【图像处理】-012 同态滤波

在上一篇中，在实现底帽变换用于校正不均匀光照引起的变化时，发现使用同态滤波也可以达到同样的效果，因此，对同态滤波进行了一些调研。

1 理论依据

1.1 图像形成模型

形如$f(x,y)$的二维函数用于表示图像，在空间坐标(x,y)处，f的值或幅度是一个正的标量，其物理意义由图像源决定。当一幅图像由物理过程产生时，其亮度值正比于物理源（例如电磁波）所辐射的能量。因此，$f(x,y)$一定是非零的和有限的，即

$$0 < f(x,y) < \infty \tag{1}$$

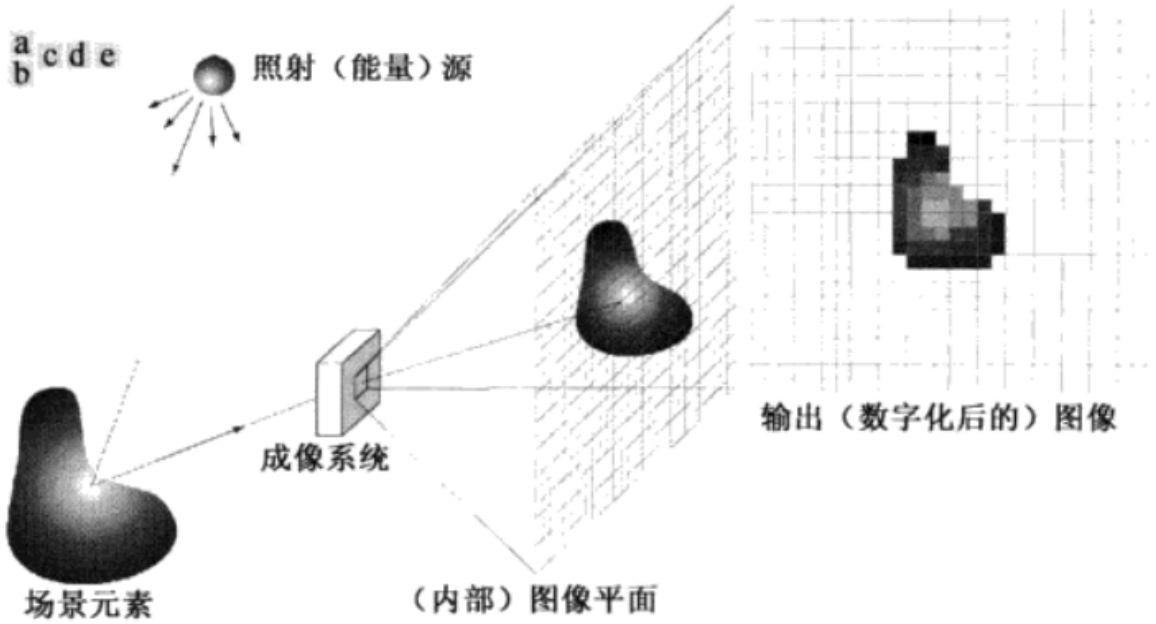


图 2.15　数字图像获取过程的一个例子：(a)能量(“照射”)源；(b)场景元素；(c)成像系统；(d)场景到图像平面的投影；(e)数字化后的图像

函数$f(x,y)$可以由两个分量来表征：

- **入射分量**：入射到被观察场景的光源照射总量，用$i(x,y)$表示；
- **反射分量**：场景中物体所反射的光照总量，用$r(x,y)$表示。

这两个分量的乘积合并形成$f(x,y)$，

$$f(x,y)=i(x,y)r(x,y) \tag{2}$$

其中，

$$0<i(x,y)<\infty \tag{3}$$

$$0<r(x,y)<1 \tag{4}$$

式(4)指出反射分量显示在0（全吸收）和1（全反射）之间。$i(x,y)$的性质取决于反射源，而$r(x,y)$的性质则取决于成像物体的特性。这种表示还可以用于照射光通过一个媒体形成图像的情况。

1.2 同态滤波

上面谈到的图像形成模型可用于开发一种频率域处理过程，该过程通过同时压缩灰度范围和增强对比度来改善一幅图像的表观。一幅图像$f(x,y)$可以表示为其照射分量和反射分量的乘积，即

$$f(x,y)=i(x,y)r(x,y) \tag{5}$$

上式不能直接用于对照射和反射分量的操作，因为乘积的傅立叶变换不是傅立叶变换的乘积。

$$F[f(x,y)]\neq F[i(x,y)]F[r(x,y)] \tag{6}$$

由数学可以，乘积的对数等于对数的和，而和的傅立叶积分等于傅立叶积分的和，因此，可以定义：

$$z(x,y)=ln(f(x,y))=ln(i(x,y))+ln(r(x,y)) \tag{7}$$

那么，

$$F[z(x,y)] = F[ln(f(x,y))] = F[ln(i(x,y))] + F[ln(r(x,y))] \tag{8}$$

或

$$Z(u,v) = F_i(u,v) + F_r(u,v) \tag{9}$$

其中，$F_i(u,v)$和$F_r(u,v)$分别表示$ln(i(x,y))$和$ln(r(x,y))$的傅立叶变换。 用滤波器$H(u,v)$对$Z(u,v)$进行滤波，有：

$$S(u,v) = H(u,v)Z(u,v) = H(u,v)F_i(u,v) + H(u,v)F_r(u,v) \tag{10}$$

在空间域中，滤波后的图像是：

$$s(x,y) = F^{-1}\{S(u,v)\} = F^{-1}\{H(u,v)F_i(u,v)\} + F^{-1}\{H(u,v)F_r(u,v)\} \tag{11}$$

由定义

$$i^{'}(x,y) = F^{-1}\{H(u,v)F_i(u,v)\} \tag{12}$$

$$r^{'}(x,y) = F^{-1}\{H(u,v)F_r(u,v)\} \tag{13}$$

所以：

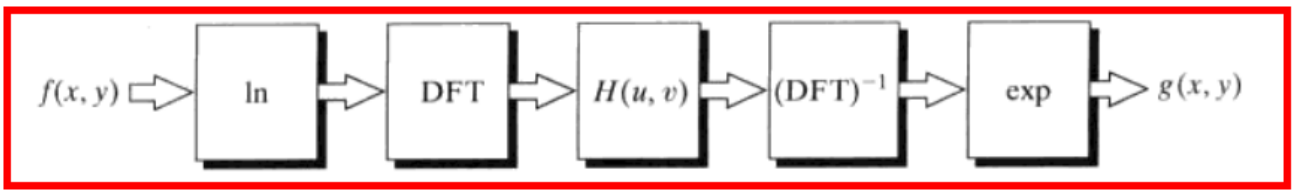
$$s(x,y) = i^{'}(x,y) + r^{'}(x,y) \tag{14}$$

因为$z(x,y)$是通过输入图像的自然对数形成的，因此可以通过对滤波后的图像的指数这一反处理过程来形成输出图像。

$$g(x,y) = e^{s(x,y)} = e^{i^{'}(x,y)}e^{r^{'}(x,y)} = i_0(x,y)r_0(x,y) \tag{15}$$

其中， $i_0(x,y) = e^{i^{'}(x,y)}$ (16) $r_0(x,y) = e^{r^{'}(x,y)}$ (17) 是输出图像的照射和反射分量。

该方法是以称之为同态系统的一类系统的特殊情况为基础的，方法的关键是照射分量和反射分量的分离，然后，同态滤波函数$H(u,v)$可以对各分量进行操作。



由上图可以看出同态滤波的过程如下：

- (1) 对输入图像进行对数变换，原理上对应将照射分量和反射分量分离；
- (2) 进行傅立叶变换；
- (3) 在频域对图像进行滤波处理；
- (4) 对滤波处理的结果进行反傅立叶变换；
- (5) 对反傅立叶变换的结果进行指数变换，得到输出结果图像。

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，特别是在不同物体连接不部分。这些特征导致图像取对数之后的傅立叶变换的低频成分与照射相关联，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。这种控制需要指定一个滤波器函数$H(u,v)$,它可以对傅立叶变换的低频和高频部分用不同的方法处理。

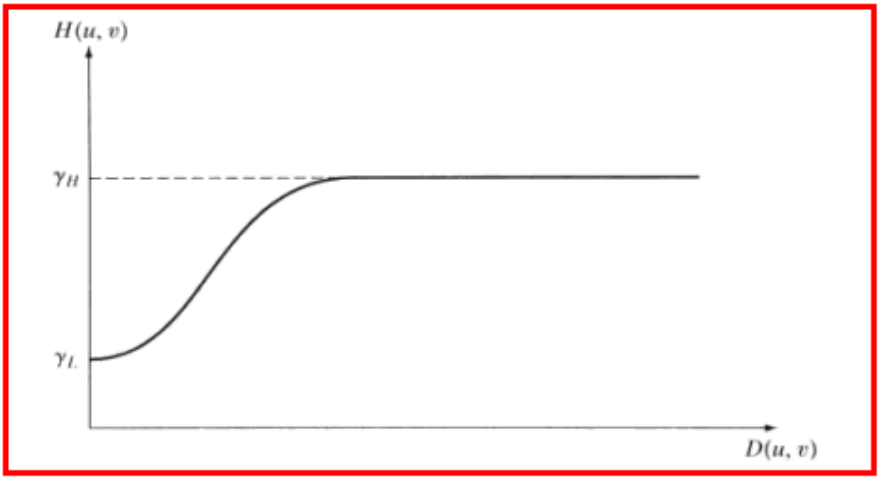


图 4.61　圆形对称同态滤波器函数的径向剖面图。垂直轴位于频率矩形的中心，$D(u,v)$是距中心的距离

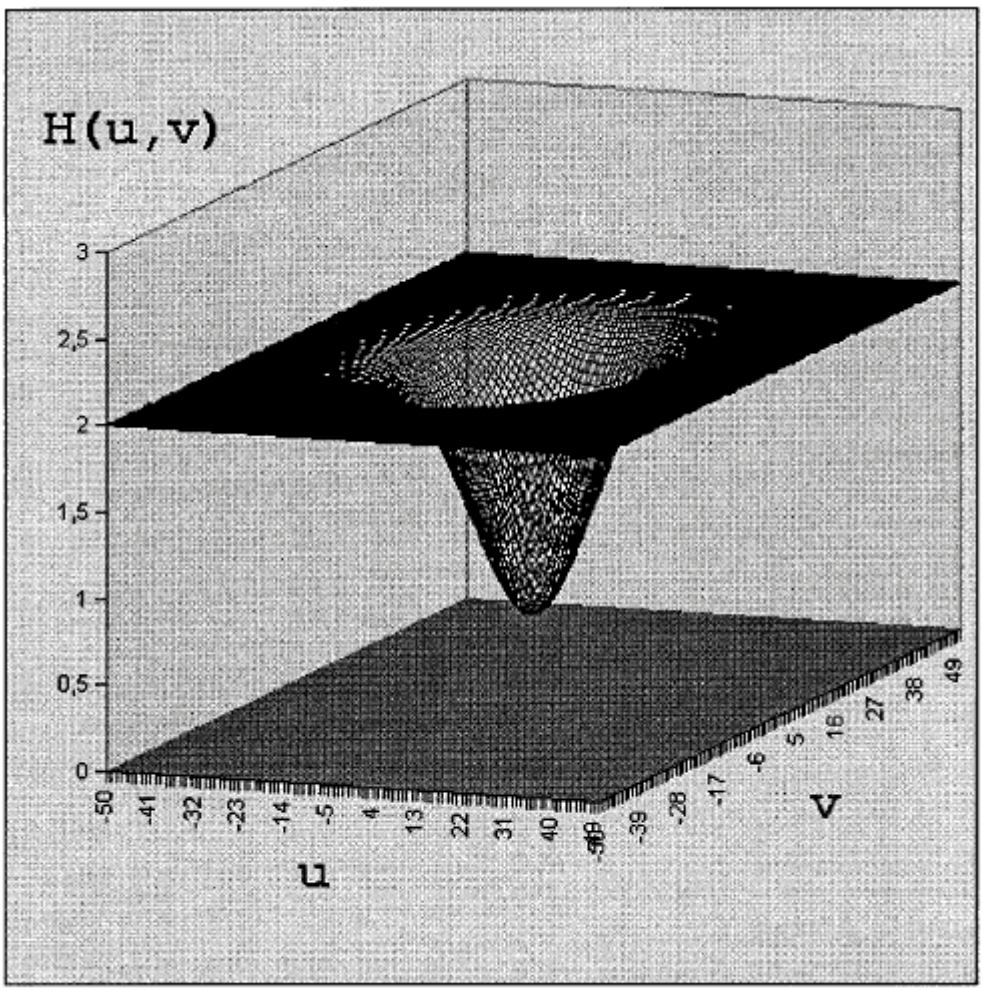


Fig. 3. Three-dimensional plot of a typical frequency-domain high-pass filter function.

上图显示的是一个可用的滤波器剖面图。如果γ_H和γ_L选定，而$\gamma_H > 1$且$\gamma_L < 1$，那么该滤波器函数取象于衰减低频（照射）分量，而增强高频（反射）分量。最终结果是同事进行动态范围的压缩和对比度的增强。

$$H(u,v) = (\gamma_H - \gamma_L)[1 - e^{-c[D^2(u,v)/D_0^2]}] + \gamma_L \tag{18}$$

2 实现

2.1 Matlab实现

```
function [image_out] = myHF(image_in, rh, rl, c, D0)
% 同态滤波器
% 输入为需要进行滤波的灰度图像，同态滤波器的参数rh, rl,c, D0
% 输出为进行滤波之后的灰度图像

[m, n] = size(image_in);
P = 2*m;
Q = 2*n;
% 取对数
image1 = log(double(image_in) + 1);
fp = zeros(P, Q);
%对图像填充0,并且乘以(-1)^(x+y) 以移到变换中心
for i = 1 : m
    for j = 1 : n
        fp(i, j) = double(image1(i, j)) * (-1)^(i+j);
    end
end
% 对填充后的图像进行傅里叶变换
F1 = fft2(fp);
% 生成同态滤波函数，中心在(m+1,n+1)
Homo = zeros(P, Q);
a = D0^2;
% 计算一些不变的中间参数
r = rh-rl;
for u = 1 : P
    for v = 1 : Q
        temp = (u-(m+1.0))^2 + (v-(n+1.0))^2;
        Homo(u, v) = r * (1-exp((-c)*(temp/a))) + rl;
    end
end
%进行滤波
G = F1 .* Homo;
% 反傅里叶变换
gp = ifft2(G);
% 处理得到的图像
image_out = zeros(m, n, 'uint8');
gp = real(gp);
g = zeros(m, n);
for i = 1 : m
    for j = 1 : n
        g(i, j) = gp(i, j) * (-1)^(i+j);
    end
end
% 指数处理
ge = exp(g)-1;
% 归一化到[0, L-1]
mmax = max(ge(:));
```

```
mmin = min(ge(:));
range = mmax-mmin;
for i = 1 : m
    for j = 1 : n
        image_out(i,j) = uint8(255 * (ge(i, j)-mmin) / range);
    end
end
end
```

调用方法:

```
clear all;
close all;
clc;
image1 = imread('../images/61.jpg');
image1 = rgb2gray(image1);
[m, n] = size(image1);
image2 = myHF(image1, 0.6, 0.55, 1, 80);
% 显示图像
subplot(1,2,1), imshow(image1), title('原图像');
subplot(1,2,2), imshow(image2), title('D0 = 80');
```

2.2 OpenCV实现

```
#include "../include/importOpenCV.h"
#include "../include/baseOps.h"
#include "../include/opencv400/opencv2/core.hpp"
#include <iostream>



/*
//High-Frequency-Emphasis Filters
H(u,v)=(\gamma_{H}-\gamma_{L})[1-e^{-c[D^{2}(u,v)/D_{0}^2]}]+\gamma_{L}
根据输入的滤波器尺寸，半径D0，调节系数c，上界rh,下界rl，计算H(u,v)表示的高通滤波器，
返回复数形式的滤波器，通过realIm返回该滤波器的实部。
注意，返回的复数滤波器的虚部为0。
*/
cv::Mat Butterworth_Homomorphic_Filter(cv::Size sz, int D, int c, float rh, float rl,
cv::Mat& realIm)
{
    cv::Mat single(sz.height, sz.width, CV_32F);
    cv::Point centre = cv::Point(sz.height/2, sz.width/2);
    double radius;
    float upper = rh ;
    float lower = rl ;
    long dpow = D*D;
    float W = (upper - lower);
    for(int i = 0; i < sz.height; i++)
```

```
    {
        for(int j = 0; j < sz.width; j++)
        {
            radius = pow((float)(i - centre.x), 2) + pow((float) (j - centre.y), 2);
            float r = exp(-c*radius/dpow);
            if(radius < 0)
                single.at<float>(i,j) = upper;
            else
                single.at<float>(i,j) =W*(1 - r) + lower;
        }
    }
    single.copyTo(realIm);

    //将实数滤波器扩展成虚部为0的复数滤波器
    cv::Mat butterworth_complex;
    //make two channels to match complex
    cv::Mat butterworth_channels[] = { cv::Mat_<float>(single), cv::Mat::zeros(sz,
CV_32F)};
    cv::merge(butterworth_channels, 2, butterworth_complex);
    return butterworth_complex;
}

/*
//DFT 返回功率谱Power
计算输入图像的傅立叶变换,
返回该图像的傅立叶变换的功率谱;
通过image_comples返回该图像的复数形式的傅立叶变换值;
通过image_phase返回该图像傅立叶变换对应的相位;
通过image_mag返回该图像傅立叶变换的幅度谱

注意：这里计算的不是标准的傅立叶变换，因为为了进行同态滤波，对输入图像进行了src = log(1+src)的操作。
*/
cv::Mat FFT2(cv::Mat frame_bw, cv::Mat& image_complex, cv::Mat &image_phase, cv::Mat
&image_mag)
{
    cv::Mat frame_log;
    //将8UC1转换成32FC1;
    frame_bw.convertTo(frame_log, CV_32F);
    frame_log = frame_log/255;//归一化
    //对输入图像进行自然对数变换，ln(src+1)
    frame_log += 1;
    cv::log( frame_log, frame_log);

    /*2.Expand the image to an optimal size The performance of the DFT depends of the
image size.
    It tends to be the fastest for image sizes that are multiple of 2, 3 or 5.
    We can use the copyMakeBorder() function to expand the borders of an image.*/
    cv::Mat padded;
    int M = cv::getOptimalDFTSize(frame_log.rows);
    int N = cv::getOptimalDFTSize(frame_log.cols);
    cv::copyMakeBorder(frame_log, padded, 0, M - frame_log.rows, 0, N - frame_log.cols,
cv::BORDER_CONSTANT, cv::Scalar::all(0));
```

```
    /*Make place for both the complex and real values The result of the DFT is a
complex.
    Then the result is 2 images (Imaginary + Real), and the frequency domains range is
much
    larger than the spatial one. Therefore we need to store in float ! That's why we
will
    convert our input image "padded" to float and expand it to another channel to hold
    the complex values. Planes is an arrow of 2 matrix (planes[0] = Real part,
planes[1] = Imaginary part)*/
    cv::Mat image_planes[] = { cv::Mat_<float>(padded), cv::Mat::zeros(padded.size(),
CV_32F)};
    /*Creates one multichannel array out of several single-channel ones.*/
    cv::merge(image_planes, 2, image_complex);
    /*Make the DFT The result of thee DFT is a complex image : "image_complex"*/
    cv::dft(image_complex, image_complex);
    /***************************/
    //Create spectrum magnitude//
    /***************************/
    /*Transform the real and complex values to magnitude NB: We separe Real part to
Imaginary part*/
    cv::split(image_complex, image_planes);
    //Starting with this part we have the real part of the image in planes[0] and the
imaginary in planes[1]
    cv::phase(image_planes[0], image_planes[1], image_phase);
    cv::magnitude(image_planes[0], image_planes[1], image_mag);
    //Power
    cv::pow(image_planes[0],2,image_planes[0]);
    cv::pow(image_planes[1],2,image_planes[1]);
    cv::Mat Power = image_planes[0] + image_planes[1];
    return Power;
}

/*
傅立叶逆变换

注意，这里为了进行同态滤波，对逆变换的结果进行了e为底的指数变换之后，
对结果进行了-1之后归一化的操作，对应在傅立叶正变换的过程中的ln(src+1)操作。
*/
void IFFT2(cv::Mat input, cv::Mat& inverseTransform)
{
    /*Make the IDFT*/
    cv::Mat result;
    cv::idft(input, result, cv::DFT_SCALE);
    /*Take the exponential*/
    cv::exp(result, result);
    std::vector<cv::Mat> planes;
    cv::split(result, planes);
    cv::magnitude(planes[0], planes[1], planes[0]);
    planes[0] = planes[0] - 1.0;
    cv::normalize(planes[0], planes[0], 0, 255, cv::NORM_MINMAX);
    planes[0].convertTo(inverseTransform, CV_8U);
}
```

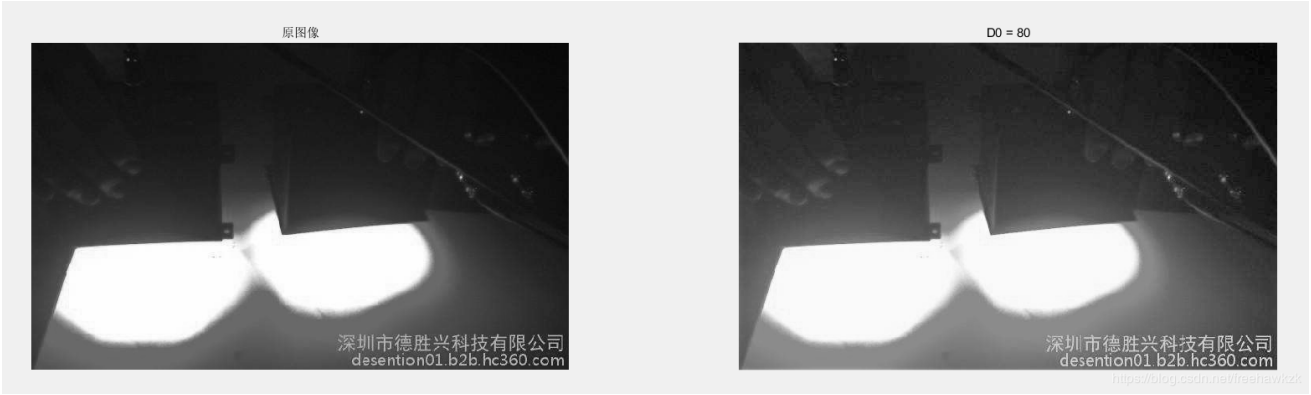
```
//傅立叶变换时交换四个象限
void ShiftFFT2(cv::Mat &fImage)
{
    //For visualization purposes we may also rearrange the quadrants of the result, so
that the origin (0,0), corresponds to the image center.
    cv::Mat tmp, q0, q1, q2, q3;
    /*First crop the image, if it has an odd number of rows or columns.
    Operator & bit to bit by -2 (two's complement : -2 = 111111111....10)
    to eliminate the first bit 2^0 (In case of odd number on row or col,
    we take the even number in below)*/
    fImage = fImage(cv::Rect(0, 0, fImage.cols & -2, fImage.rows & -2));
    int cx = fImage.cols/2;
    int cy = fImage.rows/2;
    /*Rearrange the quadrants of Fourier image so that the origin is at the image
center*/
    q0 = fImage(cv::Rect(0, 0, cx, cy));
    q1 = fImage(cv::Rect(cx, 0, cx, cy));
    q2 = fImage(cv::Rect(0, cy, cx, cy));
    q3 = fImage(cv::Rect(cx, cy, cx, cy));
    /*We reverse each quadrant of the frame with
    its other quadrant diagonally opposite*/
    /*We reverse q0 and q3*/
    q0.copyTo(tmp);
    q3.copyTo(q0);
    tmp.copyTo(q3);
    /*We reverse q1 and q2*/
    q1.copyTo(tmp);
    q2.copyTo(q1);
    tmp.copyTo(q2);
}

int main()
{
    //将工作目录设置到EXE所在的目录。
    SetCurrentDirectoryToExePath();

    cv::Mat src = cv::imread("../images/61.jpg");
    cv::cvtColor(src, src, cv::COLOR_BGR2GRAY);
    cv::imshow("原图", src);


    cv::Mat img;
    cv::Mat imgHls;
    std::vector<cv::Mat> vHls;
    if (src.channels() == 3)
    {
        cvtColor(src, imgHls, cv::COLOR_BGR2HSV);
        split(imgHls, vHls);
        vHls[2].copyTo(img);
    }
    else
        src.copyTo(img);
```

```
    cv::Mat img_complex,img_mag,img_phase;
    cv::Mat fpower = FFT2(img,img_complex,img_phase,img_mag);
    ShiftFFT2(img_complex);
    ShiftFFT2(fpower);
    int D0 = 80;
    int n = 1;
    int w = img_complex.cols;
    int h = img_complex.rows;

    float rH = 0.6;
    float rL =0.55;
    cv::Mat filter,filter_complex;
    filter_complex = Butterworth_Homomorphic_Filter(cv::Size(w,h),D0,n,rH,rL,filter);
    cv::mulSpectrums(img_complex, filter_complex, filter_complex, 0);
    ShiftFFT2(filter_complex);

    cv::Mat result;
    IFFT2(filter_complex,result);
    cv::Mat dst;
    if(src.channels() == 3)
    {
        vHls.at(2)= result(cv::Rect(0,0,src.cols,src.rows));
        merge(vHls,imgHls);
        cvtColor(imgHls, dst, cv::COLOR_HSV2BGR);
    }
    else
        result.copyTo(dst);

    cv::imshow("dst", dst);


    cv::waitKey();
    cv::destroyAllWindows();
    return 0;
}
```

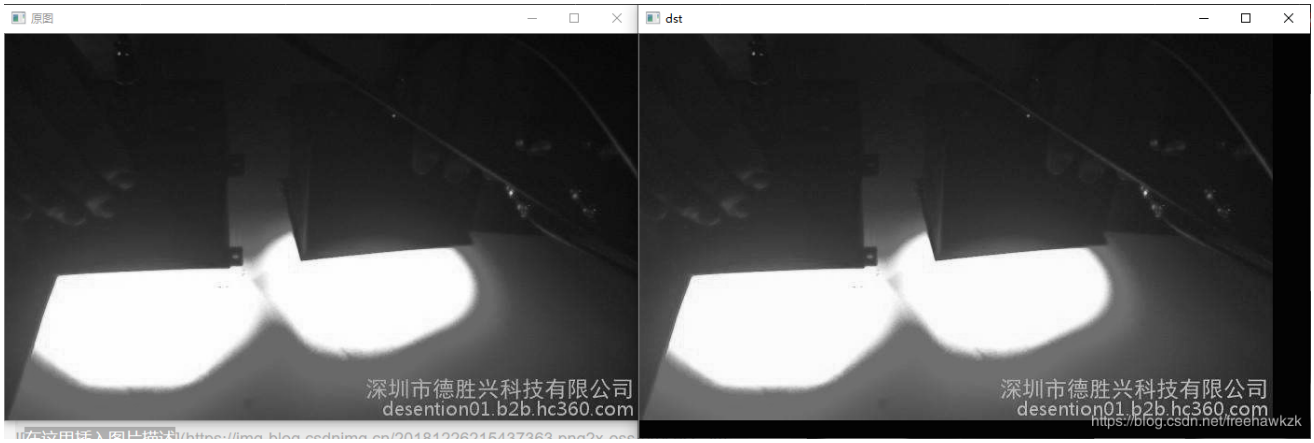
3 效果

为了测试算法的效果，我在网上找了些图片进行测试。测试图片的版权归原作者所有，本人不主张对测试图片及结果的任何版权。图片来源于百度图片搜索。

3.1 Matlab



3.2 OpenCV



可以看出，处理结果的尺寸相比原图发生了变化，因为进行DFT时对图像进行了填充，我暂时没有注意去将结果的边缘取消。