# 项目二

## 水下图像复原

### 复原简介：

该复原方法使用了自动红色通道水下图像复原的方法，实际上也是基于去雾算法做的，下面是这个方法的具体步骤：

退化模型：

红色通道先验：

水下光估计：

类似大气光估计的方法，在对水下光进行估计时，获取红色通道图像中前10%最亮的像素，并在这组像素中选择退化图像中最亮的像素。

透射率估计：

其中，，，分别为水光三个通道的分量，也就是的三个分量。，表示绿色和蓝色通道的相对衰减系数。

透射率图精细化：

使用引导滤波（Guided Filter）对粗略的透射率图进行精细化处理，以捕捉退化图像的细节，避免恢复场景时产生伪影。

最终反演：

其中，是自己设置的透射率下限值。

色彩校正：

其中，

人工照明处理：

当水下图像是在有人工照明的情况下进行处理的时候，可以使用这个公式：

### 关键代码：

水体光估计：

% 选择红色通道中最亮的10%像素

R = I(:,:,1);

num\_pixels = numel(R);

num\_bright\_pixels = round(0.4 \* num\_pixels);

[sorted\_R, ~] = sort(R(:), 'descend');

threshold = sorted\_R(num\_bright\_pixels);

bright\_pixels = R >= threshold;

% 从中选择红色分量最低的像素

A = zeros(1, 3);

for c = 1:3

channel = I(:,:,c);

A(c) = mean(channel(bright\_pixels));

end

透射率计算：

% 计算透射率

R = I(:,:,1);

G = I(:,:,2);

B = I(:,:,3);

t = 1 - min(min((1 - R) / (1 - A(1)), G / A(2)), B / A(3));

引导滤波：

[hei, wid] = size(p);

N = boxfilter(ones(hei, wid), r);

mean\_I\_r = boxfilter(I(:, :, 1), r) ./ N;

mean\_I\_g = boxfilter(I(:, :, 2), r) ./ N;

mean\_I\_b = boxfilter(I(:, :, 3), r) ./ N;

mean\_p = boxfilter(p, r) ./ N;

mean\_Ip\_r = boxfilter(I(:, :, 1).\*p, r) ./ N;

mean\_Ip\_g = boxfilter(I(:, :, 2).\*p, r) ./ N;

mean\_Ip\_b = boxfilter(I(:, :, 3).\*p, r) ./ N;

cov\_Ip\_r = mean\_Ip\_r - mean\_I\_r .\* mean\_p;

cov\_Ip\_g = mean\_Ip\_g - mean\_I\_g .\* mean\_p;

cov\_Ip\_b = mean\_Ip\_b - mean\_I\_b .\* mean\_p; var\_I\_rr = boxfilter(I(:, :, 1).\*I(:, :, 1), r) ./ N - mean\_I\_r .\* mean\_I\_r;

var\_I\_rg = boxfilter(I(:, :, 1).\*I(:, :, 2), r) ./ N - mean\_I\_r .\* mean\_I\_g;

var\_I\_rb = boxfilter(I(:, :, 1).\*I(:, :, 3), r) ./ N - mean\_I\_r .\* mean\_I\_b;

var\_I\_gg = boxfilter(I(:, :, 2).\*I(:, :, 2), r) ./ N - mean\_I\_g .\* mean\_I\_g;

var\_I\_gb = boxfilter(I(:, :, 2).\*I(:, :, 3), r) ./ N - mean\_I\_g .\* mean\_I\_b;

var\_I\_bb = boxfilter(I(:, :, 3).\*I(:, :, 3), r) ./ N - mean\_I\_b .\* mean\_I\_b;

a = zeros(hei, wid, 3);

for y=1:hei

for x=1:wid

Sigma = [var\_I\_rr(y, x), var\_I\_rg(y, x), var\_I\_rb(y, x);

var\_I\_rg(y, x), var\_I\_gg(y, x), var\_I\_gb(y, x);

var\_I\_rb(y, x), var\_I\_gb(y, x), var\_I\_bb(y, x)];

Sigma = Sigma + eps \* eye(3);

cov\_Ip = [cov\_Ip\_r(y, x), cov\_Ip\_g(y, x), cov\_Ip\_b(y, x)];

a(y, x, :) = cov\_Ip \* ((Sigma + eps) \ eye(3));

end

end

b = mean\_p - a(:, :, 1) .\* mean\_I\_r - a(:, :, 2) .\* mean\_I\_g - a(:, :, 3) .\* mean\_I\_b;

q = (boxfilter(a(:, :, 1), r).\* I(:, :, 1)...

+ boxfilter(a(:, :, 2), r).\* I(:, :, 2)...

+ boxfilter(a(:, :, 3), r).\* I(:, :, 3)...

+ boxfilter(b, r)) ./ N;

透射率精细化：

t\_R\_refined = guided\_filter(I, t\_R, 15, 1e-6);

t\_G\_refined = guided\_filter(I, t\_G, 15, 1e-6);

t\_B\_refined = guided\_filter(I, t\_B, 15, 1e-6);

色彩校正：

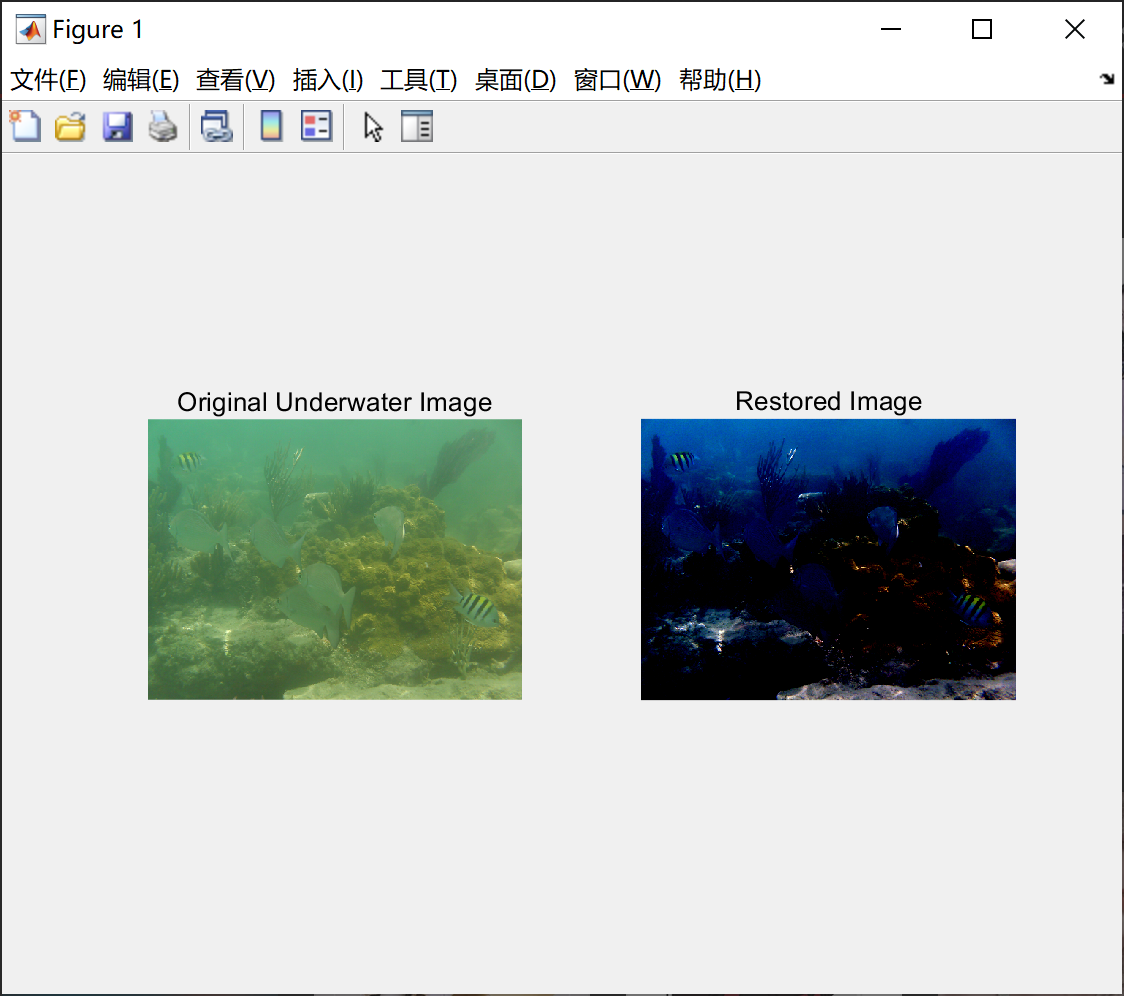
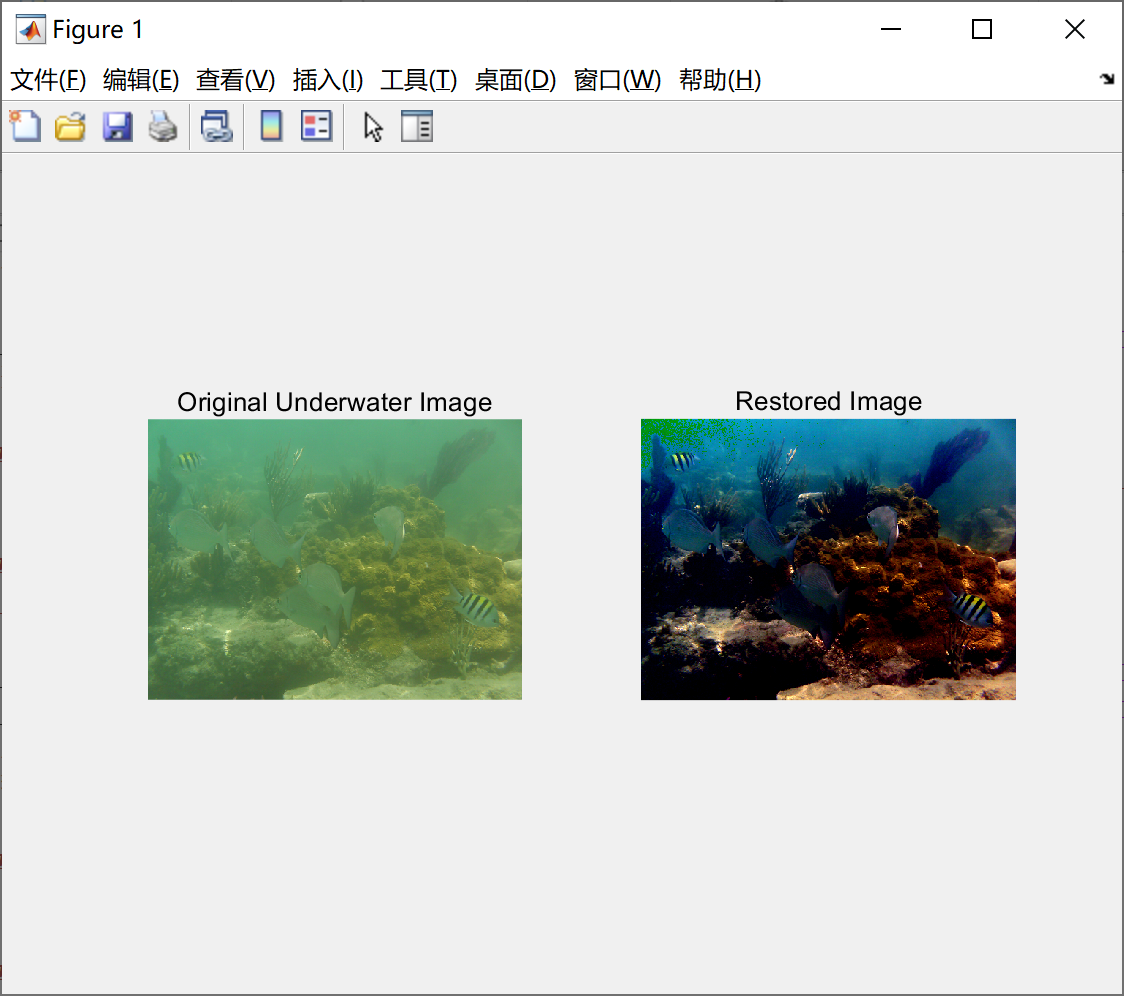
J\_R = (I(:,:,1) - A(1)) ./ max(t\_R\_refined, t0) + (1 - A(1)) \* A(1);

J\_G = (I(:,:,2) - A(2)) ./ max(t\_G\_refined, t0) + (1 - A(2)) \* A(2);

J\_B = (I(:,:,3) - A(3)) ./ max(t\_B\_refined, t0) + (1 - A(3)) \* A(3);

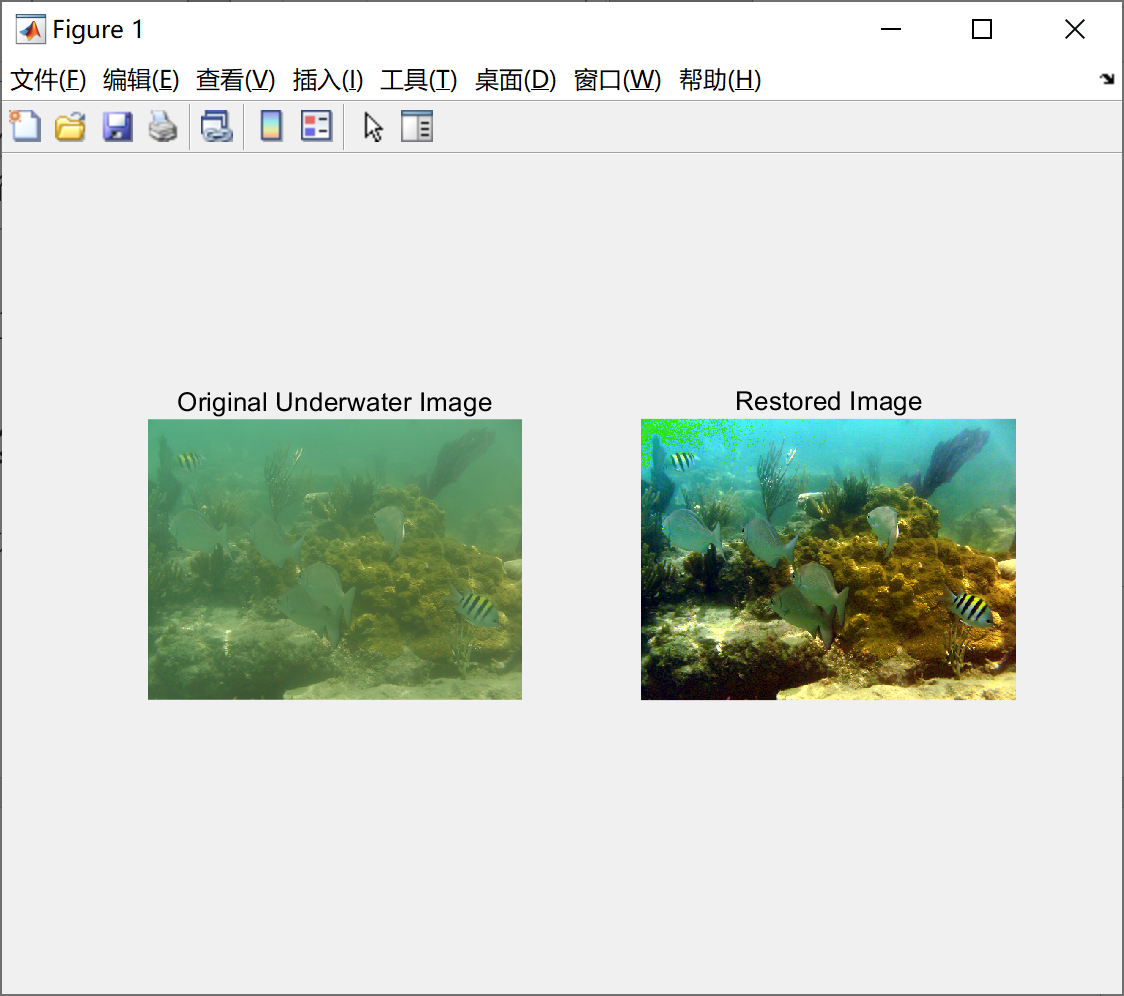
### 实验心得：

在实验的过程中，一开始遇到了图像过深的情况，后面经过调整发现，红色通道中最亮像素选取的百分比会影响图像颜色的深度，选取的百分比越小，就会有更多的像素被视为亮度较高的候选像素，表现出来的效果就是颜色过深。

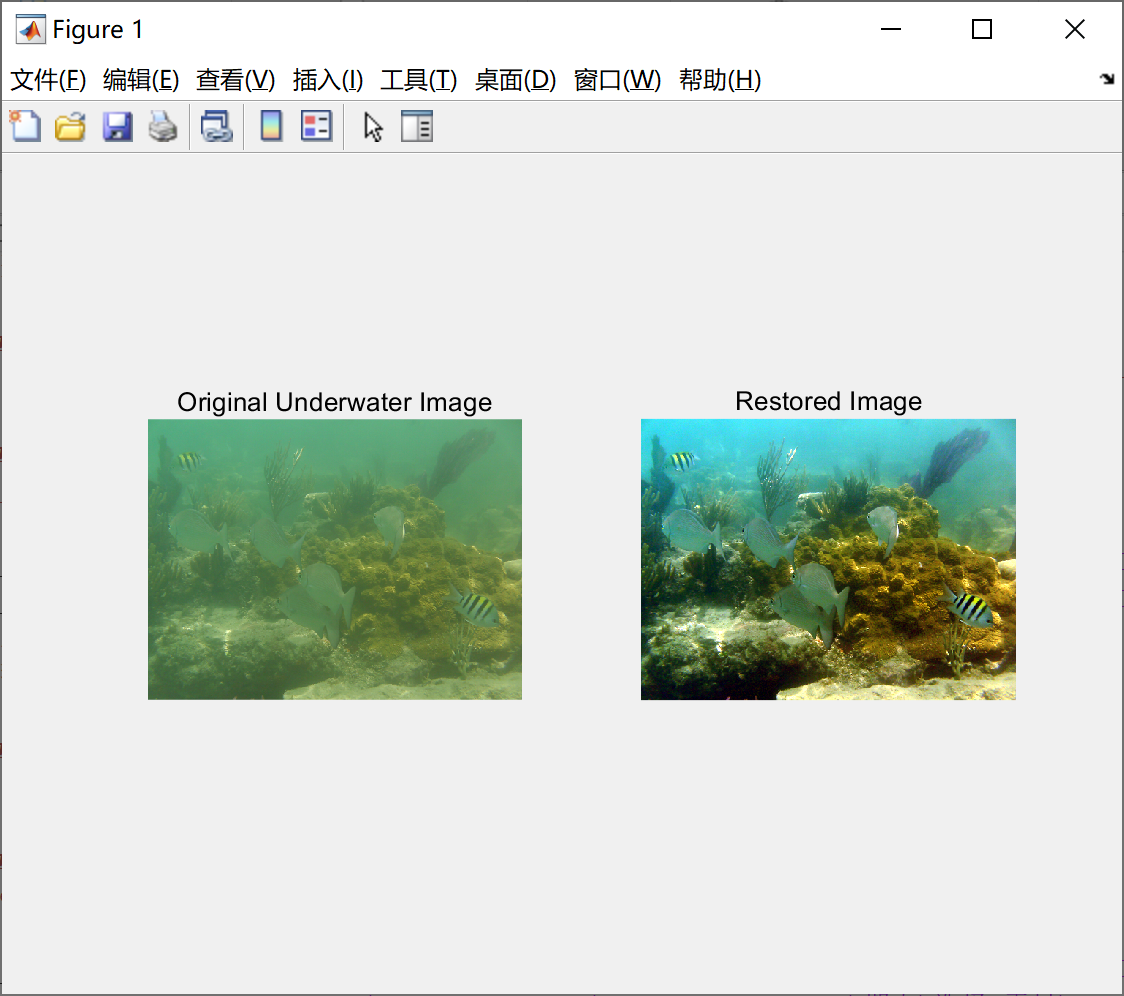
 

（左图是选取10%，右图是选取50%）

但是我仍然觉得饱和度较高，于是我加入了色彩平衡函数SimplestColorBalance，下图是进行色彩平衡后的结果：



可以看出要比之前的效果好上很多，但是左上角出现了绿色斑块，经过我的调整发现是最小透射率导致的结果，我将最小透射率调整为0.4之后，绿色斑块就消失了，下图是调整结果。



## 雾化图像复原

### 复原简介：

该复原方法使用了暗通道先验的去雾算法，基于“在无雾的自然图像中，大多数非天空区域的至少一个颜色通道（R、G、B）在某些局部补丁内的像素值非常低或者接近于零”的假设，暗通道先验算法认为雾天图像会因为雾的存在提高图像整体的亮度，进而提高暗通道的值，因此雾天图像的暗通道（即局部补丁内像素值的最小值）要显著高于无雾图像。

算法中使用到的公式有这些：

I(x) 是在位置**x**的雾天图像的像素值。

J(x)是在位置**x**的无雾图像的像素值。

t(x)是在位置**x**的透射率，表示物体表面到相机的光传输的比例。

A 是大气光，表示雾天中散射的背景光。

Ω(x) 表示以**x**为中心的局部补丁。

表示在颜色通道**c**上的像素值。

雾气退化模型如下：

求解暗通道：

全局大气光估计：

但提出这个方法的作者（何恺明）认为这样找到的最亮点可能是一些白色的物体而不是雾最密集的区域，因此提出从观测图像的暗通道中选取最亮的前0.1%的点，这些点对应到原图中再去找最亮的点作为的估计，三通道分别找到，，。

透射率估计：

k为图像中一个点，通过求得以点k为中心的局部窗口的传输率，从一个点求解到全图，进而得到全图的传输率。

得到去雾后的图像：

通过求每个通道的每个点得到最终的去雾图像。

### 关键代码：

暗通道计算：

% 输入: I - 输入图像

% patch\_size - 补丁大小

% 输出: dark\_channel - 暗通道图像

% 图像尺寸

[height, width, ~] = size(I);

% 初始化暗通道图像

dark\_channel = zeros(height, width);

% 半补丁大小

half\_patch = floor(patch\_size / 2);

% 遍历图像

for i = 1:height

for j = 1:width

% 计算局部补丁的边界

i\_min = max(1, i - half\_patch);

i\_max = min(height, i + half\_patch);

j\_min = max(1, j - half\_patch);

j\_max = min(width, j + half\_patch);

% 提取局部补丁

local\_patch = I(i\_min:i\_max, j\_min:j\_max, :);

% 计算局部补丁中的最小值

dark\_channel(i, j) = min(local\_patch(:));

end

end

大气光估计：

% 归一化大气光

norm\_I = bsxfun(@rdivide, I, reshape(A, 1, 1, 3));

% 计算暗通道

dark\_channel = get\_dark\_channel(norm\_I, patch\_size);

% 计算透射率

omega = 0.95; % 雾霾去除因子

t = 1 - omega \* dark\_channel;

透射率估计：

% 归一化大气光

norm\_I = bsxfun(@rdivide, I, reshape(A, 1, 1, 3));

% 计算暗通道

dark\_channel = get\_dark\_channel(norm\_I, patch\_size);

% 计算透射率

omega = 0.95; % 雾霾去除因子

t = 1 - omega \* dark\_channel;

这里的雾霾去除因子ω（omega）表示去雾程度，0表示不去雾

得到去雾图像：

J = zeros(size(I));

for c = 1:3

J(:, :, c) = (I(:, :, c) - A(c)) ./ t + A(c);

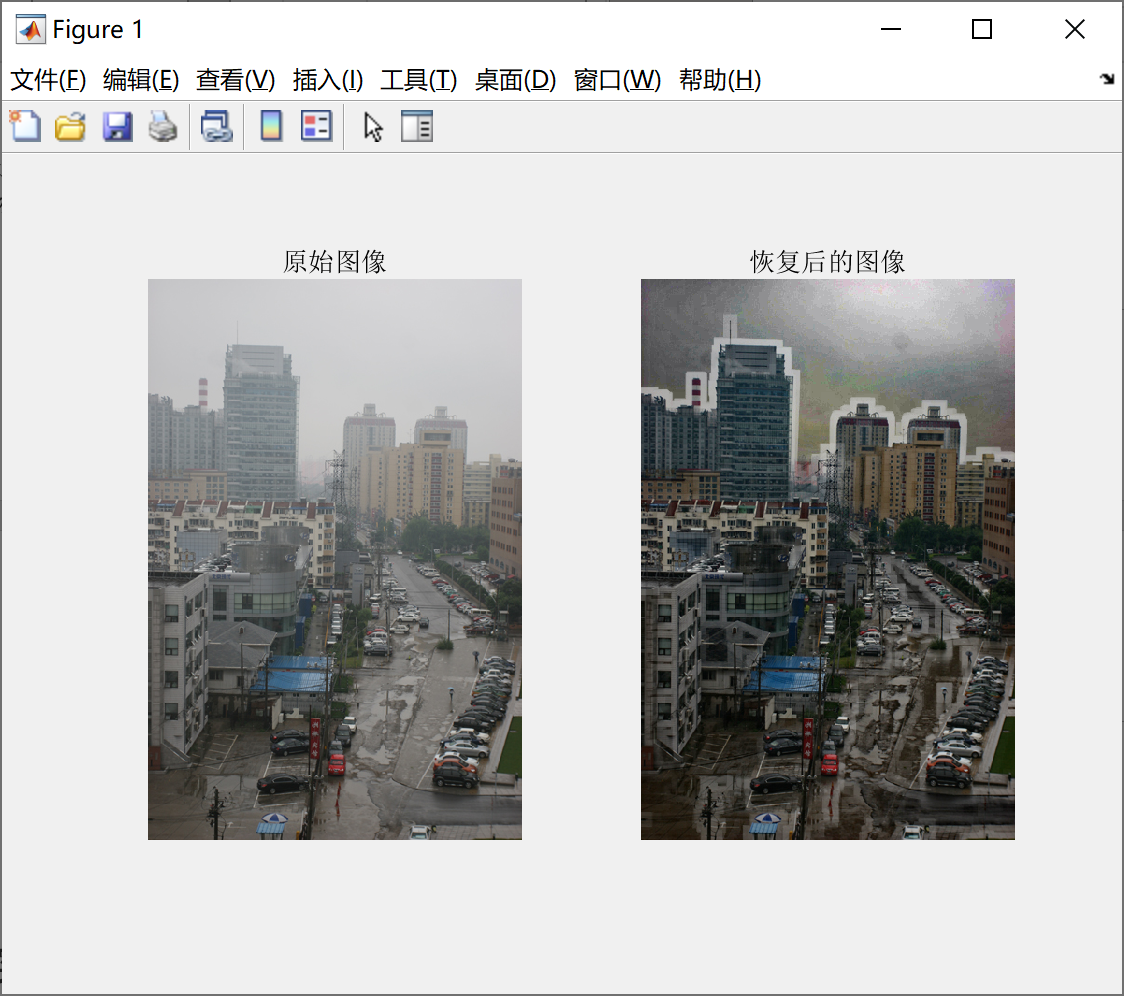
end

% 确保结果在 [0, 1] 范围内

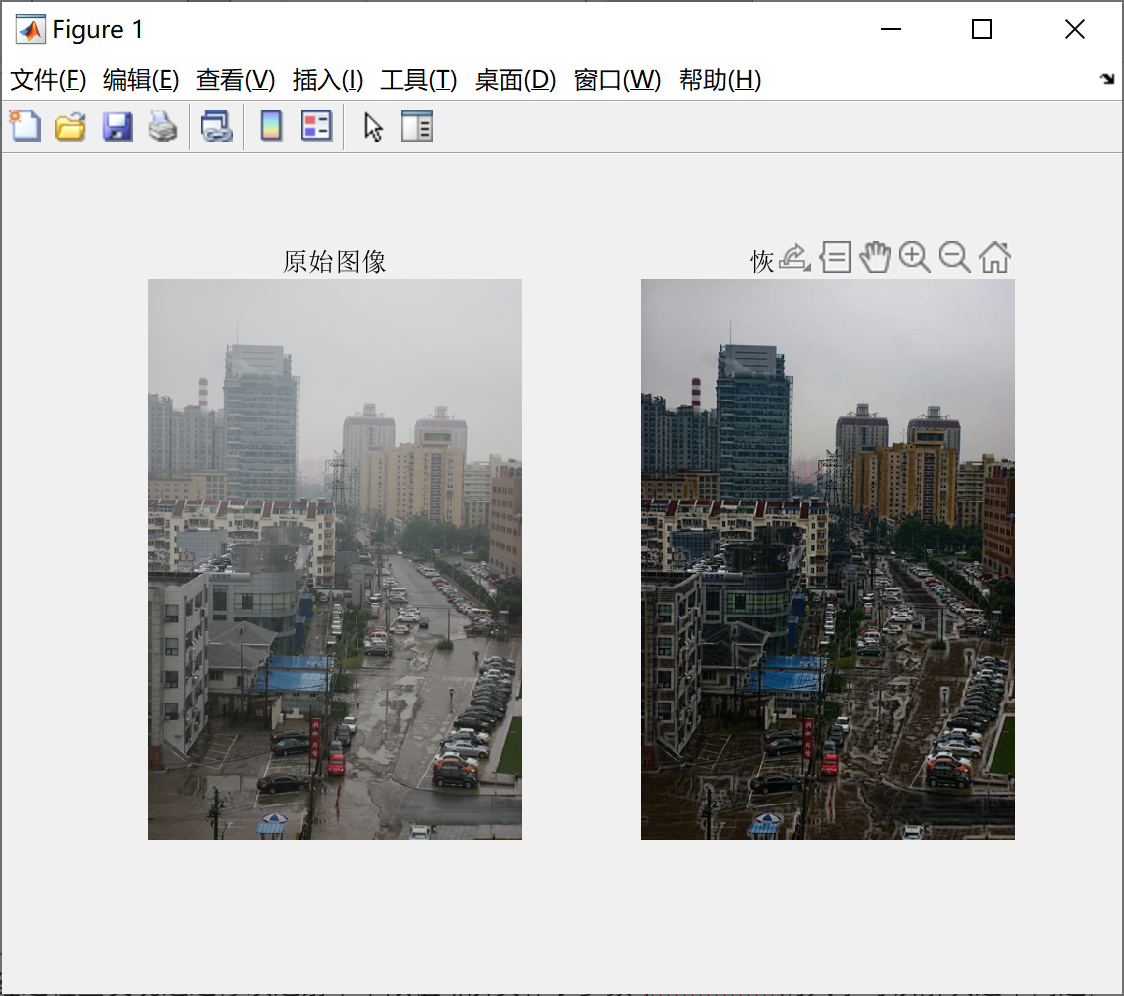
J = max(min(J, 1), 0);

### 实验心得：

在实验最开始得到的处理后的图像效果并不理想，在物体边缘会出现一些白块，如图：



经过检查发现通过修改透射率下限值以及补丁参数patch\_size的大小可以解决这个问题，下面是调整参数后的效果：



经过我的反复调整发现，参数patch\_size过小会导致暗通道的计算过于局部，产生噪声，图像细节过多，去雾效果不明显，还有可能产生伪影。过大则会导致图像的细节不够充分，看起来很模糊。参数过小会导致图像局部过亮，产生噪声。过大则会导致去雾效果减弱，图像的雾霾去除不完全。

## 运动模糊图像

### 复原简介：

鉴于之前的卷积法复原图像需要提前知道PSF（卷积核）的缺点，作者提出了盲区卷积的方法，利用图像的梯度分布和模糊核的形态两个先验信息，再加上构建的相机噪声，就得到了求解复原图像的基本模型。

重建公式：

其中，是输入的模糊图像，为模糊卷积核，为潜在图像（也就是需要得到的清晰图像），为每个像素的传感器噪声，为离散图像的卷积。

数学建模：

后验概率：

计算图像梯度，之后进行多尺度模糊核估计，通过变分贝叶斯来迭代优化模糊核，最后用估计出来的模糊核进行反卷积，复原图像。

### 关键代码：

% 模糊核估计函数

function K = inferKernel(Gx\_s, Gy\_s, K, num\_iterations)

% 模糊核估计的函数

% Gx\_s - 缩放后图像的x方向梯度

% Gy\_s - 缩放后图像的y方向梯度

% K - 初始模糊核

% num\_iterations - 迭代次数

learning\_rate = 0.001;

for iter = 1:num\_iterations

% 如果输入梯度包含颜色通道，分别处理每个通道

if size(Gx\_s, 3) > 1

B\_hat\_x = zeros(size(Gx\_s));

B\_hat\_y = zeros(size(Gy\_s));

for channel = 1:size(Gx\_s, 3)

B\_hat\_x(:, :, channel) = conv2(Gx\_s(:, :, channel), K, 'same');

B\_hat\_y(:, :, channel) = conv2(Gy\_s(:, :, channel), K, 'same');

end

else

B\_hat\_x = conv2(Gx\_s, K, 'same');

B\_hat\_y = conv2(Gy\_s, K, 'same');

end

% 计算误差

error\_x = Gx\_s - B\_hat\_x;

error\_y = Gy\_s - B\_hat\_y;

% 计算梯度

grad\_K\_x = zeros(size(K));

grad\_K\_y = zeros(size(K));

if size(Gx\_s, 3) > 1

for channel = 1:size(Gx\_s, 3)

grad\_K\_x = grad\_K\_x + conv2(rot90(error\_x(:, :, channel), 2), Gx\_s(:, :, channel), 'valid');

grad\_K\_y = grad\_K\_y + conv2(rot90(error\_y(:, :, channel), 2), Gy\_s(:, :, channel), 'valid');

end

else

grad\_K\_x = conv2(rot90(error\_x, 2), Gx\_s, 'valid');

grad\_K\_y = conv2(rot90(error\_y, 2), Gy\_s, 'valid');

end

% 更新模糊核

K = K + learning\_rate \* (grad\_K\_x + grad\_K\_y);

% 归一化模糊核

K = K / sum(K(:));

end

end

% 变分贝叶斯迭代优化

for iter = 1:num\_iterations\_vb

K\_sample = mu\_K + sigma\_K .\* randn(size(K));

Lx\_sample = mu\_Lx + sigma\_Lx .\* randn(size(Lx));

Ly\_sample = mu\_Ly + sigma\_Ly .\* randn(size(Ly));

% 计算梯度

[grad\_mu\_K, grad\_sigma\_K] = compute\_gradients\_K(K\_sample, B\_corrected, Lx\_sample, Ly\_sample);

[grad\_mu\_Lx, grad\_sigma\_Lx] = compute\_gradients\_L(Lx\_sample, B\_corrected, K\_sample);

[grad\_mu\_Ly, grad\_sigma\_Ly] = compute\_gradients\_L(Ly\_sample, B\_corrected, K\_sample);

% 添加总变差正则化梯度

grad\_tv\_Lx = grad\_total\_variation(Lx\_sample);

grad\_tv\_Ly = grad\_total\_variation(Ly\_sample);

grad\_mu\_Lx = grad\_mu\_Lx + lambda\_tv \* grad\_tv\_Lx;

grad\_mu\_Ly = grad\_mu\_Ly + lambda\_tv \* grad\_tv\_Ly;

% 梯度裁剪

grad\_mu\_K = max(min(grad\_mu\_K, clip\_threshold), -clip\_threshold);

grad\_sigma\_K = max(min(grad\_sigma\_K, clip\_threshold), -clip\_threshold);

grad\_mu\_Lx = max(min(grad\_mu\_Lx, clip\_threshold), -clip\_threshold);

grad\_sigma\_Lx = max(min(grad\_sigma\_Lx, clip\_threshold), -clip\_threshold);

grad\_mu\_Ly = max(min(grad\_mu\_Ly, clip\_threshold), -clip\_threshold);

grad\_sigma\_Ly = max(min(grad\_sigma\_Ly, clip\_threshold), -clip\_threshold);

% 更新变分分布的参数

mu\_K = mu\_K + learning\_rate \* grad\_mu\_K;

sigma\_K = sigma\_K + learning\_rate \* grad\_sigma\_K;

mu\_Lx = mu\_Lx + learning\_rate \* grad\_mu\_Lx;

sigma\_Lx = sigma\_Lx + learning\_rate \* grad\_sigma\_Lx;

mu\_Ly = mu\_Ly + learning\_rate \* grad\_mu\_Ly;

sigma\_Ly = sigma\_Ly + learning\_rate \* grad\_sigma\_Ly;

% 计算并输出总变差

tv\_Lx = total\_variation(mu\_Lx);

tv\_Ly = total\_variation(mu\_Ly);

fprintf('Iteration %d: TV\_Lx = %f, TV\_Ly = %f\n', iter, tv\_Lx, tv\_Ly);

end

### 实验心得：

在反卷积的过程中出了问题，mu\_K的值全为ｎａｎ，

for iter = 1:num\_iterations\_vb

% 采样

if iter == 6

K\_sample = mu\_K + sigma\_K .\* randn(size(K));

temp = K\_sample; % 在第一次迭代时记录 K\_sample 的值到 temp

else

K\_sample = mu\_K + sigma\_K .\* randn(size(K));

end

Lx\_sample = mu\_Lx + sigma\_Lx .\* randn(size(Lx));

Ly\_sample = mu\_Ly + sigma\_Ly .\* randn(size(Ly));

% 计算梯度

[grad\_mu\_K, grad\_sigma\_K] = compute\_gradients\_K(K\_sample, B\_corrected, Lx\_sample, Ly\_sample);

[grad\_mu\_Lx, grad\_sigma\_Lx] = compute\_gradients\_L(Lx\_sample, B\_corrected, K\_sample);

[grad\_mu\_Ly, grad\_sigma\_Ly] = compute\_gradients\_L(Ly\_sample, B\_corrected, K\_sample);

% 更新变分分布的参数

mu\_K = mu\_K + learning\_rate \* grad\_mu\_K;

sigma\_K = sigma\_K + learning\_rate \* grad\_sigma\_K;

mu\_Lx = mu\_Lx + learning\_rate \* grad\_mu\_Lx;

sigma\_Lx = sigma\_Lx + learning\_rate \* grad\_sigma\_Lx;

mu\_Ly = mu\_Ly + learning\_rate \* grad\_mu\_Ly;

sigma\_Ly = sigma\_Ly + learning\_rate \* grad\_sigma\_Ly;

end

在这个过程中发生了梯度爆炸，设置iter值为6就全为空值了，之后加上了梯度阈值就解决了这个问题。

