杭州電子科技大学

数 字 图 像 处 理

（研 2018 级）

班　　级	自动化 6 班
姓　　名	陈安琪
学　　号	181060065
任课教师	武薇
完成日期	2018 年 11 月 22 日

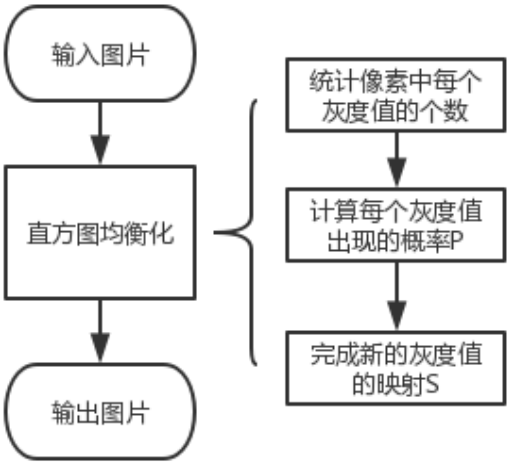
1. 直方图均衡化

1.1 图像分析

观察“image1-1.jpg”和“image1-2.png”两幅医学影像 CT 图，图片整体看上去偏暗，骨骼的结构不够清晰，曝光不足，应该是灰度级过于集中，想到用直方图均衡化处理。

1.2 实验步骤

流程图：



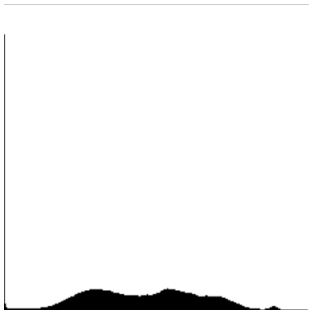
映射方法的函数如下：

$$s_k = \sum_{j=0}^{k} \frac{n_j}{n} \quad k = 0, 1, 2, \cdots, L-1$$

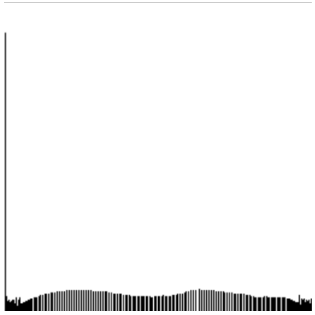
1.3 处理前后对比

直方图分布的结果图如下图 1 所示：

图 1



原图直方图



均衡化后的直方图

图片的显示结果如下图 2 所示：

图 2

原图

自己写的函数

OpenCV 自带函数

从上面两图可以看出，均衡化后的直方图分布一般都不是均匀的，均衡化的图像有明显的增强，骨骼的结构更加的清晰，自己写的函数，没有自带的函数效果好，我觉得可能是灰度值为小数的时候，我直接强制类型转化成 int 型，这中间可能存在一定问题。

1.4 结论

直方图均衡化对于背景和前景都太亮或者太暗的图像非常有用，这种方法尤其是可以带来 X 光图像中更好的骨骼结构显示以及曝光过度或者曝光不足照片中更好的细节。计算量较少，当然也会损失一些细节。

1.5 核心代码

```
Mat HistogramEqual(Mat Img){

     Mat Out = Img;
    int ImgSize = Img.cols * Img.rows;
    double Gray_Level_New[256];          //存放新的灰度级
    int GrayVal_Add[256];                //存放各个灰度值的个数
    double P[256];                       //存放各级灰度值所占的比列 也就是概率


    cvtColor(Out, Out, CV_BGR2GRAY);

    /*初始化各参数*/
    int idx = 0;
    for (int i = 0; i < 256; i++){
        GrayVal_Add[i] = 0;
        P[i] = 0;
        Gray_Level_New[i] = 0;

    }


    /*统计每个灰度值的个数*/
    for (int i = 0; i < Img.rows; i++)
        for (int j = 0; j < Img.cols; j++){
            GrayVal_Add[Out.ptr<uchar>(i)[j]] += 1;
        }

    for (int i = 0; i < 256; i++){
        P[i] = (double)GrayVal_Add[i] / ImgSize;
    }


    /*建立新的灰度级别的数组*/
    double sum = 0;
    for (int i = 0; i < 256; i++){

        sum += P[i];
        Gray_Level_New[i] = sum * 255;
    }

    /*结果*/
    for (int i = 0; i < Out.rows; i++){
        for (int j = 0; j < Out.cols; j++){
```

```
44.             Out.ptr<uchar>(i)[j] = (int)Gray_Level_New[(int)Out.ptr<uchar>(i)[j]];
45.         }
46.     }
47.
48.     return Out;
49. }
```

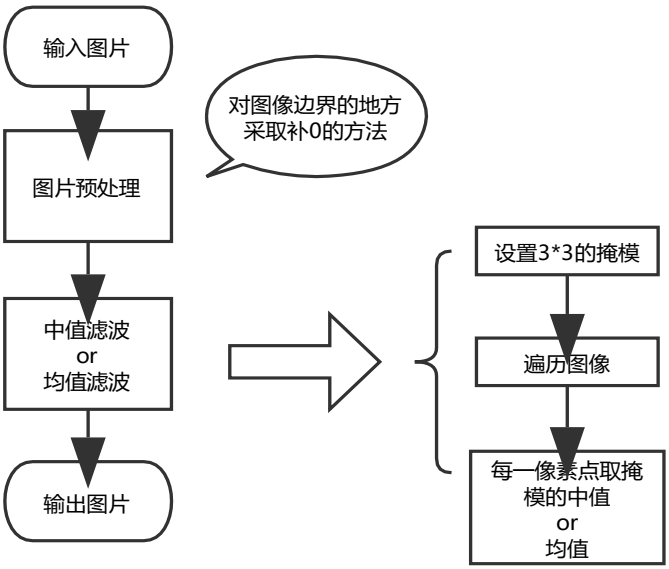
2. 中值滤波和均值滤波

2.1 图像分析

观察图像“image4.jpg”，可以看到图像上有很多一粒一粒的点（黑白噪点），属于椒盐噪声，自然想到用中值滤波，顺便做一个均值滤波的比较。

2.2 处理流程

流程图：



2.3 处理前后对比

图 3

原图　　中值滤波　　均值滤波

2.4 结论

观察图像知道，中值滤波和均值滤波，都有起到平滑图像，滤去噪声的功能，但是，对于椒盐噪声，中值滤波通过选取合适的点来代替噪点，它所表现的处理效果更好，跟书上说的结论一致。

2.5 核心代码

中值滤波器的代码：

```
1.      Mat ZhongZhi(Mat Img){
2.      Mat Img2_Border;
3.      cvtColor(Img, Img, CV_BGR2GRAY);
4.      copyMakeBorder(Img, Img2_Border, 1, 1, 1, 1, BORDER_CONSTANT, cv::Scalar(0, 0, 0));
5.      //对图像进行扩充，对图像边缘增加一圈灰度为 0 的像素点
6.      int A[9];
7.      int temp = 0;
8.      //namedWindow("1");
9.      //imshow("1", Img2_Border);
10.     for (int i = 1; i < Img2_Border.rows - 1; i++){
11.         for (int j = 1; j < Img2_Border.cols - 1; j++){
12.
13.             /* 这是 3*3 的掩模 用数组 A[]来存储掩模的灰度值 */
14.             /* 代码写得有点臭 复杂度高 */
15.             A[0] = Img2_Border.ptr<uchar>(i - 1)[j - 1];
16.             A[1] = Img2_Border.ptr<uchar>(i - 1)[j];
17.             A[2] = Img2_Border.ptr<uchar>(i - 1)[j + 1];
18.             A[3] = Img2_Border.ptr<uchar>(i)[j - 1];
19.             A[4] = Img2_Border.ptr<uchar>(i)[j];
20.             A[5] = Img2_Border.ptr<uchar>(i)[j + 1];
21.             A[6] = Img2_Border.ptr<uchar>(i + 1)[j - 1];
22.             A[7] = Img2_Border.ptr<uchar>(i + 1)[j];
23.             A[8] = Img2_Border.ptr<uchar>(i + 1)[j + 1];
24.
25.             /**************  插入排序  ***************/
26.             for (int m = 0; m<9; m++)
27.                 for (int n = m; (j>0 && A[n]<=A[n - 1]); n--){
28.                     temp = A[n - 1];
29.                     A[n - 1] = A[n];
30.                     A[n] = temp;
31.                 }
32.             Img2_Border.ptr<uchar>(i)[j] = A[4]; //选择中值
33.         }
```

```
34.
35.     }
36.     return Img2_Border;
37. }
```

均值值滤波器的代码：

```
1.      Mat JunZhi(Mat Img){
2.       Mat Img2_Border;
3.      cvtColor(Img, Img, CV_BGR2GRAY);
4.      copyMakeBorder(Img, Img2_Border, 1, 1, 1, 1, BORDER_CONSTANT, cv::Scalar(0, 0, 0));
5.          //对图像进行扩充，对图像边缘增加一圈灰度为 0 的像素点
6.
7.      for (int i = 1; i < Img2_Border.rows - 1; i++)
8.          for (int j = 1; j < Img2_Border.cols - 1; j++){
9.              /* 这是 3*3 的掩模    代码写得还是很臭 */
10.             Img2_Border.ptr<uchar>(i)[j] =
11.                             (Img2_Border.ptr<uchar>(i-1)[j-1] + \
12.                              Img2_Border.ptr<uchar>(i-1)[j] + \
13.                              Img2_Border.ptr<uchar>(i-1)[j+1] + \
14.                              Img2_Border.ptr<uchar>(i)[j-1] + \
15.                              Img2_Border.ptr<uchar>(i)[j] + \
16.                              Img2_Border.ptr<uchar>(i)[j+1] + \
17.                              Img2_Border.ptr<uchar>(i+1)[j-1] + \
18.                              Img2_Border.ptr<uchar>(i+1)[j] + \
19.                              Img2_Border.ptr<uchar>(i+1)[j+1])/(9.0);
20.         }
21.     return Img2_Border;
22. }
```

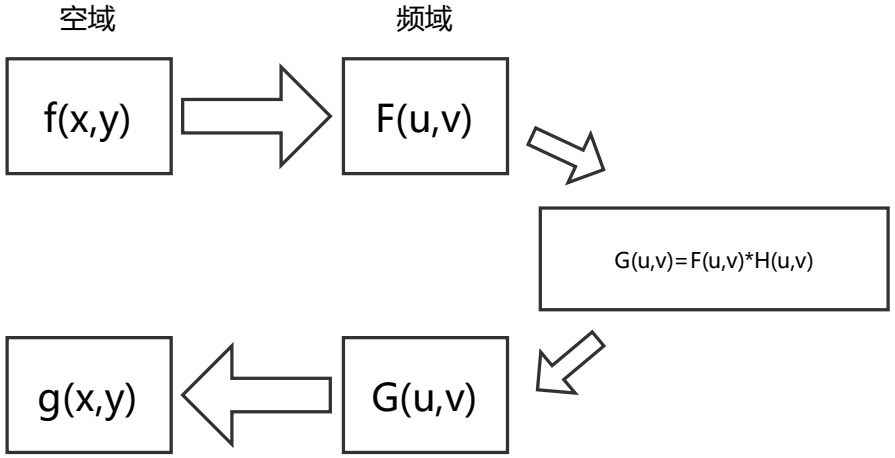
3. 低通滤波

3.1 图像分析

观察图“image2-1.jpg”和“image2-2.gif”两幅图片，存在条纹状和锯齿状的噪声，由于噪声变化的程度有明显差异，想到了频域的方法，试着用理想低通滤波器处理它。

3.2 处理流程

频域图像增强的基本过程如下图：



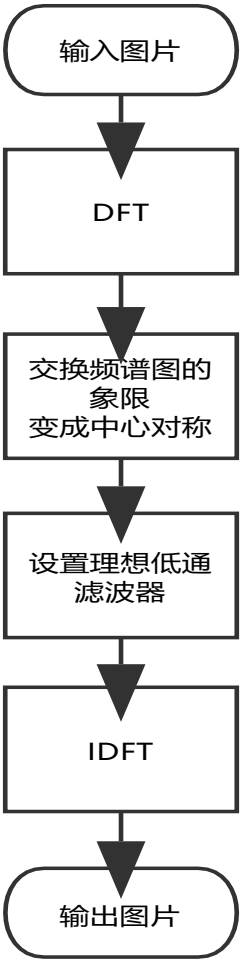
频率图像处理的关键级是 H(u,v)的设置。

理想低通滤波的 H(u,v)如下所示：

$$H(u,v)=\begin{cases}1 & D(u,v)\le D_0\\ 0 & D(u,v)>D_0\end{cases}$$

其中 D 表示频谱图像中，每个像素点距离频谱图原点(原图 （长/2，宽/2)的位置)的距离。D0 为我们自己设置的滤波范围。

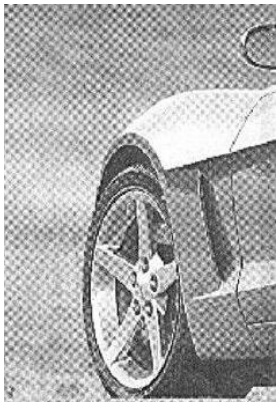
理想低通滤波的流程图：



3.3 处理前后对比

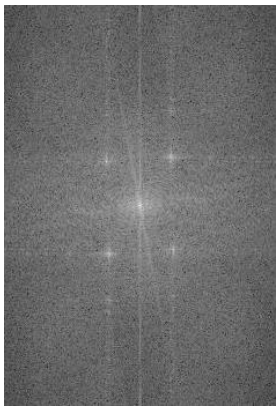
处理前后的图片如图 3 所示：

图 3

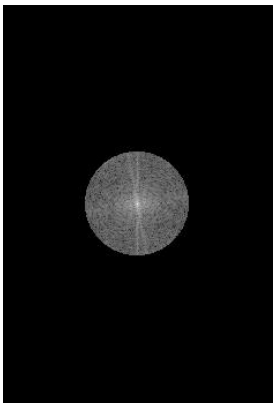


原图

理想低通处理后



原图频谱图



处理后的频谱图

3.4 结论

观察图像可以知道，我们可以看到图像明显平滑了很多，低通滤波器的效果就是让低频信号通过，而把高频信号过滤掉。

换句话说就是高频信号（变化比较剧烈，如边缘，锯齿等）会被过滤掉，对图像起到平滑作用。

3.5 核心代码

```
img = imread('C:\Users\NH3\Desktop\数字图像处理\image2-1.jpg');
img = rgb2gray(img);
figure(2);
imshow(img);

img_fft = fft2(img);
img_fftshift = fftshift(img_fft);

figure(1);
imshow(log(abs(img_fftshift)),[]);

img_size = size(img_fftshift);
N = img_size(1)/2;
M = img_size(2)/2;

%%%%%%%%%%%%%%%%%%%%%%%%理想低通滤波%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
D = 45;   %%% 参数 D 可以调，其实这些滤波器都很像，也可以设置成环形，类似带通滤波
for n = 1:img_size(1)
    for m = 1:img_size(2)
        dist = sqrt(abs(m - M)*abs(m - M)+ abs(n - N)*abs(n - N));
        if (dist > D )
            img_fftshift(n,m) = 0;
        end
    end
end
figure(3)
imshow(log(abs(img_fftshift)),[]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

img1 = ifftshift(img_fftshift);
img2 = ifft2(img1);
img = uint8(real(img2));
figure(4);
imshow(img);
```

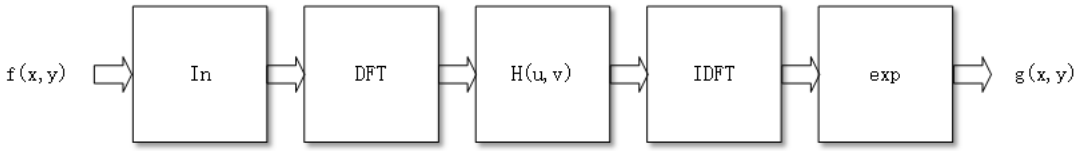
4. 同态滤波

4.1 图像分析

观察图像“image3-1.tif”和“image3-2.jpg”,可以很容易的想到，这组图片是由于光照不均匀引起的，我们所希望看到的细节的灰度很暗，图片的层次感很差，所以决定用同态滤波来处理这类图片，处理的过程其实和前面的低通滤波类似。

4.2 处理流程

同态滤波的算法流程如下图所示：



对于一副图像 f(x,y)可由照射分量 i(x,y)和反射分量 r(x,y)的乘积，即

$$f(x,y) = i(x,y)r(x,y)$$

上式不能直接用于对照度和反射的频率分量进行操作，因此上式取对数

$$Inf(x,y) = lni(x,y) + lnr(x,y)$$

对上式两边取傅里叶变换：

$$\Im\{Inf(x,y)\} = \Im\{Ini(x,y)\} + \Im\{Inr(x,y)\}$$

图像的照射分量通常由慢的空间变化来表征，而反射分量往往引起突变，特别是在不同物体的连接部分。这些特性导致图像取对数后的傅里叶变换的低频成分与照射相联系，而高频成分与反射相联系。

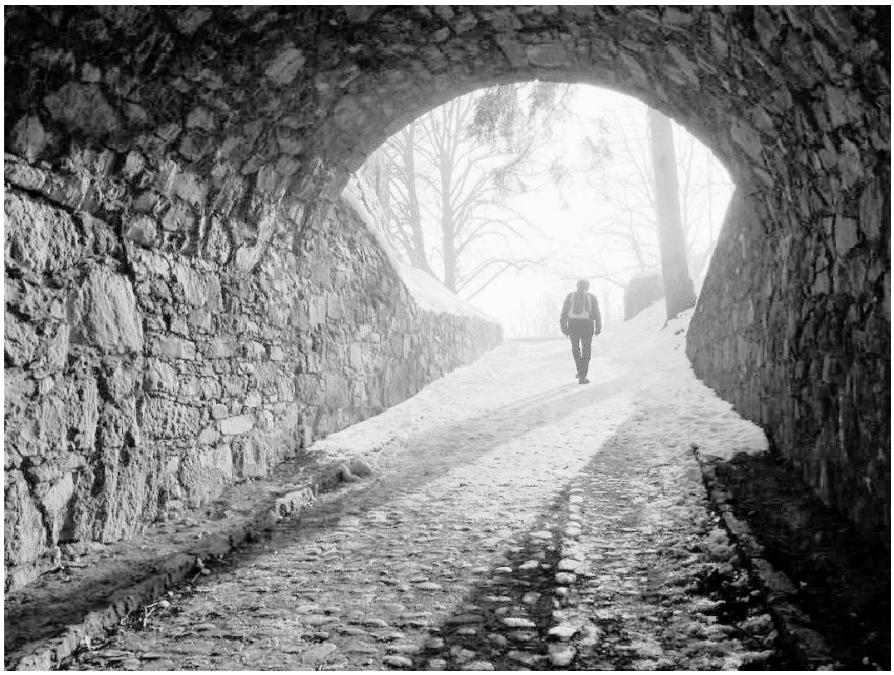
使用同态滤波器可以更好地控制照射分量和反射分量。这种控制器需要指定一个滤波器函数 H(u,v)，它可用不同的可控方法影响傅里叶变换的低频和高频。如果 γ L 和 γ H 选定，而 γ L<1 且 γ H>1，那么滤波器函数趋近于衰减低频（照射）的贡献，而增强高频反射的贡献。最终结果是同时进行动态范围的压缩和对比度的增强。

4.3 处理前后对比

处理结果如图 4 所示：

图 4

原图



同态滤波+直方图均衡

处理后的图片的细节明显更容易分辨了，明显有光照的感觉！我试着调了一下 rH,rL,c 等参数，发现过了某一值，图片就整体变白，但是整体上，参数的调节对图片没有明显效果，最后的单单同态滤波后的，比较好的结果，还是整体偏灰（如下图 4.3.1 所示），所以再用到直方图均衡化来处理。

图 4.3.1

4.4 结论

图像的同态滤波是把频率过滤和灰度变换结合起来的一种图像处理方法，其是以图像的照度/ 反射率模型作为频域处理的基础，通过调整图像灰度范围和增强对比度来改善图像的质量。使用这种方法可以使图像处理符合人眼对于亮度响应的非线性特性，避免了直接对图像进行傅立叶变换处理的失真。该方法消除图像上照明不均的问题，增强暗区的图像细节，同时又不损失亮区的图像细节。

4.5 Matlab 代码

```
img = imread('C:\Users\NH3\Desktop\数字图像处理\image3-2.jpg');
img_size = size(img);
figure('Name','原图');
% img = imresize(img,0.5);
imshow(img);

img_fft = fft2(log(double(img) + 1));
img_fftshift = fftshift(img_fft);
figure('Name','fft');
imshow(img_fftshift);

%%%%%%%%%%%%%%%% 这一段参数可调 %%%%%%%%%%%%%%%%%%%%%%
D0 = 2000;   %max = (768/2)^2
rH = 1;
rL = 0.15;
c = 1;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
M = img_size(2) / 2;    %(M,N)为频谱图像的中心点
N = img_size(1) / 2;
H = zeros(img_size(1),img_size(2));
for v = 1:img_size(2)
    for u = 1:img_size(1)
        H(u,v) = (rH - rL) * (1 - (exp( -c * ((u-M)^2+(v-N)^2) / (D0^2)))) + rL;
    end
end
img_homo_pinyu = img_fftshift .* H;
img_homo_real = real(ifft2(ifftshift(img_homo_pinyu)) - 1);
img_homo_kongyu = exp(img_homo_real);  %结果偏灰，灰度级过于集中，想到直方图均衡化

img_result = histeq(img_homo_kongyu);  %之前已经写过直方图均衡了，现在直接调用


figure('Name','同态结果+直方图均衡');
imshow(img_result);
```