**实验报告**

专业：计算机科学与技术

日期：2022/10/28

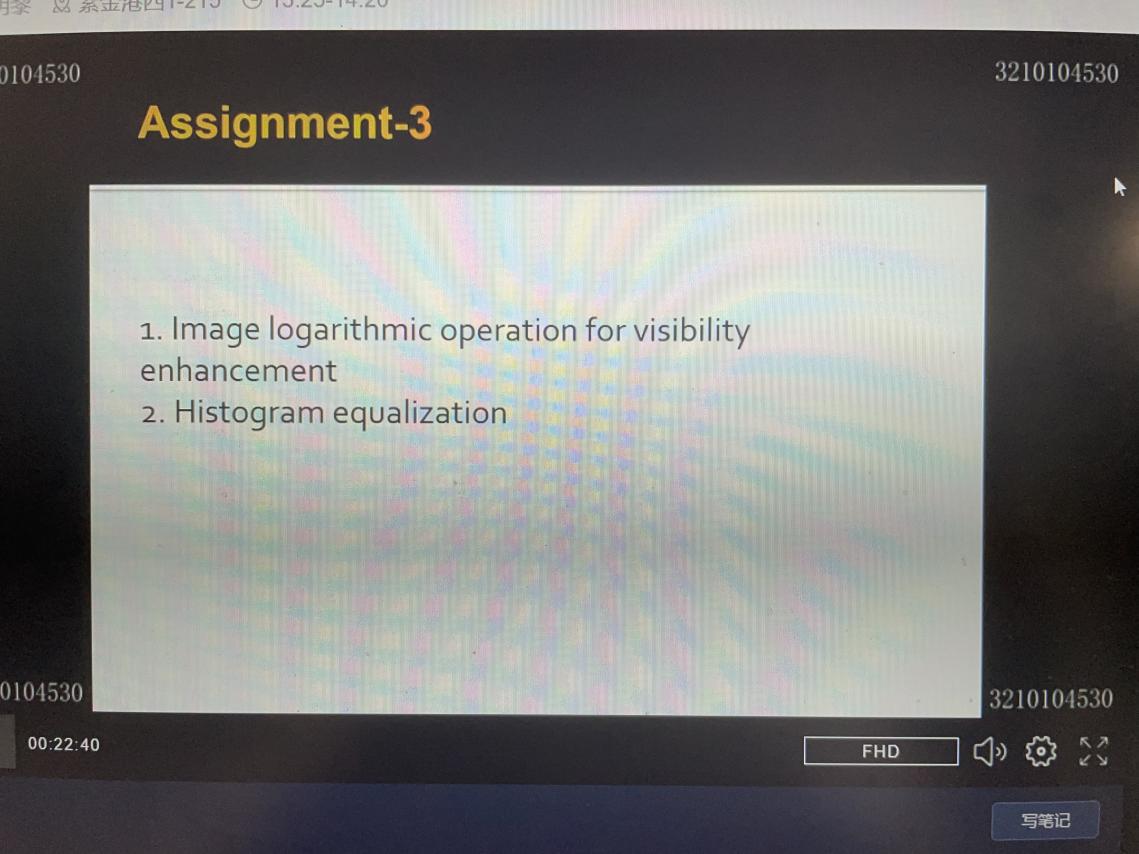
课程名称： 图像信息处理 指导老师： 宋明黎 成绩：

实验名称： 对数变换与直方图均衡化

**一、实验目的和要求**

完成下述实验任务：

1. 以二进制形式读取 .bmp文件头信息及数据信息（256色灰度图像与RGB 24位色图像）
2. 对灰度图像进行对数变换
3. 对灰度图像进行直方图均衡化



**二、实验内容和原理**

.bmp图像信息的贮存，往往有固定的格式，用二进制的方式打开bmp文件，可以设置相对映的数据结构接受文件头部信息、数据头部信息以及palette信息与bitmap data，从而进行相应的数据操作；本实验选用256色（512\*512像素）灰度图片与RGB模式24位色图像文件进行对数变换与直方图均衡化，在牺牲一定细节的前提下，加强图像的对比度与清晰度。通过灰度图像与RGB图像大同小异的处理方法，增进对图像处理进行对数变换或直方图均衡化的理解。

对数变换能将图像中，范围较窄的低灰度值映射为范围较宽的灰度值，或将范围较宽的高灰度值映射为范围较窄的灰度值，适用于扩展图像中的暗像素值，同时压缩更高灰度级的值。

常用公式： new gray level = log(1+original gray level) / log(1 + max gray level).

直方图均衡化的基本思想是对在原始图像中像素个数多的灰度级进行展宽，而对像素个数少的灰度级进行缩减，从而达到清晰增强图像的目的。在RGB图像中，采用三个颜色通道各自平方图均衡化的处理办法

**三、实验步骤与分析**

1.建立存储图像信息的数据结构（头文件）与bmp附带信息基本一一对应。

typedef unsigned char bit;

typedef unsigned short word;

typedef unsigned long dword;

struct tagBITMAPFILEHEADER

{

word bftype; //文件类型

dword bfsize; //文件大小（字节为单位）

word bfreserved1; //保留，必须设为0

word bfreserved2; //保留，必须设为0

dword bfoffbits; //到实际图像实际开始的偏移量

};

struct tagBITMAPINFOHEADER

{

dword bisize; //结构所需字节数

long biwidth; //图像宽度（像素为单位）

long biheight; //图像长度（像素为单位）--正值对映图像倒立

word biplanes; //位面数，常为1

word bibitcounts; //比特/像素比

dword bicompression; //压缩类型

dword bisizeimage; //图像大小

long bixpelspermeter; //水平分辨率

long biypelspermeter; //垂直分辨率

dword biclrused; //颜色索引数

dword biclrimportant; //重要影响颜色索引数

};

struct PALETTE

{

bit R; //存贮调色板中分别索引下的RGB值

bit G;

bit B;

bit reserved; //保留值为必须为0

};

1. 二进制打开.bmp文件，读取文件内数据信息，构建256色颜色索引表，记录bitmap data区域对映像素索引矩阵。

//提取bmp文件头部信息

fp = fopen("lena.bmp", "rb");

fread(&(file.bftype), 1, sizeof(word), fp);

fread(&(file.bfsize), 1, sizeof(dword), fp);

fread(&(file.bfreserved1), 1, sizeof(word), fp);

fread(&(file.bfreserved2), 1, sizeof(word), fp);

fread(&(file.bfoffbits), 1, sizeof(dword), fp);

fread((&information), sizeof(tagBITMAPINFOHEADER), 1, fp);

//获取bmp大小像素信息

picwidth = information.biwidth;

picheight = information.biheight;

//获取调色板索引表---256色位图

PALETTE\* rgblib = (PALETTE\*)malloc(sizeof(PALETTE) \* 256);

fread(rgblib, 256 \* sizeof(PALETTE), 1, fp);

//读入bmp中bitmap data的索引值信息

bit\* rgbgray = (bit\*)malloc(sizeof(bit) \* picwidth \* picheight);

fread(rgbgray, picwidth \* picheight \* sizeof(bit), 1, fp);

1. 对灰度图像进行对数变换

//对数变换

//记录bitmap data的信息

for (i = 0;i < picwidth \* picheight;i++)

logchange[i] = rgbgray[i];

//寻找所有像素中最大灰度值

for (i = 0;i < picwidth \* picheight;i++)

if (logchange[i] > max)

max = logchange[i];

//通过对数值将所有像素灰度值归化为 0 - 1 区间内

for (i = 0;i < picheight \* picwidth;i++)

logre[i] = log(logchange[i] + 1) / log(max + 1);

//寻找所有像素归化值在 0 - 1 间的分布情况

for (i = 0;i < picwidth \* picheight;i++)

{

if (logre[i] > maxi)

maxi = logre[i];

if (logre[i] < mini)

mini = logre[i];

}

result = 255 / (maxi - mini);

//经过对数变化后，重新对每一像素赋灰度值

for (i = 0;i < picheight \* picwidth;i++)

logchange[i] =(logre[i] - mini) \* result;

1. 对灰度图像进行直方图均衡化操作

//直方图均衡化

//初始化数组为 0

for (i = 0;i < 255;i++)

copy[i] = 0;

//记录bitmap data的信息

for (i = 0;i < picwidth \* picheight;i++)

histo[i] = rgbgray[i];

//统计任意灰度总像素数量

for (i = 0;i < picheight \* picwidth;i++)

copy[histo[i]]++;

//求得任意灰度值占总像素值概率

for (i = 0;i < 255;i++)

copy[i] = copy[i] / (picheight \* picwidth);

//进行概率变换对映

for (i = 0;i < 255;i++)

if (i == 0);

else

copy[i] = copy[i] + copy[i - 1];

//根据概率对每一灰度值重新赋值

for (i = 0;i < 255;i++)

copy[i] = copy[i] \* 255;

//对任意像素进行对映灰度函数值转化

for (i = 0;i < picheight \* picwidth;i++)

histo[i] = copy[histo[i]];

1. 将RGB图像中RGB值转化为YUV，再对灰度值进行对数变换，最后由YUV转化为RGB格式写入 bmp 图像文件。

//利用公式将BGR类型转换为YUV类型

for (i = 0;i < picwidth \* picheight;i++)

{

yuvpic[3 \* i] = 0.11 \* rgbpic[3 \* i] + 0.59 \* rgbpic[3 \* i + 1] + 0.3 \* rgbpic[3 \* i + 2];

yuvpic[3 \* i + 1] = 0.493 \* (rgbpic[3 \* i] - yuvpic[3 \* i]);

yuvpic[3 \* i + 2] = 0.877 \* (rgbpic[3 \* i + 2] - yuvpic[3 \* i]);

}

//对数变换

bit max = 0;

float\* logchange = (float\*)malloc(sizeof(float) \* picwidth \* picheight);

double\* logre = (double\*)malloc(sizeof(double) \* picwidth \* picheight);

double maxi = 0, mini = 1;

double result;

//记录图像所有像素的灰度值信息

for (i = 0;i < picwidth \* picheight;i++)

logchange[i] = yuvpic[3 \* i];

//寻找最大灰度值像素

for (i = 0;i < picwidth \* picheight;i++)

if (logchange[i] > max)

max = logchange[i];

//通过对数值将所有像素灰度值归化为 0 - 1 区间内

for (i = 0;i < picheight \* picwidth;i++)

logre[i] = log(logchange[i] + 1) / log(max + 1);

//寻找所有像素归化值在 0 - 1 间的分布情况

for (i = 0;i < picwidth \* picheight;i++)

{

if (logre[i] > maxi)

maxi = logre[i];

if (logre[i] < mini)

mini = logre[i];

}

result = 255 / (maxi - mini);

//经过对数变化后，重新对每一像素赋灰度值

for (i = 0;i < picheight \* picwidth;i++)

logchange[i] = (logre[i] - mini) \* result;

//将YUV格式转化为RGB格式

//若YUV相应转换值大于255，则保留255

//若YUV相应转换值小于0，则保留0

bit\* rgbpic1 = (bit\*)malloc(picheight \* picwidth \* sizeof(bit) \* 3);

for (i = 0; i < picwidth \* picheight; i++)

{

if (logchange[i] + 2.0284 \* yuvpic[3 \* i + 1] < 0)

rgbpic1[3 \* i] = 0;

else if (logchange[i] + 2.0284 \* yuvpic[3 \* i + 1] > 255)

rgbpic1[3 \* i] = 255;

else

rgbpic1[3 \* i] = logchange[i] + 2.0284 \* yuvpic[3 \* i + 1];

if (logchange[i] - 0.3781 \* yuvpic[3 \* i + 1] - 0.5798 \* yuvpic[3 \* i + 2] < 0)

rgbpic1[3 \* i + 1] = 0;

else if (logchange[i] - 0.3781 \* yuvpic[3 \* i + 1] - 0.5798 \* yuvpic[3 \* i + 2] > 255)

rgbpic1[3 \* i + 1] = 255;

else

rgbpic1[3 \* i + 1] = logchange[i] - 0.3781 \* yuvpic[3 \* i + 1]- 0.5798 \* yuvpic[3 \* i + 2];

if (logchange[i] + 1.1403 \* yuvpic[3 \* i + 2] < 0)

rgbpic1[3 \* i + 2] = 0;

else if (logchange[i] + 1.1403 \* yuvpic[3 \* i + 2] > 255)

rgbpic1[3 \* i + 2] = 255;

else

rgbpic1[3 \* i + 2] = logchange[i] + 1.1403 \* yuvpic[3 \* i + 2];

}

1. 对RGB图像分三个通道各自进行直方图均衡化。

//直方图均衡化

int j;

bit\* histor = (bit\*)malloc(sizeof(bit) \* picheight \* picwidth);

bit\* histog = (bit\*)malloc(sizeof(bit) \* picheight \* picwidth);

bit\* histob = (bit\*)malloc(sizeof(bit) \* picheight \* picwidth);

double\* copy = (double\*)malloc(sizeof(double) \* 255);

double\* copy1 = (double\*)malloc(sizeof(double) \* 255);

double\* copy2 = (double\*)malloc(sizeof(double) \* 255);

//初始化数组为 0

for (i = 0;i < 255;i++)

{

copy[i] = 0;

copy1[i] = 0;

copy2[i] = 0;

}

//记录bitmap data的信息

for (i = 0;i < picwidth \* picheight;i++)

{

histob[i] = rgbpic[3 \* i];

histog[i] = rgbpic[3 \* i + 1];

histor[i] = rgbpic[3 \* i + 2];

}

//统计任意灰度总像素数

for (i = 0;i < picheight \* picwidth;i++)

{

copy[histob[i]]++;

copy1[histog[i]]++;

copy2[histor[i]]++;

}

//求得任意灰度总占比概率

for (i = 0;i < 255;i++)

{

copy[i] = copy[i] / (picheight \* picwidth);

copy1[i] = copy1[i] / (picheight \* picwidth);

copy2[i] = copy2[i] / (picheight \* picwidth);

}

//进行概率变换对映

for (i = 0;i < 255;i++)

{

if (i == 0);

else

{

copy[i] = copy[i] + copy[i - 1];

copy1[i] = copy1[i] + copy1[i - 1];

copy2[i] = copy2[i] + copy2[i - 1];

}

}

//根据概率对每一灰度值重新赋值

for (i = 0;i < 255;i++)

{

copy[i] = copy[i] \* 255;

copy1[i] = copy1[i] \* 255;

copy2[i] = copy2[i] \* 255;

}

//对任意像素进行对映灰度函数值转化

for (i = 0;i < picheight \* picwidth;i++)

{

histob[i] = copy[histob[i]];

histog[i] = copy1[histog[i]];

histor[i] = copy2[histor[i]];

}

bit\* rgbpic3 = (bit\*)malloc(sizeof(bit) \* picheight \* picwidth \* 3);

for (i = 0;i < picheight \* picwidth;i++)

{

rgbpic3[3 \* i] = histob[i];

rgbpic3[3 \* i + 1] = histog[i];

rgbpic3[3 \* i + 2] = histor[i];

}

7.写入函数

输入值：1.bmp文件头文件

2.bmp文件数据头文件

3.bmp文件字典索引表

4.bmp文件字典值记录表

5.bmp文件总像素大小（行数乘以列数）

6.bmp文件名称

void write(tagBITMAPFILEHEADER file, tagBITMAPINFOHEADER information, PALETTE\* rgblib, bit\* change,

long int number, const char name[])

{

FILE\* fp;

//打开/新建文件

fp = fopen(name, "wb");

//写入bmp图像各类数据

fwrite(&(file.bftype), 1, sizeof(file.bftype), fp);

fwrite(&(file.bfsize), 1, sizeof(file.bfsize), fp);

fwrite(&(file.bfreserved1), 1, sizeof(file.bfreserved1), fp);

fwrite(&(file.bfreserved2), 1, sizeof(file.bfreserved2), fp);

fwrite(&(file.bfoffbits), 1, sizeof(file.bfoffbits), fp);

fwrite(&information, 1, sizeof(information), fp);

fwrite(rgblib, 1, sizeof(PALETTE) \* 256, fp);

fwrite(change, 1, sizeof(bit) \* number, fp);

//关闭bmp图像文件

fclose(fp);

}

**四、实验环境及运行方法**

用C语言编写，vs正常编译环境运行，其中需要以下语句忽略vs对fopen函数的安全性中断。

#pragma warning(disable:4996) //忽略vs对fopen的安全性错误

对于实验结果，点击运行即可。最后灰度图像与RGB24位色图像各输出两张图片，分别为对数转换（logchange.bmp）bmp图像文件与直方图均衡化（histo.bmp）bmp图像文件。

1. **实验结果展示**



输入灰度图片 输入24位色RGB图片



↑ 对数变换生成图片 ↓ ↑ 直方图均衡化生成图片 ↓

1. **心得体会**

在学习了图像形态学变化的一系列操作后，对数变换与直方图均衡化的学习又使我更进一步地去探索灰度图像的奥秘。

使用这两个变化手段后，图像的对比度与清晰度感觉得到了极大的提升，虽然在一些细节方面被模糊而忽略了；但总体而言，因为对比度地增强，使得灰度图像更加具有立体感、真实感，仿佛渐渐走出了二维的平面，展现的 lena 图像也更加的栩栩如生。

当然，这次的直方图均衡化操作建立在255分块的基础上，我也考虑是否使用其他块数的分级方法：首先，经过数学推导，其他的分块方法在代码实现上会有更大的难度，因为更新后的灰度值或 RGB三通道分别的数值并不是简单的概率乘以255（256色或0 - 255分类）的总色数，而需要用255 / x \* [ a - k ]+A这一公式替代（其中 x 表示分块数目，a 表示当前块对映的新灰度值（RGB颜色通道对映值）块编号，k 为当前块灰度值（RGB颜色通道对映值）编号，A 为当前块初始灰度值（初始RGB颜色通道对映值）），并且减少分块数量，也会对直方图均衡化的最终效果带来负面影响，因而最终还是采用了 255 的分块方法。

受老师课件中彩色图像案例的引导，在做完灰度图像后，我尝试对彩色图像进行实验，也取得了成功，RGB 模式与灰度图像的处理大同小异，在对数变换时，只需要将RGB模式转换为YUV模式，在对Y进行灰度值对数变换操作，最后再将YUV转换为RGB即可；而作直方图均衡化操作时，仅需要简单将RGB模式分通道进行直方图均衡化即可。

另一方面，在学习了直方图的匹配的理论知识后，我也想进行一些尝试，但因为程序跑的结果无法验证，且有点不如人意，就没有附在程序的后面，在直方图的匹配中，若采用255的简单分块方式（对于256 色图像或RGB通道0 - 255分类而言），目标图像的灰度值（RGB数值）可以有效的对映与模范图像的灰度值（RGB数值）函数对应表，但是采用其他分块方法时，这一对映就变得比较复杂，因为能力有限，没办法将其推理清晰与阐述清楚，还请见谅。