

2025秋CS1003.21计算机程序设计大作业实验报告

C语言数字图像处理

PB25111633 胡运晨

0. 准备工作

0.1 主结构

1. 读入图像（已提供），保存至 `unsigned char* image`
2. 声明输出图像用的指针 `unsigned char* output` 并**申请内存**
3. 输出提示词，并读入操作类型
4. `switch` 至不同task的主函数
5. 写出函数图像（已提供）
6. **释放内存**

代码如下：

```
1  #define _CRT_SECURE_NO_WARNINGS
2  #include <stdio.h>
3  #include <math.h>
4  #include <stdlib.h>
5  #define PI 3.14159265358979323846
6  #define STB_IMAGE_IMPLEMENTATION
7  #define STB_IMAGE_WRITE_IMPLEMENTATION
8  #include "D:\\USTC\\Program\\ImageProcessing\\stb_image.h"
9  #include "D:\\USTC\\Program\\ImageProcessing\\stb_image_write.h"
10 #include "task.h"
11 int main()
12 {
13     int width, height, channels;
14     unsigned char* image =
15     stbi_load("D:\\USTC\\Program\\ImageProcessing\\ImageProcessing\\TestData\\alpha1.jpg", &width, &height, &channels, 1); //读入图像用的函数
16     unsigned char* output = (unsigned char*)malloc(width * height *
17     channels);
18     int ctrl;
19     printf("要进行何种操作？\n1-图像的平移旋转和放缩\n2-图像的直方图均衡化\n3-图像的空间域滤波去噪\n4-彩色图像的K-means算法有损压缩\n5-图像的频域滤波\n6-图像的四叉树分裂合并算法\n");
20     scanf("%d", &ctrl);
21     switch (ctrl) {
22     case 1:
23         task1(image, output, width, height, channels);
24         break;
25     }
```

```
stbi_write_jpg("D:\\USTC\\Program\\ImageProcessing\\ImageProcessing\\TestData\\task1.jpg", width, height, 1, output, 100); //写出函数的图像
```

```

25     stbi_image_free(image);
26     free(output); //释放内存的函数
27     return 0;
28 }

```

0.2 基本函数

1. 坐标转换函数：将输入的行数、列数转换为一维数组中的坐标（坐标=行数*宽度+列数），返回坐标

```

1 int trans(int row, int column, int width) {
2     return row * width + column;
3 }

```

- 2.

1. Task1 图像的平移旋转和放缩

1.1 数据准备

1. 输入的角度为角度制，需要转换为弧度制，实现代码为 `angle = angle * PI / 180.0`，其中 `PI` 为圆周率，已在头文件中定义
2. 行数和列数的计算：使用 `i` 控制计算循环，则行数 `row = i / width`，列数 `column = i % width`
3. 注意到，若进行放缩操作，输出图像的宽高会发生变化，因此利用以下代码计算其宽高

```

1 newWidth = (int)(width * scale);
2 newHeight = (int)(height * scale);

```

4. 在确定 `newWidth, newHeight` 大小后再申请相应的内存

1.2 旋转算法

```

1 originrow = cos(angle) * (row - centrerow) + sin(angle) * (column -
    centrecolumn) + centrerow;
2 origincolumn = -sin(angle) * (row - centrerow) + cos(angle) * (column -
    centrecolumn) + centrecolumn;

```

1. 平移：将现有的行列坐标转换为以输入的中心为原点的坐标（减掉中心点的坐标）
2. 旋转：参考已提供的矩阵旋转公式进行反向旋转，得到在原图像中对应的坐标

1.3 放缩算法

```

1 originrow = row / scale;
2 origincolumn = column / scale; //放缩

```

直接将行列数除以放缩的倍数即可得到对应原图像中的像素点行列数

1.4 最近邻插值法

```
1 nearestNeighbor(image, output, i, round(originrow) * width +  
  round(origincolumn), width, height);  
2 void nearestNeighbor(unsigned char* image, unsigned char* output, int i, int  
  origini, int width, int height) {  
3     if (origini >= width * height || origini < 0) {  
4         output[i] = 0; //边界处理  
5     }  
6     else {  
7         output[i] = image[origini];  
8     }  
9 } //最近邻插值法
```

1. 最近邻: 借助 `<math.h>` 中的 `round` 函数实现四舍五入, 找到最近的点
2. 边界处理: 在原图像坐标之外的点 (`origini >= width * height || origini < 0`) 直接赋0

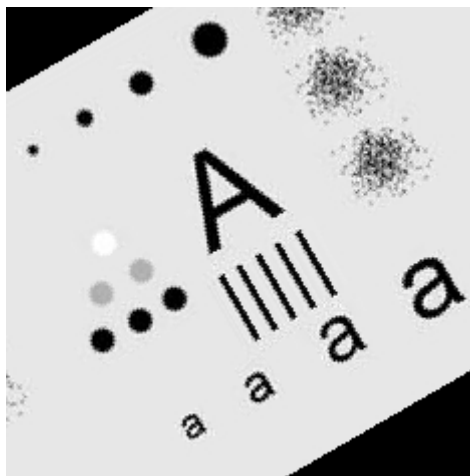
1.5 双线性插值法

```
1 void bilinearNeighbor(unsigned char* image, unsigned char* output, int row,  
  int column, double originrow, double origincolumn, int width, int height) {  
2     if (trans(originrow, origincolumn, width) >= width * height ||  
  trans(originrow, origincolumn, width) < 0) {  
3         output[trans(row, column, width)] = 0; //边界处理  
4     }  
5     else {  
6         int f_a = image[trans((int)originrow, (int)origincolumn, width)];  
7         int f_b = image[trans((int)originrow, (int)origincolumn + 1,  
  width)];  
8         int f_c = image[trans((int)originrow + 1, (int)origincolumn,  
  width)];  
9         int f_d = image[trans((int)originrow + 1, (int)origincolumn + 1,  
  width)];  
10        double f_e = (originrow - (int)originrow) * (f_c - f_a) + f_a;  
11        double f_f = (originrow - (int)originrow) * (f_d - f_b) + f_b;  
12        double f_xy = (origincolumn - (int)origincolumn) * (f_f - f_e) +  
  f_e;  
13        if (f_xy > 255) {  
14            f_xy = 255;  
15        }  
16        else if (f_xy < 0) {  
17            f_xy = 0;  
18        }  
19        output[trans(row, column, width)] = f_xy;  
20    }  
21 } //双线性插值法
```

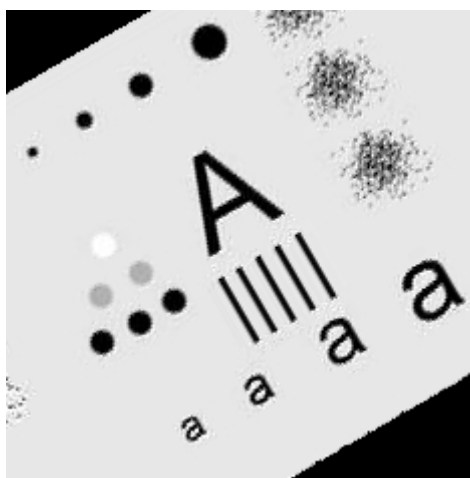
1. 该算法直接翻译自大作业文档, 不再赘述
2. 边界处理: 同1.3.2
3. 插值数据 `f_xy` 溢出: 小于0的数据直接赋0, 大于255的数据直接赋255

1.6 图像展示

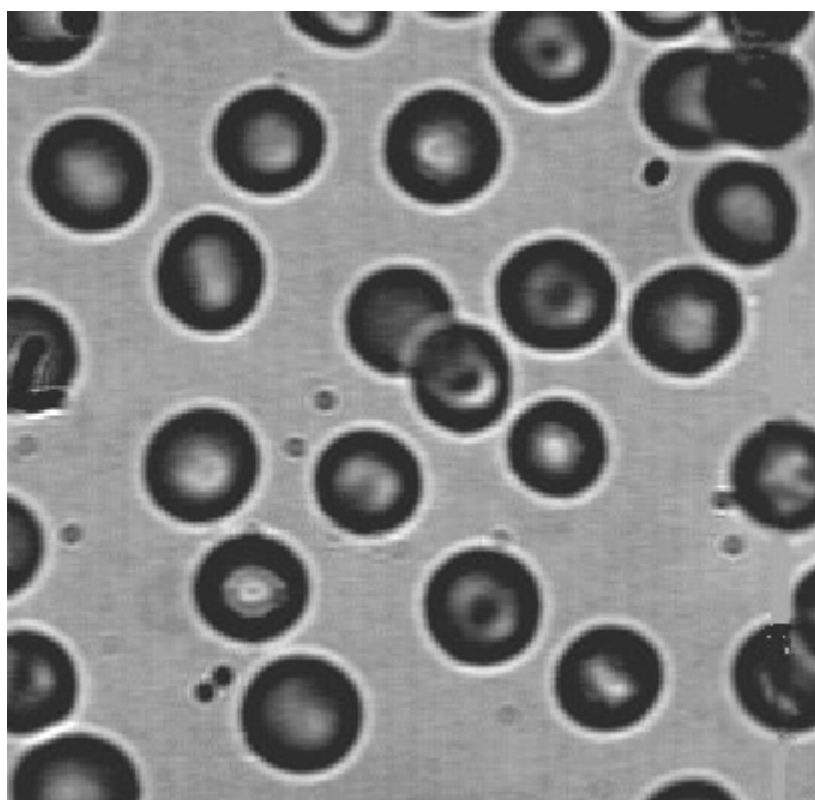
使用最临近插值法绕点(100,100)旋转30度:



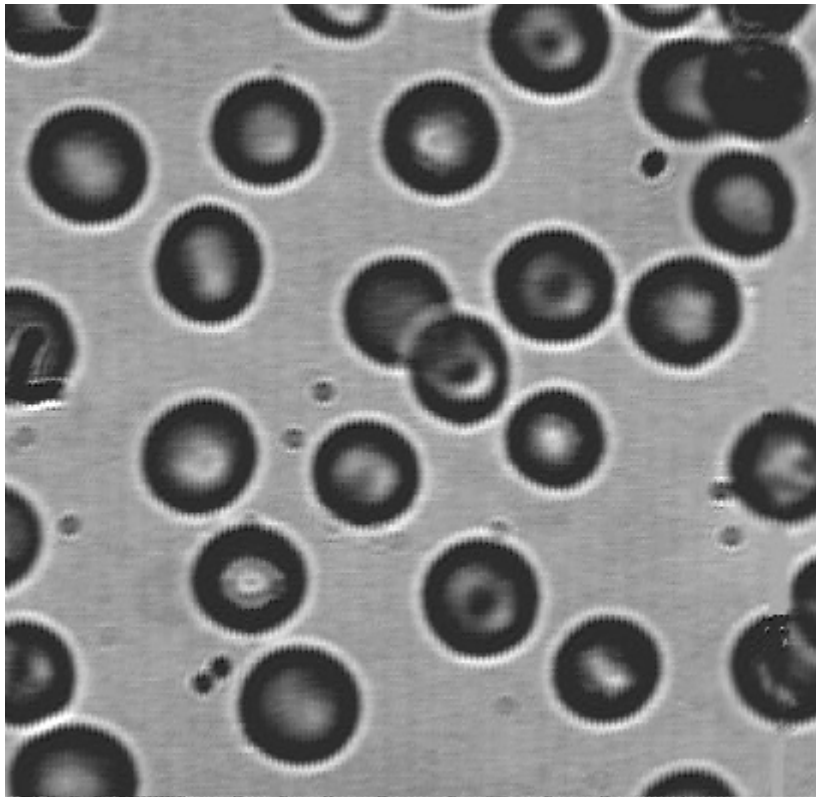
使用双线性插值法绕点(100,100)旋转30度:



使用最临近插值法将图片 blood.jpg 放大1.5倍:



使用双线性插值法将图片 blood.jpg 放大1.5倍:



1.7 插值法视觉效果差异

从两组图像看，相较于最临近插值法，双线性插值法生成的图片边缘锯齿更少，过渡更自然

2. Task2 图像的直方图均衡化

直方图绘制函数在文档中已给出，在此不再赘述，需要注意的是，该函数传入的为灰度值计数而非频率

2.1 灰度值计算

```
1 void generateHist(unsigned char* image, int* hist, int width, int height) {
2     for (int row = 0; row < height; row++) {
3         for (int column = 0; column < width; column++) {
4             hist[image[trans(row, column, width)]]++;
5         }
6     }
7 }
8 } //计算灰度值
```

依次扫描所有像素点，对 hist 中的灰度值计数+1即可

2.2 图像均衡化

```
1 void equalization(unsigned char* image, unsigned char* output, int*
  originHist, int width, int height) {
2     double temp[256] = { 0 };
3     temp[0] = originHist[0];
4     for (int i = 1; i < 256; i++) {
5         temp[i] = temp[i - 1] + originHist[i];
6     }
7     for(int i=0; i < 256; i++) {
```

```

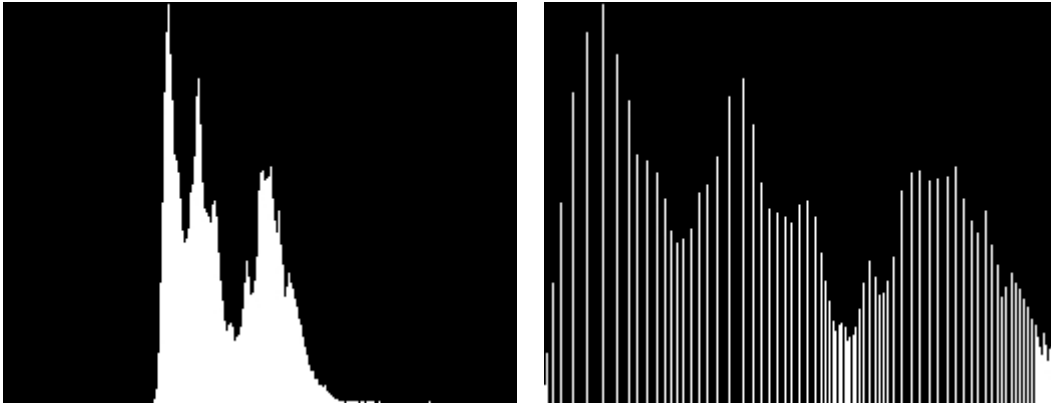
8     temp[i] /= (width * height);
9 }
10 for (int row = 0; row < height; row++) {
11     for (int column = 0; column < width; column++) {
12         output[trans(row, column, width)] = (int)((L - 1) *
temp[image[trans(row, column, width)]] + 0.5);
13     }
14 }
15 }//图像均衡化

```

1. `temp` 用于存储累计频率，先计算累计次数（类似阶乘计算），再除以像素总数得到频率
2. 依据文档中给出的均衡化公式计算新的灰度值，其中 `L` 的值已在头文件中定义

2.3 图像展示

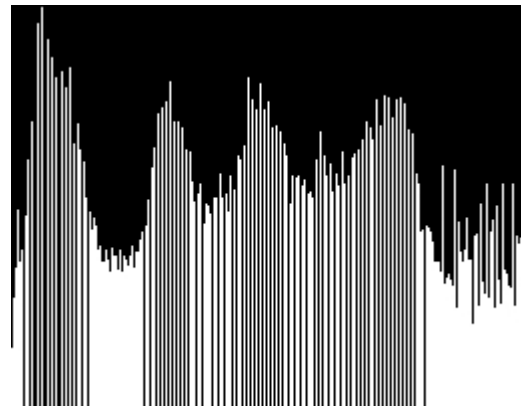
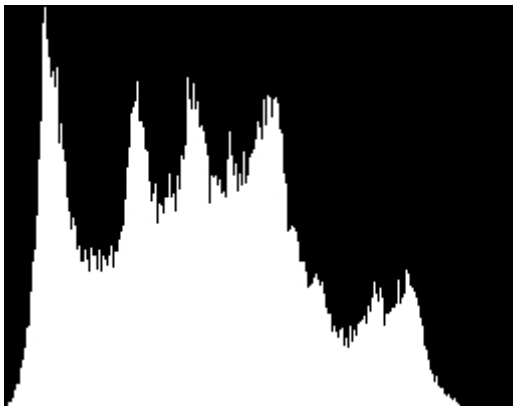
1. `pout.jpg` 均衡化前后的直方图对比



2. `pout.jpg` 均衡化前后视觉效果对比



3. `lena.jpg` 均衡化前后的直方图对比



4. lena.jpg 均衡化前后视觉效果对比



发现均衡化后灰度值分布更加均匀，但保持原有形状基本不变（约等于将直方图拉伸），且均衡化后图像更明亮，明暗对比更明显