 高斯模糊

4.7.1 实现方案

先前在图像放缩部分提到过图片缩小相当于一个低通滤波的过程，因为图片缩小后像素的取值对应的是原图部分区域的平均值，这个过程中降低了高频信息的比重，反映在图片上也就是图片中物体的边缘部分变模糊。那么既然如此，我们自然可以考虑两个问题：

1. 既然对部分区域取平均值可以实现模糊，那么是否可以在不进行图片缩小的前提下只进行图片模糊呢?
2. 对部分区域取平均可以实现模糊，那么我们是否可以自定义区域采样的规则，从而实现其他图像处理操作呢?

答案自然是可以，如果不将图片缩小，我们只需要新定义一个分辨率大小一样的图片，每个新像素的取值对应原图片对应像素位置周边像素的平均值，最后就可以得到新的模糊的图片，而平均的区域越广，模糊效果越明显。基于以上考虑，我们可以通过一个固定值的矩阵(通常被称为卷积核)，使得矩阵内部的值均为1/矩阵大小进行归一化，之后如同一个窗口在原图像上挨个像素移动，每次移动都会平均窗口内像素的值，从而实现模糊。由此，第二个问题也得到了解决，只要我们自定义这个窗口矩阵内的值，就可以实现不同的效果。

4.7.2 初步实现

最常见的模糊卷积操作被称作高斯模糊，我们可以预定义高斯模糊的卷积核，并在遍历每个像素的过程中根据卷积核的大小来确定要进行计算的范围，随后进行矩阵乘法，得到的值作为新像素的结果。

```
for (int y = 0; y < height_size; y++) {
    int y1 = y - range;
    int y2 = y + range;
    for (int x = 0; x < old_width; x++) {
        float sum_B = 0,sum_G = 0,sum_R = 0;

        int x1 = x - range;
        int x2 = x + range;
        int p = 0;
        for(int i = y1 ; i <= y2 ; i++){
            for(int j = x1 ; j <= x2 ; j++){
                sum_B += cur_row[i * actual_row_size + j * 3] * G_kernel[p];
                sum_G += cur_row[i * actual_row_size + j * 3+1] * G_kernel[p];
                sum_R += cur_row[i * actual_row_size + j * 3+2] * G_kernel[p];
                p++;
            }
        }
        new_row[y * actual_row_size + x * 3] = sum_B;
        new_row[y * actual_row_size + x * 3+1] = sum_G;
        new_row[y * actual_row_size + x * 3+2] = sum_R;
    }
}
```

4.7.3 复杂度优化：卷积核拆分

然而，我们可以看到先前的初步实现定义了四重for循环，假设图片的长和宽为n，卷积核的大小为k^2，那么总的时间复杂度就是O(n^2*k^2)，随着卷积核和图像大小的增加，图像处理的速度会显著变慢，不过这其中包括了大量的重复运算，所以我们可以通过拆分二维卷积核为一维卷积核来达到更快的效果：当一个二维卷积核M可以被拆分为一个一维卷积核m，则M=m * m^T，随后，我们对图像先做一维卷积核的横向处理，再遍历一遍图像做竖向处理：

```
for (int x = 0; x < old_width; x++) { //这是竖向处理的部分，横向和这个极其类似
    for (int y = 0; y < old_height; y++) {
        float sum_B = 0, sum_G = 0, sum_R = 0;

        for (int k = -range; k <= range; k++) {
            int yk = y + k;
            if (yk < 0) yk = 0;
            if (yk >= old_height) yk = old_height - 1;

            int idx = yk * actual_row_size + x * 3;
            float w = kernel_1D[k + range];

            sum_B += new_row[idx] * w;
            sum_G += new_row[idx + 1] * w;
            sum_R += new_row[idx + 2] * w;
        }
```

```
            int _idx = y * actual_row_size + x * 3;
            new_row[_idx] = fminf(255.0f, fmaxf(0.0f, sum_B));
            new_row[_idx + 1] = fminf(255.0f, fmaxf(0.0f, sum_G));
            new_row[_idx + 2] = fminf(255.0f, fmaxf(0.0f, sum_R));
        }
    }
```

这样也可以达到完全相同的结果，而且总计算时间为2 * n^2 * k = O(n^2*k)，在k较大时理论上有很显著的效果。

此外，还必须注意实现多线程时的一些注意事项：按照常理，为了内存连续性的处理，我们应该令纵向遍历为外循环，横向遍历为内循环。然而在高斯模糊的竖向处理中，我们如果想要开启OpenMP进行多线程加速，那么假设我们以纵向循环为外循环，那么OpenMP分配的多线程就会将多个行分为一组进行处理，可是这里的处理是纵向的，在不同线程处理上下相邻的行时由于有效卷积核的大小大于1，会导致并行读入和写入。因此必须以横向遍历为外循环，这样线程之间才能保证互不影响，且速度有效提升：

```
#pragma omp parallel for schedule(guided)
for (int y = 0; y < old_height; y++) {//y 先
    for (int x = 0; x < old_width; x++) {
      ......
    }
}

#pragma omp parallel for schedule(guided)
for (int x = 0; x < old_width; x++) {// x 先
    for (int y = 0; y < old_height; y++) {
      ......
    }
}
```

以下为实现效果：右边为原图，中间为错误处理的图，左侧为正确处理的图，采用了41大小的高斯核做了1次迭代操作：

4.8 锐化

4.8.1 普通实现

锐化操作本质上和模糊没有区别，都是通过不同的核来对某个像素及其周围像素的值进行线性加权平均得到新的结果。只不过锐化使用的核更特殊，往往表现为在保证归一化的情况下，矩阵越靠近边缘，越可能出现负值，越靠近中间，越可能出现一个较大的正数。利用这个特点，经过简单的思考就可以发现，如果一个区域像素的颜色比较相近，那么混合后的颜色也应该是接近的，如果出现了物体边缘，那么这个卷积核就可以放大边缘和相邻像素的差异，实现对边缘像素的增强。

由于不同的输入最终会调用不同的卷积核，所以预先用静态一维数组进行统一管理：

```
static float LAPLACIAN_KERNEL[] = {
    -1, -1, -1,
    -1, 9 , -1,
    -1, -1, -1
};
......
static float* KERNALS[] = {//数组统一管理
    VOID_KERNEL,         //0 无操作，便于类型变量输入越界时不做任何操作
    LAPLACIAN_KERNEL,    //1 拉普拉斯算子，进行锐化操作
    SOBEL_KERNEL,        //2 浮雕效果
    MASK_KERNEL          //3 较弱的锐化效果
};
```

可以发现，为了保证计算的结果不超出范围，所有卷积核的内部值相加均为1。为了保证计算速度，预定义的核均为3x3的大小，然而，如果我们想要实现一定的锐化程度的变化，定义过多的卷积核又回占用大量空间，所以我们使用局内多次迭代的方法：

```
for(size_t i = 0 ; i < iter ; i++){...}
```

4.8.2 复杂度优化：边缘不处理

正如先前所提到的，和高斯模糊不一样，我们预定义的核均为3x3的大小。因此，对于大部分图像，实际上我们只需要对边缘的一圈单像素处理卷积核边缘越界的情况，中间的大面积范围的操作都不需要做判断操作。此外，实际上锐化处理时，边缘一圈的像素即便不做处理，也不会有很大的影响。因此，我们可以将原遍历操作改写为空出边缘一圈像素进行无边缘判断的处理，最后将原图像的边缘一圈复制到新图像的边缘:

```
int row_offset = (old_width - 1) * 3;
for (int y = 1; y < old_height-1; y++) {     //遍历时不经过第一行和最后一行
    //中间行的第一和最后一个像素直接复制
    int col_offset = y * actual_row_size;
    new_row[offset] = cur_row[offset];
    new_row[offset+1] = cur_row[offset+1];
    new_row[offset+2] = cur_row[offset+2];
    col_offset += row_offset;
    new_row[offset] = cur_row[offset];
    new_row[offset+1] = cur_row[offset+1];
    new_row[offset+2] = cur_row[offset+2];
    for (int x = 1; x < old_width-1; x++) {//遍历时不经过中间行的第一和最后一个像素
        ......
    }
}
```

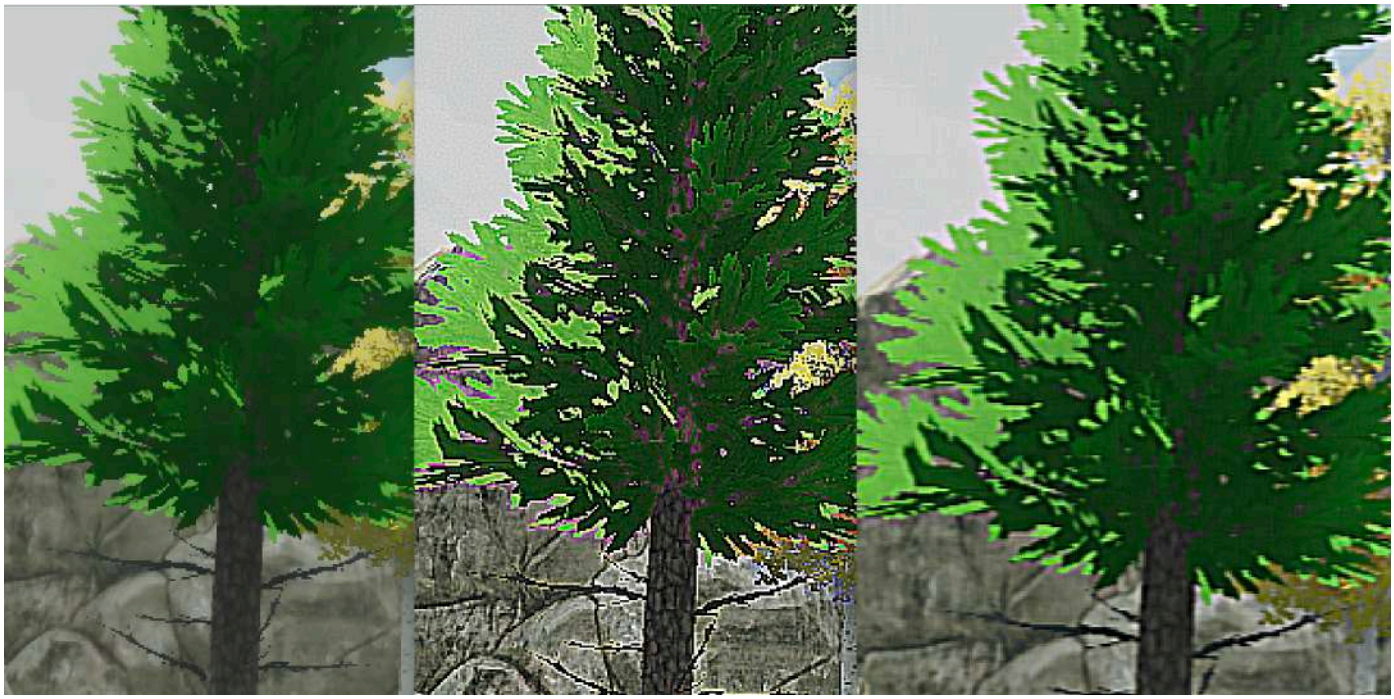
最后在迭代外部复制边缘信息：

```
memcpy(new_row, cur_row, actual_row_size);//复制第一行和最后一行
memcpy(new_row+(old_height-1)*actual_row_size, cur_row+(old_height-1)*actual_row_size,
actual_row_size);
```

如此，可以提高一定的处理速度，减少大量不必要的边缘判断。

4.8.3 效果优化：预模糊降噪

然而，仅利用卷积核进行图片处理很多时候会导致出现边缘区域出现异常色斑，这是因为在锐化强度较大时，如果原图像本身的像素间过渡就不平滑，那么这些不平滑的噪声点或者小区域也会被识别成边缘信息，进而也被放大，甚至出现类似饱和度的增强。所以为了提高锐化效果，我们可以先通过高斯模糊预处理提高平滑度，再进行锐化处理。下面我们选一个像素风3D场景的树作为例子：



以上左图为原图，中间为用拉普拉斯核纯锐化的结果，可以看到叶子边缘出现了紫色的异常色斑，右图为用较弱的高斯模糊预处理后再锐化的结果，相对来讲降低了一定的色彩异常。

```
./build/bmpedit -i pic_in/tree.bmp -o pic_out/blur_sharp.bmp -op blur 0.1 1 -op sharp 1 1
```

4.9 滤镜

卷积核矩阵除了可以对图片的多个像素进行操作以外，也可以通过预定义好的矩阵对单个像素的RGB进行变换操作实现一些常见的滤镜操作。

4.9.1 矩阵升维“实现”非线性变换

然而，在实现过程中，我发现反色操作，也就是将RGB颜色的值x全部转换为255-x，不属于线性变换，也就无法直接用3x3的矩阵实现反色的计算，然而如果要额外为非线性变化的滤镜做特殊判断，不利于统一管理，代码格式也不够美观。因此我们可以将3x3的矩阵扩大为4x4，并将RGB增加一位固定为1的值，这样，将矩阵设置成以下内容，就可以实现反色操作了，也可以扩展实现其他的简单非线性变换。

```
static float INVERT_MATRIX[] = {
    -1,     0,      0,     255,
    0,     -1,      0,     255,
    0,      0,     -1,     255,
    0,      0,      0,       0
};
```

和锐化/模糊部分类似，也通过type预先确定矩阵内容。由于预定义的核均为4x4，所以展开循环进行运算，此外这里没有多像素通道的混合，所以计算量相对较低。

```
//根据type在循环外确定matrix内容
float* matrix = FILTER_MATRICES[type];
//循环内固定大小的矩阵乘法
int R = 0,G = 0,B = 0,M = 1;
```

```
B = cur_row[offset+x];
G = cur_row[offset+x+1];
R = cur_row[offset+x+2];

float new_R = matrix[0] * R + matrix[1] * G + matrix[2] * B + matrix[3] * M;
float new_G = matrix[4] * R + matrix[5] * G + matrix[6] * B + matrix[7] * M;
float new_B = matrix[8] * R + matrix[9] * G + matrix[10] * B + matrix[11] * M;

cur_row[offset+x]   = (int)fmax(0, fmin(255, new_B));
cur_row[offset+x+1] = (int)fmax(0, fmin(255, new_G));
cur_row[offset+x+2] = (int)fmax(0, fmin(255, new_R));
```

目前已经实现的滤镜包括灰度、复古和反色。

4.10 二值化

4.10.1 二值化类型

提到灰度滤镜，在图像处理方面还有一个名为二值化的操作。一般通过图像的灰度图和一个特定阈值来将整个图像呈现出明显的黑白视觉效果。常用于提取图像中的目标物体，将背景以及噪声区分开来，可能作为图像预处理过程中的图像分割环节，为后续的机器学习和物体检测提供前期步骤。其中，二值化也分成多个类型，以满足不同图像处理的需要。常见的算法类型有如下五种：

算法名称	算法公式
threshold(Gray,127,255,cv2.THRESH_BINARY)	像素点的灰度值大于阈值设其灰度值为最大值，小于阈值的像素点灰度值设定为0。
threshold(Gray,127,255,cv2.THRESH_BINARY_INV)	大于阈值的像素点的灰度值设定为0，而小于该阈值的设定为255。
threshold(Gray,127,255,cv2.THRESH_TRUNC)	像素点的灰度值小于阈值不改变，反之将像素点的灰度值设定为该阈值。
threshold(Gray,127,255,cv2.THRESH_TOZERO)	像素点的灰度值小于该阈值的不进行任何改变，而大于该阈值的部分，其灰度值全部变为0。
threshold(Gray,127,255,cv2.THRESH_TOZERO_INV)	像素点的灰度值大于该阈值的不进行任何改变，小于该阈值其灰度值全部设定为0。

以最基本的binary为例，C语言实现如下：

```
void binarize_norm(unsigned char* pixel, int * threshold) {
    int gray = 0.299f * pixel[2] + 0.587f * pixel[1] + 0.114f * pixel[0];
    unsigned char value = (gray > *threshold) ? 255 : 0;
    pixel[0] = pixel[1] = pixel[2] = value;
}
```

先得到灰度值，之后简单根据threshold做出判断，接着赋值即可。

4.10.2 函数指针统一管理

然而，二值化操作是不能通过矩阵变化实现的非线性变换，因此使用先前锐化/模糊的静态数组管理核再根据类型确定索引位置是不可行的。不过可以发现，不同类型的二值化操作输入的参数是一致的，所以我们可以利用函数指针来统一管理：

```
void binarize_norm(unsigned char* pixel, int threshold) {......}
void binarize_inv(unsigned char* pixel, int threshold) {......}
...//other functions
typedef void (*BinarizeFunc)(unsigned char*, int);
BinarizeFunc get_binarize_func(int type) {
    switch(type) {
        case 0: return binarize_norm;
        case 1: return binarize_inv;
        case 2: return binarize_trunc;
        case 3: return binarize_tozero;
        case 4: return binarize_tozero_inv;
        default: return binarize_norm;
    }
}
```

这样，在图像处理函数内部，就只需要调用返回函数指针的函数即可：

```
void image_Binarization(BMPImage* image,int threshold,int type) {
    if(!image) return ;

    int row_size = image->dib_header.width * 3;
    int padding = (4 - (row_size % 4)) % 4;
    int actual_row_size = row_size + padding;
    int height_size = image->dib_header.height;

    BinarizeFunc bin_ptr = get_binarize_func(type);

    #pragma omp parallel for schedule(guided)
    for (int y = 0; y < height_size; y++) {
        unsigned char* cur_row = image->pixel_data + y * actual_row_size;
        for (int x = 0; x < row_size; x+=3) {
            bin_ptr(&cur_row[x], threshold);
        }
    }
}
```

最后生成结果如下：

从左到右，上三张分别是：灰度图、反二进制阈值化、反阈值化为0；下三张分别是：二进制阈值化、截断阈值化、阈值化为0

4.11 操作平均运行时间和优化效果

在对大部分外循环启用OpenMP，开启O3编译优化，允许NEON使用的情况下，对一张4k(3840×2160)大小，占用空间24.9MB的图片进行全部可执行的操作，测得50次的平均计算时间，包括各个实现优化前后的数据

操作描述	时间	时间优化	时间
亮度增加(复杂) 30	22.75 ms	HSV分离操作+定值加法+SIMD	0.87 ms
色温增加 60	22.49 ms	HSV分离操作	17.84 ms
饱和度增加 30	22.97 ms	HSV分离操作	20.46 ms
明度增加 30	22.66 ms	HSV分离操作	20.42 ms
拉普拉斯锐化迭代1次	46.69 ms	边缘不处理	33.68 ms
高斯模糊5x5核迭代1次	79.20 ms	2D转双1D处理	65.58 ms
复古滤镜	8.00 ms	暂无	7.57 ms
旋转90度	63.92 ms	正余弦预处理	31.80 ms
缩放至1920x4320	29.82 ms	暂无	29.83 ms
和另一张较小的图片融合	32.62 ms	多线程、NEON	11.96 ms
切割左下角1000x1000的区域	0.68 ms	暂无	0.56 ms
水平翻转	1.75 ms	暂无	1.54 ms
二值化	3.96 ms	暂无	3.91 ms

可以看到，优化过的函数具有以下表现：

1. 定值加法+SIMD的使用使得亮度在非较大幅度的增加时运算速度极大幅加快，甚至可以和直接memcpy的切割操作相提并论
2. HSV的分离操作避免了冗余的部分参数传入和运算，有常数级的微幅提升
3. 锐化部分的边缘不处理+边缘复制通过减少循环内的大量多余分支提高了常数级的计算，实践效果却十分明显，提升了38%的速度
4. 高斯模糊2D转1D有了算法理论上的复杂度优化，不过有20%左右的速度提升，对于更大的卷积核应该效果更明显
5. 旋转部分cos和sin的预处理降低内部循环的计算，提升2倍速度
6. 融合部分，由于例子中的融合是图片较小的情况，所以优化后的判断使得整体减少了一张图片的生成时间，并且SIMD加速也有一定效果，最后速度提升了3倍左右

5 AI使用

5.1 代码框架初步实现

由于一开始虽然了解了BMP文件的大致格式和存储方法，但是仍没有较明确的写代码思路，所以让deepseek根据项目文档要求生成一份可以读写的框架。随后得到如下部分。可以看到Deepseek生成了三个结构清晰的结构体，同时对两个文件头采用压缩内存，和BMP文件字节数保持一致，这一点如果不了解，很容易忽略并且半天找不到问题所在。同时也一并生成了学起来不难，但是写起来又略微繁琐，很容易出问题的读写函数。可以说，在这个意义上AI工具可以很快地帮助我们节省一些不必要浪费的时间，对于一些冷门的知识也可以快速获取，降低了学习成

本，也提高了开发效率。

```
#pragma pack(push, 1) // Ensure no padding in structs
typedef struct {
......
} BMPHeader;

typedef struct {
......
} DIBHeader;
#pragma pack(pop)

typedef struct {
    BMPHeader file_header;
    DIBHeader dib_header;
    unsigned char* pixel_data;
} BMPImage;
```

5.2 帮助文档和目录生成

在写完代码之后，我们还需要一份help文档帮助用户了解如何使用，但这一部分写起来也比较繁琐，涉及到打表、对齐和易懂又不繁琐的表述。因此让deepseek直接根据已经完成的代码部分总结help文档的内容，得到的文档部分如下，可以看到既做了合适的排版，也清晰简洁地做了说明。

```
void print_help() {
    printf("BMP图像处理工具\n");
    printf("支持24位无压缩BMP文件格式\n\n");
    printf("用法: ./bmpedit1 -i 输入文件.bmp -o 输出文件.bmp -op <操作> [参数]\n");
    printf("选项:\n");
    printf("  -i <文件名>                  输入BMP文件路径 (必填)\n");
    printf("  -o <文件名>                  输出BMP文件路径 (必填)\n");
    printf("  -op <操作> <参数1> <参数2>  要执行的操作及参数\n");
    printf("                                 可通过多次输入-op进行多种操作,并按输入顺序执行\n\n");
    printf("可用操作:\n");
    printf("  addL <值>                    调整亮度 (范围: -100到100)\n");
    ......
}
```

此外，report接近完成时，可以根据内容生成目录，但是人力完成需要上下翻找，保证标题和目录中的一致，因此采用AI生成可以很快解决这个问题，所以我将文档丢给AI，以及一份上次report的目录格式，参照进行新的目录生成，得到如下结果，和预期一致。

```
# CS219 Project 3 : BMP Image Processing
## 目录
### 1 项目介绍
- 1.1 基本信息
- 1.2 使用说明

### 2 BMP读入
- 2.1 BMP文件格式
```

```
- 2.2 BMP文件的C语言读写实现
- 2.3 32位BMP文件的兼容读入

### 3 用户输入管理
- 3.1 输入解析和操作结构体
- 3.2 内存管理和鲁棒性
......
```

5.3 函数顶层注释

由于写代码的过程中，频繁进行改动，后续优化的时候也多次进行修改，所以最终导致代码没有有效注释，为了提升可读性，将原代码了给AI，生成了对应的注释。

6 总结

本项目实现了一个基于C语言的BMP图像处理程序，通过去透明通道、字节扩展等操作支持8/24/32位BMP图像的读入，进行多种常见图片处理操作以及支持24位BMP的写出。程序通过直接解析BMP文件头与像素数据，避免了外部库依赖。用户输入采用命令行参数解析，支持多操作队列顺序执行，并具备完善的错误检查与内存管理。核心功能包括基于HSV模型的色温、饱和度与明度调整，采用Photoshop的alpha遮罩算法优化亮度变化效果，以及通过双线性插值与区域采样实现高质量的图像缩放。此外，还实现了裁剪、旋转、混合、锐化、模糊、滤镜和二值化等操作。

在实现过程中，通过多种手段降低了时间开销(包括常数级和复杂度)，也运用多种算法优化了清晰度。对于旋转与缩放功能，采用双线性插值和区域采样算法提升了清晰度。混合操作通过多次条件判断降低更多处理的可能，并在均等混合情况使用SIMD指令实现加速。卷积核与矩阵变换通过拆分二维卷积核为一维卷积核，将高斯模糊的时间复杂度从$O(n^2k^2)$优化至$O(n^2k)$，并利用函数指针和静态矩阵数组统一管理锐化、模糊和滤镜的实现，减少了冗余计算。此外，SIMD指令集和OpenMP多线程的引入进一步加速了亮度调节和图片混合等操作，使得多数操作在4k大小的图像上平均耗时较低。

当然也有较多不足之处和难以解决的问题：

- 由于处理的是24位BMP文件，所以RGB通道的每一个值正好是8位无符号整数，导致大部分运算始于 unsigned char、终于 unsigned char，然而很多时候计算时却常常需要用到浮点数，想要使用NEON指令集，就必须要先将unsigned char 转化为浮点数，再批量处理，最终效果反而不如直接计算来的快。因此SIMD加速仅针对特定比例混合（比如均等混合，只需要相加之后按位右移一位即可）以及内存连续时候的部分函数，未全面覆盖其他操作。
- 很多时间优化有的时候难以下手，其原因在于：
    1. 很多操作没有有效的算法优化
    2. 竖直方向的遍历内存极其不连续，即便复杂度优化了，可能反而因为内存问题导致实际速度变慢
    3. 虽然是图片处理，但是大部分处理其实没有涉及矩阵/向量乘法等在连续的空间上进行简单加减乘除的操作，导致SIMD加速的实际提升有限，甚至不如不做提升
    4. 为了极致性能，可能OpenMP，O3优化，SIMD指令都会用到，他们之间有些情况也会出现冲突，在特殊情况下速度反而降低
- 此外，OpenMP的使用也可能造成和实际时间优化和清晰度算法的计算问题，其中时间部分，最令人印象深刻的在于，两个图像的并行处理本来可以采用OpenMP，但是如果图像处理的内部也调用了OpenMP，那么实际上的速度反而因为线程调度问题变得更慢；清晰度部分，对于锐化/模糊等涉及卷积核操作的函数，由于有效的核大小超过1，所以并行执行可能会导致写入/读入冲突，出现横向条纹。

- 此外，未实现更复杂的二值化、未利用傅立叶变换和逆变换得到频域信息并在数学层面优化图片处理效果、未实现涉及噪声、高低频信息等更复杂的图片处理效果。

7 参考

https://zh.wikipedia.org/wiki/BMP BMP文件格式

https://www.freeconvert.com/zh/bmp-converter/download 任意图片在线转化成BMP图片

https://zh.wikipedia.org/wiki/HSL%E5%92%8CHSV%E8%89%B2%E5%BD%A9%E7%A9%BA%E9%97%B4 HSV和HSL色彩空间

https://www.eecs.qmul.ac.uk/~phao/IP/Images/ 一些计算机视觉常用图片

https://github.com/ShiqiYu/libfacedetection NEON相关的函数实现参考

https://blog.csdn.net/zaishuiyifangxym/article/details/89556318 OpenCV二值化函数说明

https://sites.cs.ucsb.edu/~lingqi/teaching/games101.html 双线性插值、区域采样、图形学相关知识来源

https://zhuanlan.zhihu.com/p/125744132 多种模糊操作介绍

https://blog.csdn.net/sCs12321/article/details/129459772 多种锐化操作介绍