

图像信息处理

实验五-均值滤波&拉普拉斯锐化

黄昭阳 3130000696 - December 18, 2015

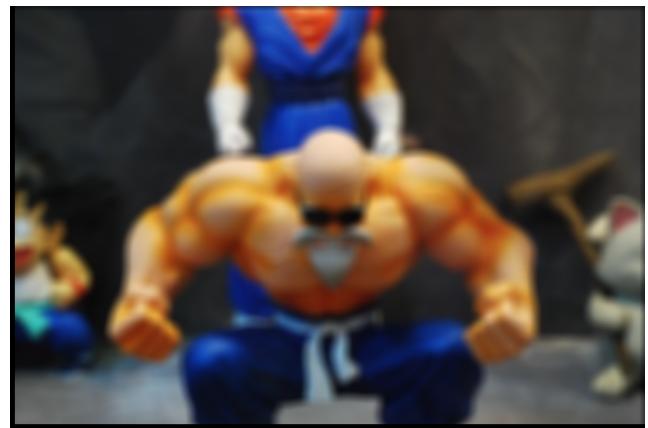
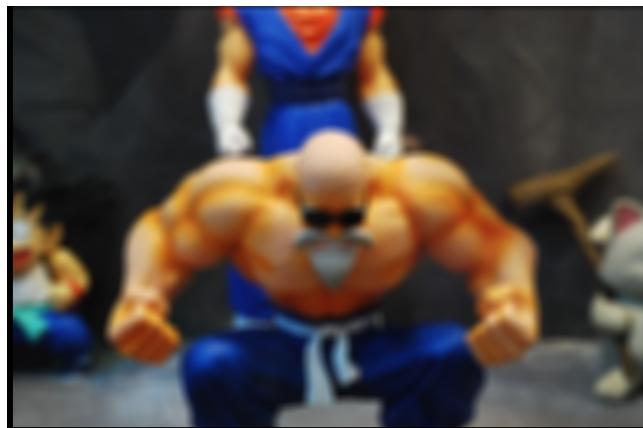


均值滤波

一、结果展示：

做了两种，一种是mean kernel的滤波，一种是gaussian kernel的滤波，其中RGB三通道图像的实现方式是在三个通道上分别做一次滤波得到最终图像，原图大小为1088x728，因此使用 $\sigma = 20$ 的高斯滤波器和宽度为20的平均值滤波器，左下为高斯滤波，右下为平均值滤波，两者相差不大。

原图：



原因分析：

高斯滤波器的sigma值理论上应当是模板半径的1/3，当使用sigma直接作为模板的直径的时候，矩阵值的分布实际上与平均值滤波去的分布相去不远了，以下为sigma = 6时高斯滤波器的分布：

```
0.0558934 0.0590865 0.0607508 0.0607508 0.0590865 0.0558934  
0.0590865 0.0624619 0.0642213 0.0642213 0.0624619 0.0590865  
0.0607508 0.0642213 0.0660302 0.0660302 0.0642213 0.0607508  
0.0607508 0.0642213 0.0660302 0.0660302 0.0642213 0.0607508  
0.0590865 0.0624619 0.0642213 0.0642213 0.0624619 0.0590865  
0.0558934 0.0590865 0.0607508 0.0607508 0.0590865 0.0558934
```

可以观察到值的分布比较相近，但是如果按照严格概率论理论，取sigma=1时，高斯核宽度为6（即半径为3），分布如下：

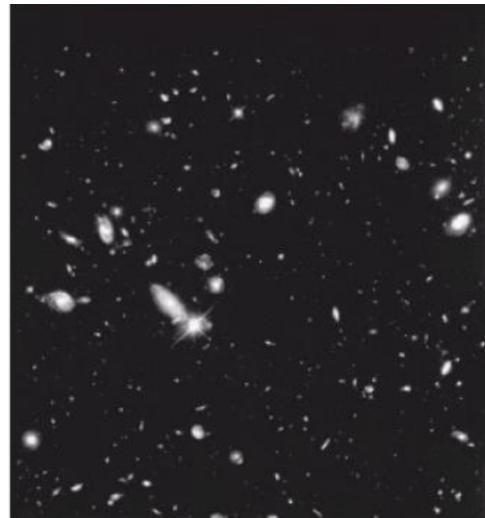
```
0.00077014 0.00569061 0.0154687 0.0154687 0.00569061 0.00077014  
0.00569061 0.0420482 0.114299 0.114299 0.0420482 0.00569061  
0.0154687 0.114299 0.310697 0.310697 0.114299 0.0154687  
0.0154687 0.114299 0.310697 0.310697 0.114299 0.0154687  
0.00569061 0.0420482 0.114299 0.114299 0.0420482 0.00569061  
0.00077014 0.00569061 0.0154687 0.0154687 0.00569061 0.00077014
```

可以观察到值的变化比较大，中心占的权重比较高，最终使用这个核做出来的效果更加符合高斯的预期，但是这又同样带来一个问题，这个核滤波的效率太低了，想要增加效率，其一可以使用该核做多次滤波；其二可以增加sigma的值使得核的宽度变大。但是这两种做法都需要耗费更多的时间，因此遇到具体问题时，应该根据问题的需求不同来做不同的抉择，sigma与核宽度的倍率关系做相应调整。

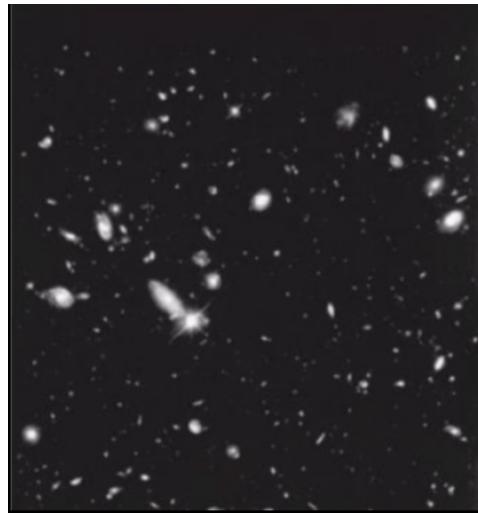
下方羽毛图是使用width = 9时做出来的效果图。图为544x584，去核的宽度为6，即高斯函数sigma=1.5，可以明显的观察到两者之间的区别。

本次只展示了彩图的滤波，也可以先调用程序中的transferToGreyBMP，然后再调用相关接口进行灰度图的滤波

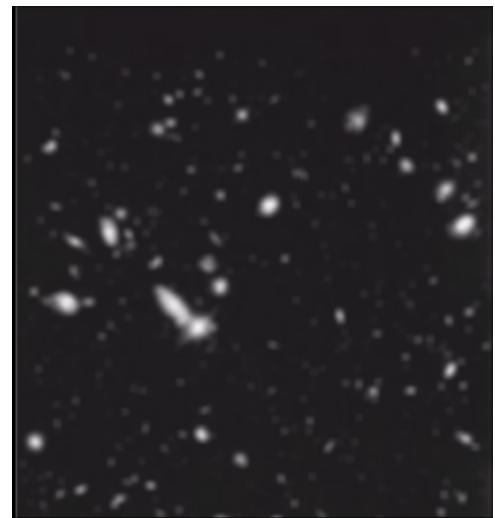
原图：



高斯：



均值：



二、代码分析

1. 均值滤波代码分析：

```
void bmpFile::MeanFilter(int length)
{
    float sum = length*length;
    //sum为矩阵总和
```

```

Matrix* filter = Matrix::ones(length);
//获取一个矩阵值全为1的矩阵作为滤波器
Matrix m[3] = {
    Matrix(bmpInfoHeader.biHeight,bmpInfoHeader.biWidth),
    Matrix(bmpInfoHeader.biHeight,bmpInfoHeader.biWidth),
    Matrix(bmpInfoHeader.biHeight,bmpInfoHeader.biWidth)
};
//用来存储滤波过后的图像值, 因为有可能是三个通道 (RGB) , 因此生成三个矩阵
int times = 1;
if(bmpInfoHeader.biBitCount == 24) times = 3;
//判断如果是24位图则是三通道, 否则为单通道灰度图, times即通道数
for(int k=0;k<times;k++)
{
    for(int i=0;i<bmpInfoHeader.biHeight;i++)
        for(int j=0;j<bmpInfoHeader.biWidth;j++)
            m[k][i][j] = Convolution(filter,i,j,times,k)/sum;
//Convolution即卷积操作, 使用滤波器filter以(i,j)为中心, 在times通道的情况下, 对第
//k通道进行滤波操作
    for(int i=0;i<bmpInfoHeader.biHeight;i++)
        for(int j=0;j<bmpInfoHeader.biWidth;j++)
            bmpFileData[i][j*times+k] = fabs(bmpFileData[i][j*times
+k]-(unsigned char)m[k][i][j])*3;
//此处生成中间的halos成分, 由于老师不做要求, 这里可以忽略。
}
exportToFile("halos.bmp");
//将halos成分删除到halos.bmp文件
for(int k=0;k<times;k++)
{
    for(int i=0;i<bmpInfoHeader.biHeight;i++)
        for(int j=0;j<bmpInfoHeader.biWidth;j++)
            bmpFileData[i][j*times+k] = (unsigned char)m[k][i][j];
}
//按既定通道数将所有的滤波后的值赋值给bmp文件的数据区
delete filter;
//删除滤波器
}

```

2. 高斯滤波代码分析:

void bmpFile::GaussianFilter(float sigma); 实现高斯核滤波, 由于程序与mean滤波大致相似, 只不过核不同, 这里不做赘述, 只是将高斯核的生成进行解释。

```

float Matrix::GaussianFunction(float sigma, float dis)
{
    return 1/(sigma*sqrt(2*PI))*exp(-sqr(dis)/(2*sqr(sigma)));
}
//根据给定的sigma和dis生成高斯函数

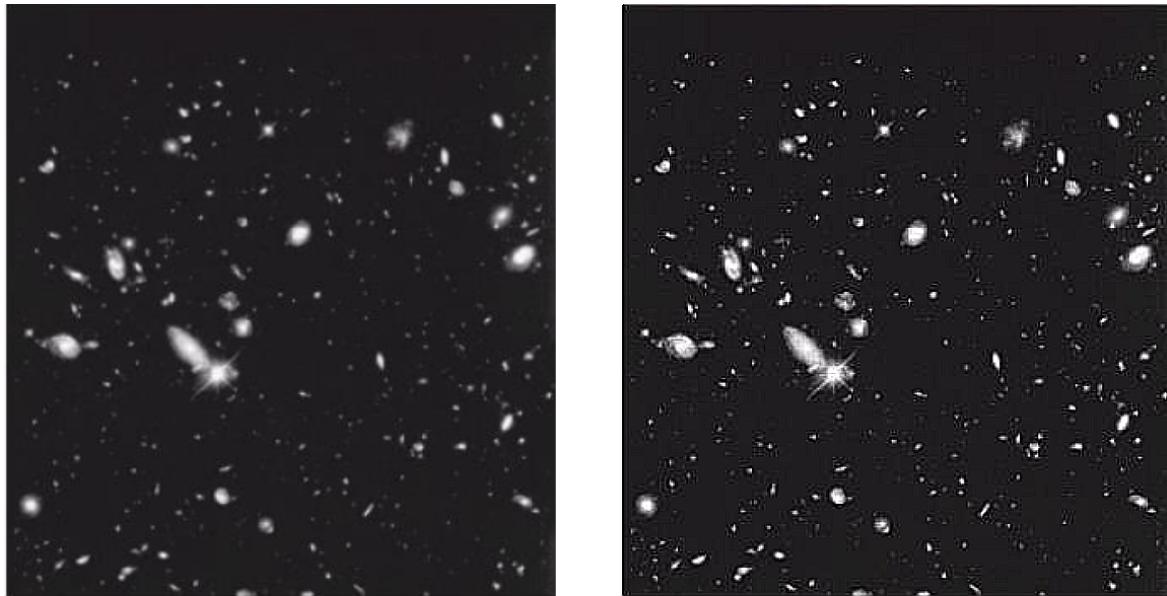
```

```
Matrix* Matrix::GaussianFilter(float sigma)
{
    if(sigma <= 0 )
    {
        #ifdef DEBUG
            cout << "Matrix::GaussianFilter" << endl;
            cout << " Sigma = " << sigma << endl;
        #endif
        return NULL;
    }
    Matrix* resultMatrix = new Matrix(int(sigma+0.5),int(sigma+0.5));
    //按照sigma大小四舍五入作为高斯核的矩阵宽度
    float center = resultMatrix->Row*1.0/2;
    //获取中心位置
    float dis;
    for(int i=0;i<resultMatrix->Row;i++)
        for(int j=0;j<resultMatrix->Col;j++)
    {
        dis = sqrt(sqrt(center-i-0.5)+sqrt(center-j-0.5));
        //计算(i,j)像素点距离矩阵中心的距离
        resultMatrix->data[i][j] = GaussianFunction(sigma,dis);
        //根据距离生成所在位置的权值
    }
    return resultMatrix;
    //返回生成的Gaussian Kernel
}
```

拉普拉斯图片增强

一、结果展示：

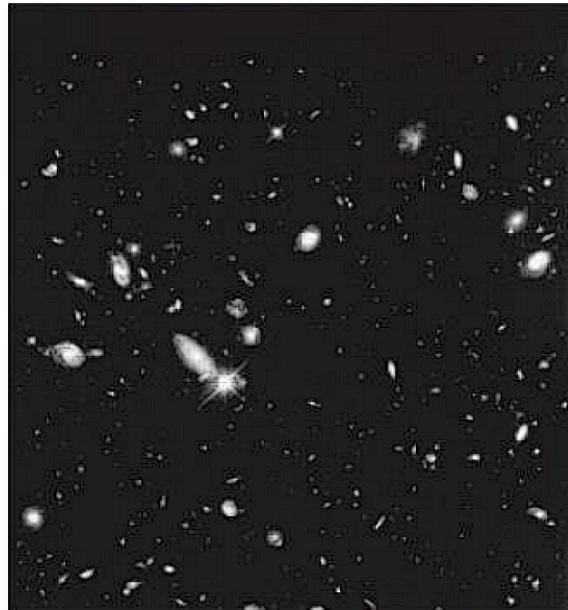




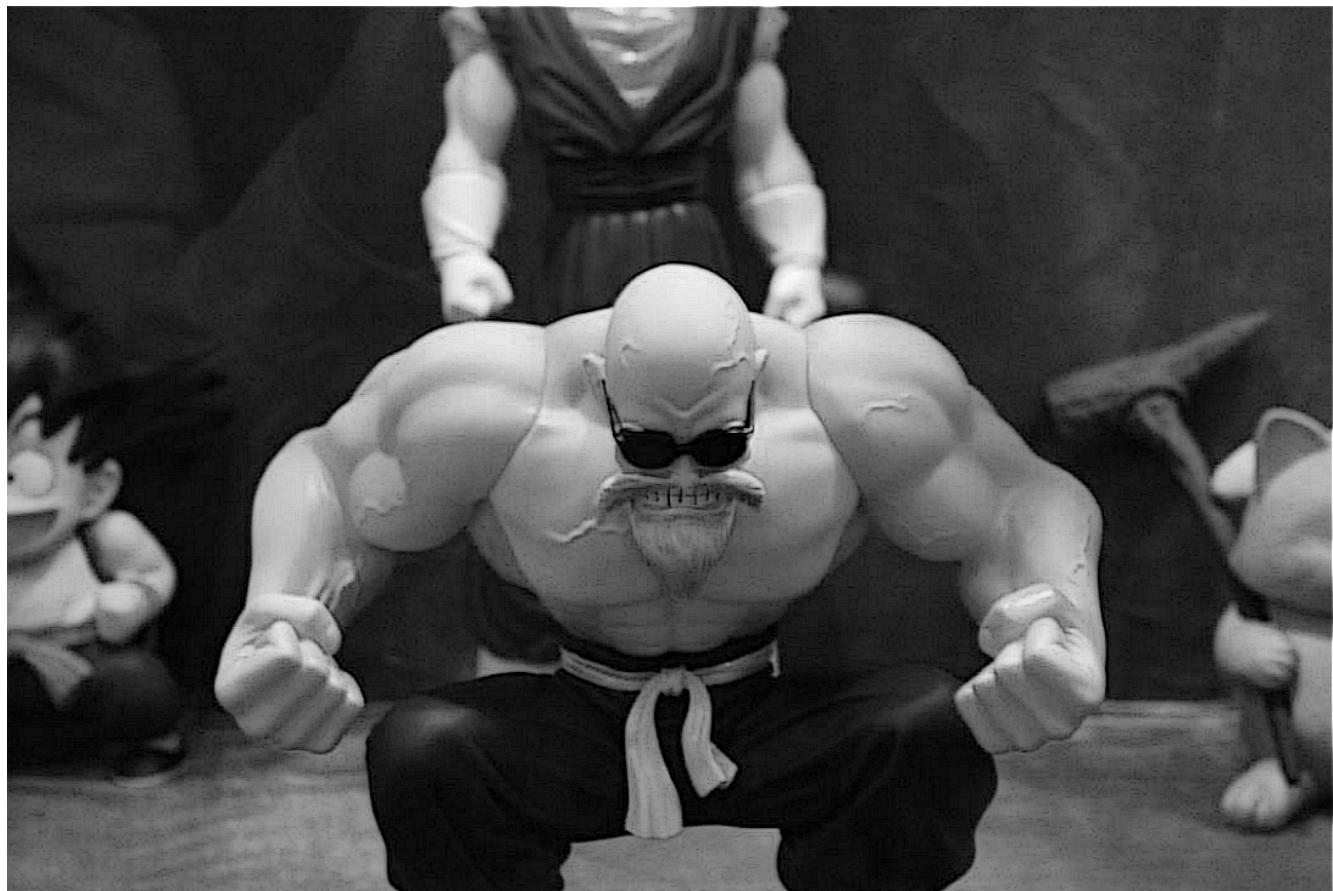
以上两张图是没有做归一化处理的拉普拉斯锐化，可以看出效果十分明显。

下面是分别是两张图对应的拉普拉斯之后，若拉普拉斯所得的值为负数，则rearrange到-255~0，如为正数则rearrange到0~255，这样可以保证整体值相对稳定，而像素intensity的值加一步拉大。





以及灰度图赏的拉普拉斯增强。下一为增强的图，下二为灰度图。





以上所有的图都可以在picture目录下找到。

二、代码分析。

```
void bmpFile::LaplacianEnhancement()
{
    int length = 3;
    Matrix* filter = Matrix::Laplacian(length);
//获取拉普拉斯kernel
    Matrix m(bmpInfoHeader.biHeight,bmpInfoHeader.biWidth);
    int times = 1;
    float min = 10000,max = -10000;
//记录两段极值用来做rearrange
    float gap;
//值域分布范围
    float range;
//rearrange的因子, 决定rearrange的范围
    if(bmpInfoHeader.biBitCount == 24) times = 3;
//如果是彩图则需要对3个维度分别做拉普拉斯 (也可以转到yuv对亮度做)
```

```

    for(int k=0;k<times;k++)
    {
        for(int i=0;i<bmpInfoHeader.biHeight;i++)
            for(int j=0;j<bmpInfoHeader.biWidth;j++)
            {
                m[i][j] = Convolution(filter,i,j,times,k);
            }
    }
    //使用拉普拉斯kernel做卷积
    if(min>m[i][j]+bmpFileData[i][j*times+k]) min = m[i][j]
    +bmpFileData[i][j*times+k];
    if(max<m[i][j]+bmpFileData[i][j*times+k]) max = m[i][j]
    +bmpFileData[i][j*times+k];
    //取极值, 用来判断gap的大小
    }
    gap = max - min;
    //获取gap, 即m的值域分布大小
    range = 255;
    for(int i=0;i<bmpInfoHeader.biHeight;i++)
        for(int j=0;j<bmpInfoHeader.biWidth;j++)
        {
            if(m[i][j]<0) m[i][j] = -m[i][j]/min*range;
            else m[i][j] = m[i][j]/max*range;
        }
    //对于正数与负数做不同的rearrange, 拉开两者的差距的同时也保证了整体灰度的稳定
    for(int i=0;i<bmpInfoHeader.biHeight;i++)
        for(int j=0;j<bmpInfoHeader.biWidth;j++)
            bmpFileData[i][j*times+k] = clip(int(m[i][j]
            +bmpFileData[i][j*times+k]));
    //因为直接相加可能存在溢出, 使用clip将两者fuse在一起
    }
}

```