**信号与系统——MATLAB综合实验**

图像处理实验报告

学号：2019011008

无92 刘雪枫

2021年9月7日

目录

# 实验目的

# 实验平台

# 目录结构

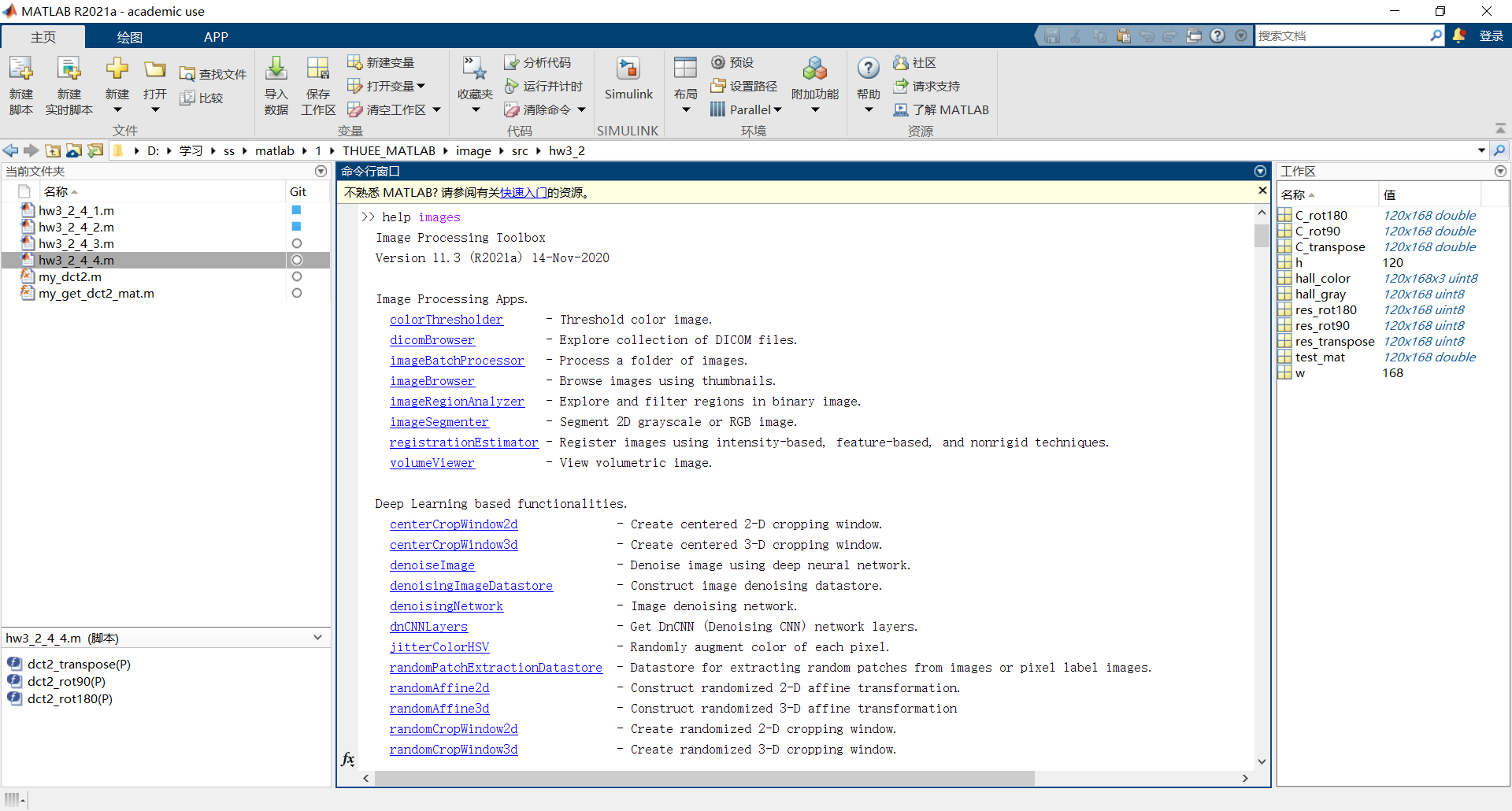
# 实验原理

# 实验内容

## 彩色图像

### MATLAB图像处理工具箱

在MATLAB中输入指令：help images，显示出很多图像处理函数：



接下来我们将利用这些图像处理函数进行实验

### 彩色图像绘制

要以图像的中心点为圆心画圆，需要将圆上的点的RGB值中的R值设为255，G和B均设为0。我们以长和宽的最小值的一半为半径，半径的0.95倍为内径画圆，关键代码为：

circle = hall\_color;

dist = (row\_idxs - (h+1)/2).^2 + (col\_idxs - (w+1)/2).^2;

radius = min(h/2, w/2)^2;

is\_in\_circle = dist <= radius & dist > 0.95 \* radius;

draw\_r\_circle = cat(3, is\_in\_circle, logical(zeros(size(is\_in\_circle))), logical(zeros(size(is\_in\_circle))));

draw\_gb\_circle = cat(3, logical(zeros(size(is\_in\_circle))), is\_in\_circle, is\_in\_circle);

circle(draw\_r\_circle) = uint8(255);

circle(draw\_gb\_circle) = uint8(0);

最后将其用imwrite函数绘制成位图，得到：



同样我们在图像上绘制棋盘。我们将棋盘的每一格的边长设置为10像素，把每个像素的横纵坐标分别除以10并取整得到的值对2的模作逻辑同或运算，结果为真则将RGB均设为0，则可以得到棋盘的效果：



本问题的完整代码位于文件hw3\_1\_3\_2.m中，得到的两个图片文件为该图片文件为hw3\_1\_3\_2\_circle.bmp和hw3\_1\_3\_2\_chess\_board.bmp。

## 图像压缩编码

### 变换域改变直流分量

预处理时，先将灰度值减去128。实际上，这个过程也可以变换域进行。由于离散余弦变换具有线性性，因此两个矩阵的离散余弦变换之和等于两个矩阵和的离散余弦变换。矩阵减去128就相当于原矩阵与元素全为128的矩阵相减，因此结果等于原矩阵的离散余弦变换减去全为128的矩阵的离散余弦变换。注意到全为128的矩阵只有直流分量，因此它的离散余弦变换只有左上角的元素非零，其余全是零。因此，只需要将原矩阵进行离散余弦变换，再把左上角元素减去一个数字即可。取hall\_gray的左上角的8\*8的部分进行验证，关键代码如下：

test\_mat = double(hall\_gray(1:8, 1:8));

C1 = dct2(test\_mat - 128);

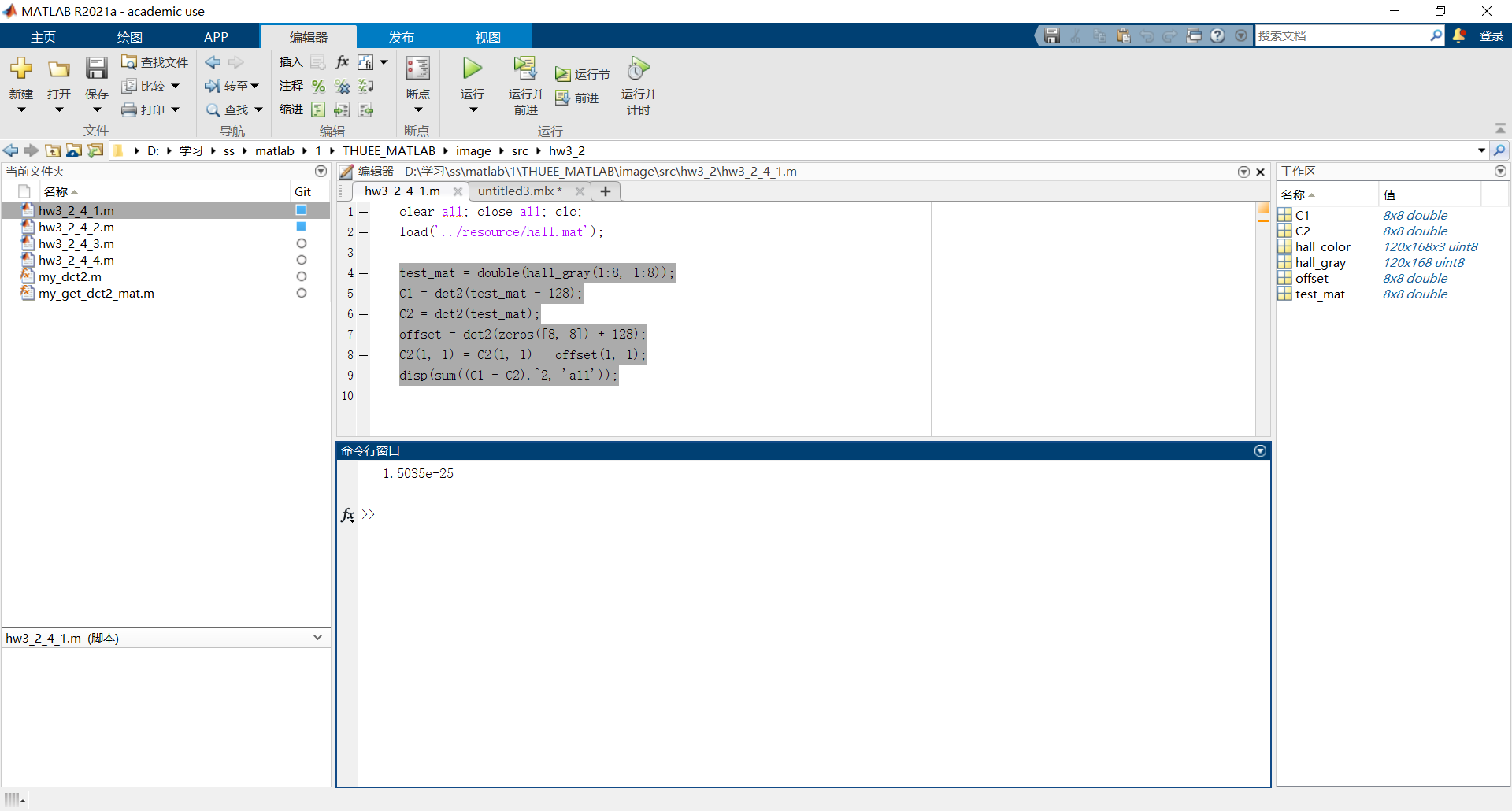
C2 = dct2(test\_mat);

offset = dct2(zeros([8, 8]) + 128);

C2(1, 1) = C2(1, 1) - offset(1, 1);

disp(sum((C1 - C2).^2, 'all'));

程序最后输出用两种方式计算的离散余弦变换之间的误差，误差采用各个元素误差的平方之和进行度量，程序输出结果如下：



可以看到，两种方法得到的离散余弦变换相差很小，几乎为零，这也验证了上述方法的正确性。本题完整代码位于文件hw3\_2\_4\_1.m中。

### 二维DCT的实现

要实现二维DCT，首先要计算DCT的变换矩阵。我将其封装为函数my\_get\_dct2\_mat，位于文件my\_get\_dct2\_mat.m。该函数接收一个整数N作为参数。返回大小为N的方阵D。该函数的关键代码如下：

D = (zeros([N - 1, N]) + [1 : 1 : N-1]') .\* [1 : 2 : 2\*N-1];

D = sqrt(2 / N) \* [zeros(1, N) + sqrt(1/2); cos(D \* pi / (2\*N))];

然后编写DCT变换函数，封装为函数my\_dct2，位于my\_dct2.m文件中。在实验过程中我注意到，虽然课件中介绍的DCT是均为方阵进行DCT，但是我注意到MATLAB提供的dct2函数的输入参数不必为方阵。因此，我对非方阵的DCT进行了猜想与推导。后面我利用我自己编写的函数与MATLAB提供的函数进行对照，可以证明我的写法是正确的。

按照离散余弦变换的定义，我编写的函数的关键代码如下：

function C = my\_dct2(P)

[h, w] = size(P);

C = my\_get\_dct2\_mat(h) \* double(P) \* my\_get\_dct2\_mat(w)';

end

下面对函数进行验证。仍然取hall\_gray的左上角8\*8的矩阵进行验证，验证代码如下：

test\_mat = double(hall\_gray(1:8, 1:8)) - 128;

res\_std = dct2(test\_mat);

res\_test = my\_dct2(test\_mat);

disp('res\_std: ');

disp(res\_std);

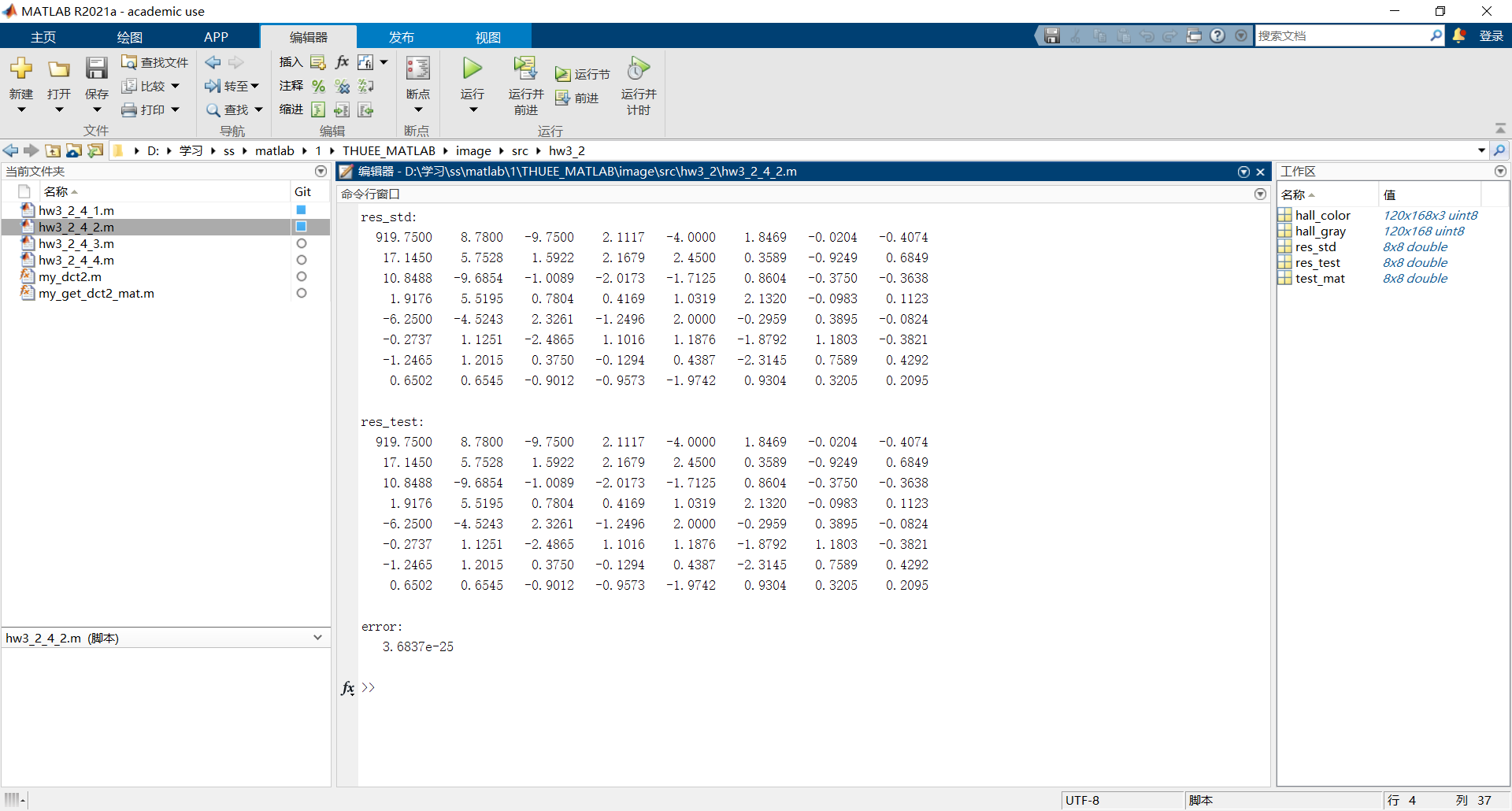
disp('res\_test: ');

disp(res\_test);

disp('error: ');

disp(sum((res\_std - res\_test).^2, 'all'));

分别打印出MATLAB给出的dct2函数的结果以及我自己编写的函数的结果，并计算误差：



可以看到两者的结果几乎完全一致，平方误差接近于零。这也验证了我自己编写的函数的正确性。本题完整代码位于文件hw3\_2\_4\_2.m中。

### 将系数矩阵部分置零

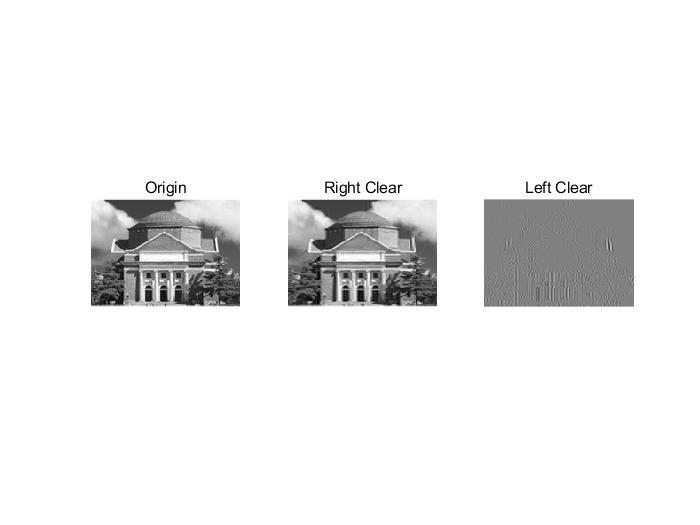
将灰度图每8个像素进行分块，每个块分别将DCT变换尔达右边四列和左边四列置零，再逆变换为图像。

由于离散余弦变换的系数矩阵中，左上角的元素代表直流和低频分量，左下角的元素代表纵向变化的高频分量，右上角的元素代表横向变化的高频分量，右下角的元素代表横向和纵向变化的高频分量。因此做出猜测：

将右侧置零，对图片整体影响不大，因为直流和低频的分量并没有受到过多的影响。但是由于横向的高频分量有所减弱，所以图片可能在横向上的色彩变化稍显模糊，但是不会过多地影响观感，因为图像的高频分量本来就比较小。

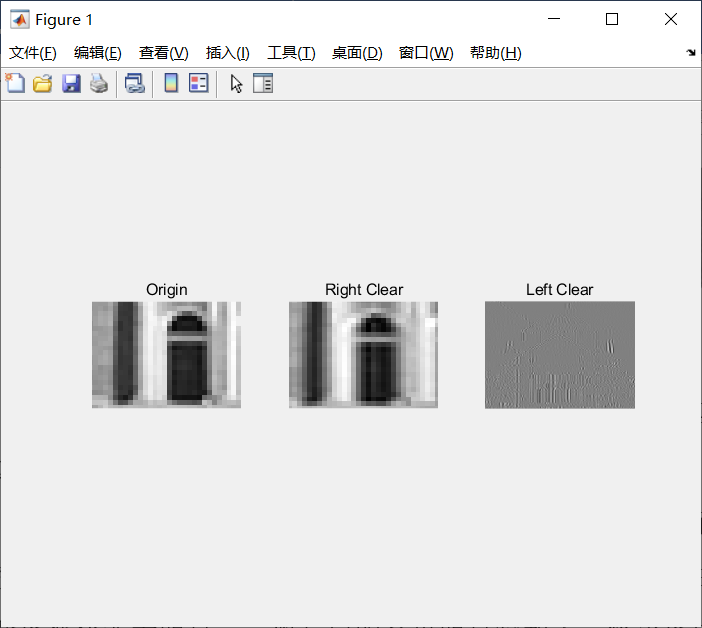
但是若将左侧四列元素置零，则会有很大的影响，因为低频分量被置为了零，所以图片原来的颜色平缓变化的部分消失了，而且图像的低频分量本来就比较大，且人眼对低频分量更敏感，因此图片质量可能严重受损。此外，由于左下角代表的是纵向变化的低频分量，因此图像纵向的灰度变化会降低；但是横向高频分量得到了很好的保留，因此图片可能会在横向上有明显的色彩变化，结果是造成图片上出现一条一条的纵向纹理。

整张图片处理结果如下：

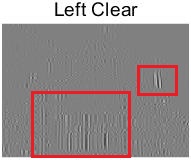


其中，左侧图为原图，中间的图为将每个小块的右侧四列置零得到的图，右边的图为将每个小块的左侧四列置零得到的图片。

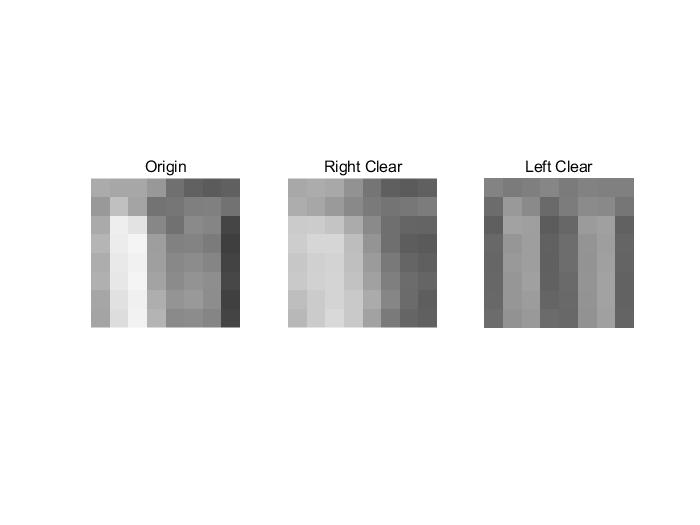
可以看到，将右侧清零得到的图片与原图差别不大，而且确实出现了轻微的横向的重影：在礼堂的大门与门之间的柱子处由黑突然转白，即存在高频分量，此处变化尤为明显——黑与白的交界明显缓和了。将该处放大查看效果更佳：



至于清除左边的四列，可以看到得到的图片已经基本上面目全非了，并且图片上有明显的纵向纹理（下图中方框内尤为明显），与预期符合：



为了更清晰看到像素点的变化，我们选取其中的一小块进行观察。本题中取 (81:88, 73:80) 部分进行观察：



如图可以看到，原图像在横向上有黑白的剧烈变化，但在清除了右侧的四列的小块中，剧烈变化得到了很大的缓和，即横向变化变得平缓了许多；而在清楚左侧四列的小块中，几乎只留下了纵向纹理，即几乎只有横向变化，而纵向变化不明显。这些均符合最开始的预期。

本题完整代码位于文件hw3\_2\_4\_3.m中。

### 将系数矩阵部分转置与旋转

下面对稀疏矩阵进行转置与旋转。

理论上分析，对于矩阵的转置，导致的结果是横向的变化幅度与纵向的变化幅度，即原来具有较强横向纹理的区块会变为较强的纵向纹理，原来具有较强纵向纹理的区块会变为较强的横向纹理。

更进一步注意到离散余弦变换和逆变换的公式：

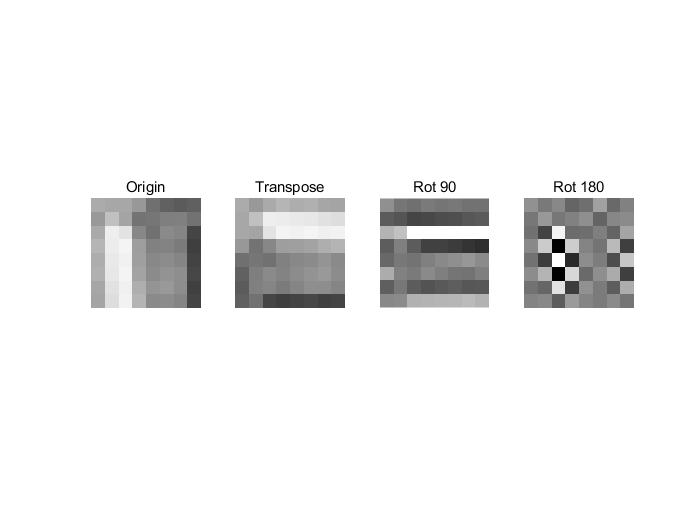
如果两边取转置：

会发现，系数矩阵的转置就是原矩阵转置的离散余弦变换！因此系数矩阵的转置做逆变换得到的就是原图像的转置！

对于矩阵旋转90度来说，由于对于一般的图片，低频分量是很大的，但是旋转90度会导致低频分量的系数变为了纵向变化高频分量的系数，因此得到的图片会有较强的横向纹理。

对于矩阵旋转180度来说，原来低频分量的系数变为了右下角的横纵向均高频的分量的系数，因此得到的图像应该会类似于棋盘状的黑白交替。

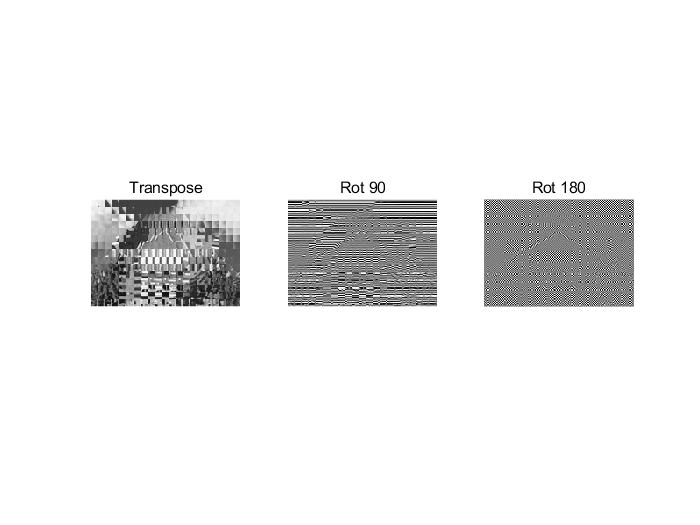
下面仍然取 (81:88, 73:80) 区块进行绘制：



可以看到，这块图片原来是有较强的纵向纹理的，即横向变化较为突出，而 在稀系数矩阵转置后得到的图片中，横向纹理比较突出。而且仔细观察就能发现，系数矩阵转置得到的图片就是原图片的转置，这与之前的理论分析是一致的。

对于旋转90度的图片来说，可以看到它有很强的横向纹理，即纵向色彩的变化非常剧烈；而系数矩阵旋转180度得到的图片也显然呈现棋盘状。这些与之前的理论分析也都是一致的。

下面将观察完整的图形：



由于图像时分成一个个小块进行变换的，因此转置的图像呈现出了明显的分块性——即每一块取转置再拼在一起，自然造成不连贯的结果。

对于旋转90度的图片，可以看到，图片显然呈现明显的横向纹理；而对于旋转180度的图片来说，其呈现的网格状、棋盘状也是符合预期的。

本题完整代码位于文件hw3\_2\_4\_4.m中。

### 差分编码的频率响应

对序列取差分的相反数（注意本题DC编码的差分是前项减后项）：

因此绘制零极点图和频响的代码为：

a = [1];

b = [-1, 1];

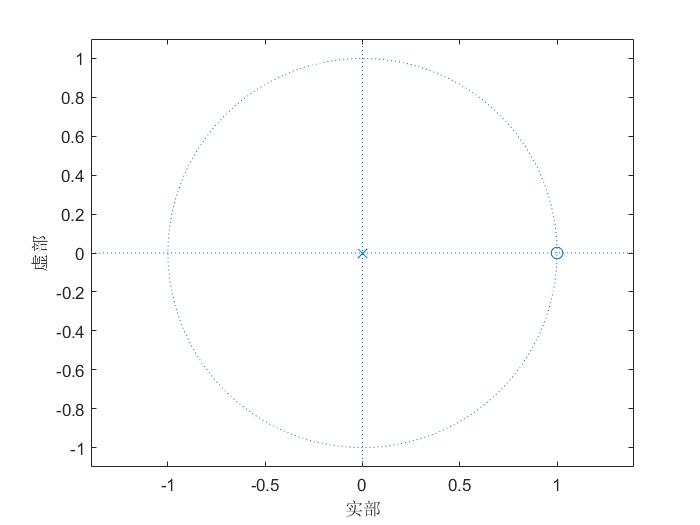
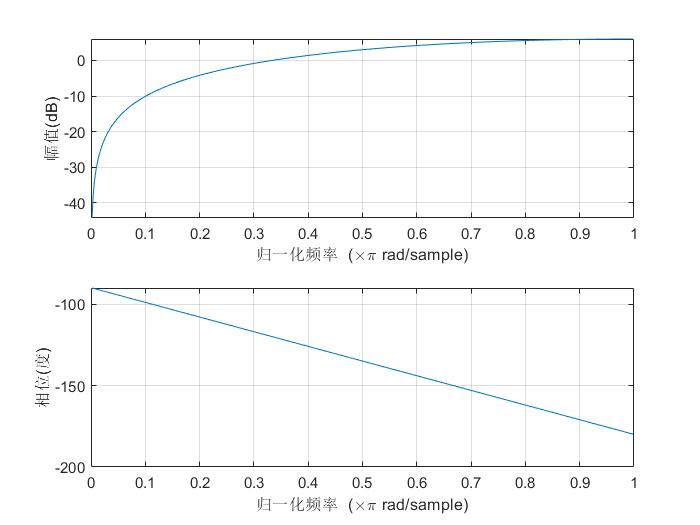
figure(1);

zplane(b, a);

figure(2);

freqz(b, a);

可以得到零极点图和频响：

从频响中可以看出，该滤波器是一个高通滤波器，这说明我们要滤除低频分量，即忽略小量。同时，这说明DC系数的高频分量更多。

本题完整代码位于文件hw3\_2\_4\_5.m中。

### DC预测误差与Category的关系

根据表格可以看出，Category是DC预测误差的绝对值加1，在对2取对数，然后向上取整得到的数。

此外，仔细观察也会发现，Category同时也是DC预测误差的二进制表示的位数。

### 利用MATLAB进行ZigZag扫描

由于在JPEG编码中，进行Zig-Zag的矩阵均为8\*8的矩阵，因此可以直接通过打表的方式完成Zig-Zag。将其封装为函数zig\_zag\_8，保存在文件zig\_zag\_8.m中。其代码如下：

function y = zig\_zag\_8(x)

order = [ ...

1, 2, 6, 7, 15, 16, 28, 29; ...

3, 5, 8, 14, 17, 27, 30, 43; ...

4, 9, 13, 18, 26, 31, 42, 44; ...

10, 12, 19, 25, 32, 41, 45, 54; ...

11, 20, 24, 33, 40, 46, 53, 55; ...

21, 23, 34, 39, 47, 52, 56, 61; ...

22, 35, 38, 48, 51, 57, 60, 62; ...

36, 37, 49, 50, 58, 59, 63, 64];

idx(order) = reshape([1 : 64], 8, 8);

y = x(idx)';

end

下面进行测试，将一个随机的8\*8的矩阵输入该函数：

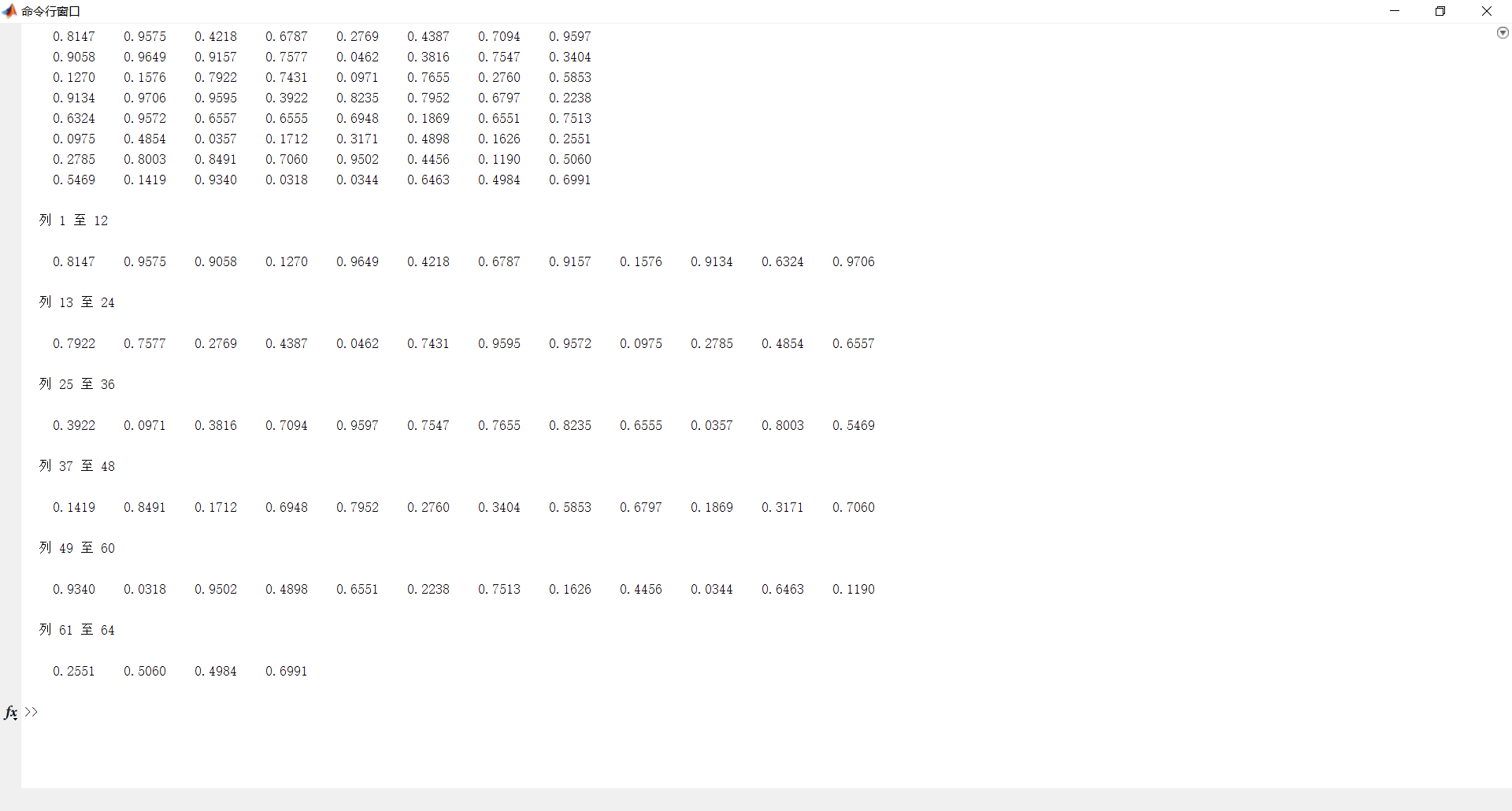
A = rand(8, 8);

y = zig\_zag\_8(A)';

disp(A);

disp(y);

结果如下图所示：



可以看到，确实将矩阵成功地进行了Zig-Zag扫描。

该测试脚本位于文件hw3\_2\_4\_7.m中。

### 图片的分块、DCT与量化

将图片划分为按8\*8的大小划分，对每一块减去128，再进行DCT，然后按量化表量化，再逐行依次排列，关键代码为：

hall\_gray = hall\_gray(1:24, 1:16);

img2proc = double(hall\_gray) - 128;

C = blockproc(img2proc, [8, 8], @(blk) zig\_zag\_8(round(dct2(blk.data) ./ QTAB)));

[h, w] =size(C);

hp = h / 64;

res = zeros([64, hp \* w]);

for i = 1 : 1 : hp

res(:, (i - 1) \* w + 1 : i \* w) = C((i - 1) \* 64 + 1 : i \* 64, 1 : w);

end

则得到的res即为本题的目标。

本题完整代码位于文件hw3\_2\_4\_8.m中。

### JPEG编码

首先我们需要一个将十进制转化为二进制数组的函数，该函数如下：

function y = dec2bin\_array(x)

if x == 0

y = [];

else

y = double(dec2bin(abs(x))) - '0';

if x < 0

y = ~y;

end

end

end

接下来进行DC编码。DC编码需要将Huffman编码与1的补码进行拼接。设上个问题得到的量化矩阵为C\_tilde（即上个问题中的res），DC编码如下：

dc = C\_tilde(1, :)';

diff\_dc = [dc(1); -diff(dc)];

category\_plus\_one = min(ceil(log2(abs(diff\_dc) + 1)), 11) + 1;

dc\_stream = arrayfun(@(i) ...

[DCTAB(category\_plus\_one(i), 2 : DCTAB(category\_plus\_one(i), 1) + 1), ...

dec2bin\_array(diff\_dc(i))]', ...

[1 : length(diff\_dc)]', 'UniformOutput', false);

dc\_stream = cell2mat(dc\_stream);

得到的dc\_stream即为DC码流。

接下来进行AC编码，采用的方法与DC编码类似，通过循环遍历每个块的信息，然后进行熵编码。

关键代码如下：

ac = C\_tilde(2 : end, :);

Size = min(ceil(log2(abs(ac) + 1)), 10);

ac\_stream = [];

ZRL = [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1];

EOB = [1, 0, 1, 0];

for i = 1 : 1 : size(ac, 2)

this\_ac = ac(:, i);

this\_Size = Size(:, i);

last\_not\_zero\_idx = 0;

not\_zero = this\_ac ~= 0;

while sum(not\_zero) ~= 0

[~, new\_idx] = max(not\_zero);

Run = new\_idx - last\_not\_zero\_idx - 1;

num\_of\_ZRL = floor(Run / 16);

Run = mod(Run, 16);

this\_tab\_row = Run \* 10 + this\_Size(new\_idx);

ac\_stream = [ac\_stream, repmat(ZRL, num\_of\_ZRL, 1), ACTAB(this\_tab\_row, 4 : ACTAB(this\_tab\_row, 3) + 3), dec2bin\_array(this\_ac(new\_idx))];

not\_zero(new\_idx) = 0;

last\_not\_zero\_idx = new\_idx;

end

ac\_stream = [ac\_stream, EOB];

end

ac\_stream = ac\_stream';

得到的ac\_stream即为AC码流。

最后将码流和图像的宽度和高度保存到文件jpegcodes.mat当中：

img\_height = size(hall\_gray, 1);

img\_width = size(hall\_gray, 2);

save('jpegcodes.mat', 'dc\_stream', 'ac\_stream', 'img\_height', 'img\_width');

本问题的完整代码位于文件hw3\_2\_4\_9.m中。

### 计算压缩比

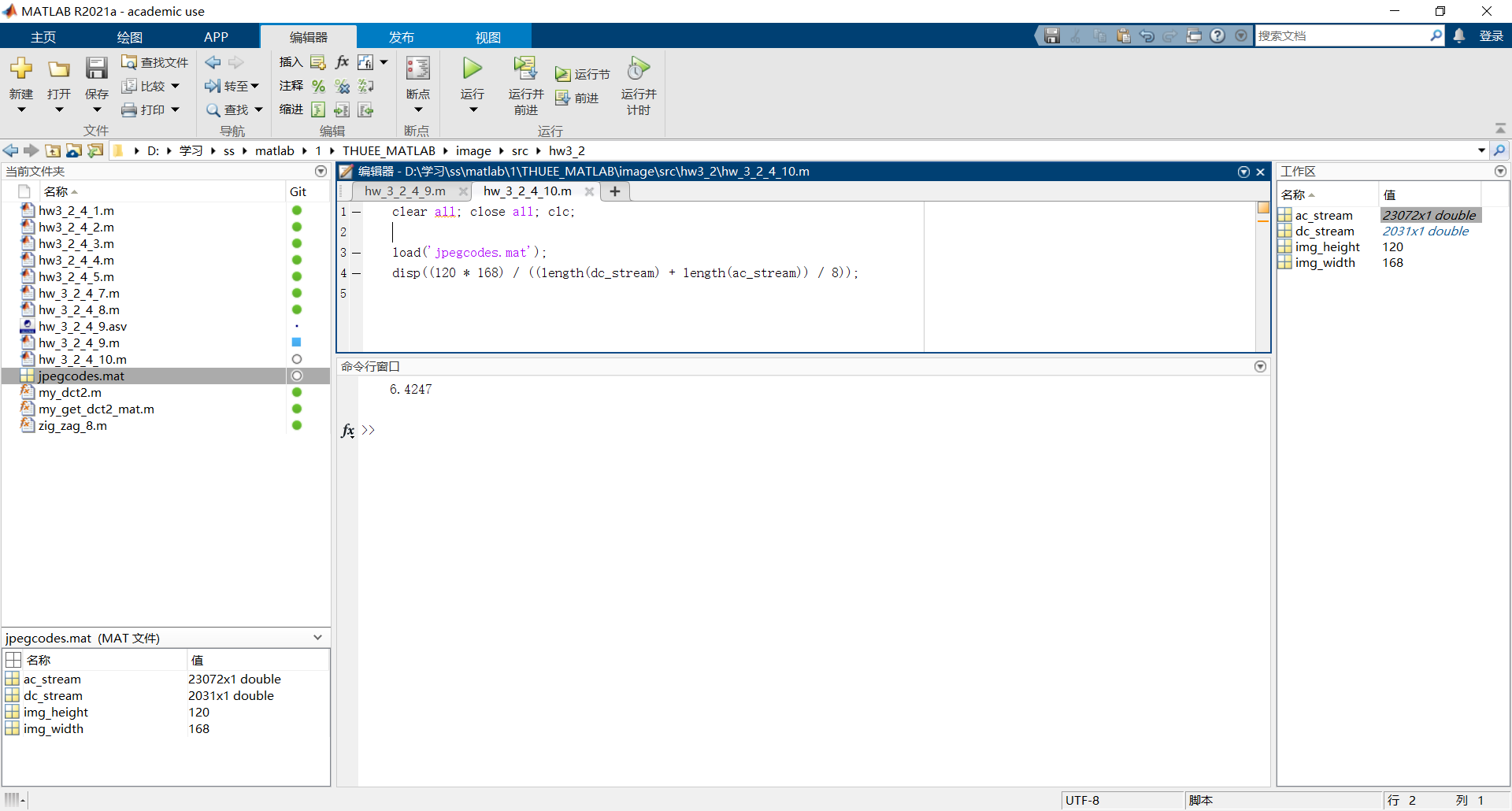
原图像为120\*168的灰度图，每个像素占一字节，因此共占用空间20160字节。而压缩后，占用空间为DC码流的比特数与AC码流的比特数值和除以8（化为字节数）。两者之比即为压缩比。

加载上一问题得到的脚本计算压缩比：

load('jpegcodes.mat');

disp((120 \* 168) / ((length(dc\_stream) + length(ac\_stream)) / 8));

运行脚本，DC码流长度为2031比特，AC码流长度为23072比特。下图为MATLAB计算结果：



因此压缩比约为6.4247。

本问题完整代码位于文件hw3\_2\_4\_10.m中。

### JPEG解码

接下来进行解码。先进行DC解码。解码与编码的过程正好相反，先进行熵解码。对此，我的解决方案是，在DC的编码表DCTAB内逐行对码流进行比对，如果比对成功则开始解码。这部分解码的关键代码如下：

hp = img\_height / 8;

wp = img\_width / 8;

dc\_decoding\_res = zeros([hp \* wp, 1]);

i = 1;

cnt = 1;

while i <= length(dc\_stream)

for j = 1 : 1 : size(DCTAB, 1)

% Omitted overbound judge here!!!

if DCTAB(j, 2 : DCTAB(j, 1) + 1)' == dc\_stream(i : i + DCTAB(j, 1) - 1)

i = i + DCTAB(j, 1);

category = j - 1;

if category == 0

dc\_decoding\_res(cnt) = 0;

else

Magnitude = dc\_stream(i : i + category - 1);

if Magnitude(1) == 1

dc\_decoding\_res(cnt) = bin2dec(char(Magnitude' + '0'));

else

dc\_decoding\_res(cnt) = -bin2dec(char(~Magnitude' + '0'));

end

end

i = i + category;

break;

end

end

cnt = cnt + 1;

end

其中dc\_decoding\_res即为熵解码的结果。

由于该结果原来是差分得到的，因此现在进行反差分：

dc\_cum = cumsum([dc\_decoding\_res(1); -dc\_decoding\_res(2:end)]);

DC解码完毕，接下来进行AC解码。

AC解码方式与DC解码大致相同，不同的有两点：一是AC解码时，需要单独考虑EOB和ZRL；二是AC的哈夫曼表中ACTAB的码长不是递增的，因此正在匹配的码长可能比剩余的AC码流总长度要长造成越界，因此需要增加越界判断。AC解码的关键代码如下：

ZRL = [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1];

EOB = [1, 0, 1, 0];

ac\_decoding\_res = zeros([63, hp \* wp]);

i = 1;

block\_cnt = 1;

inner\_block\_cnt = 1;

while i < length(ac\_stream)

if i + length(ZRL) - 1 <= length(ac\_stream) && sum(~(ac\_stream(i : i + length(ZRL) - 1) == ZRL')) == 0

inner\_block\_cnt = inner\_block\_cnt + 16;

i = i + length(ZRL);

elseif i + length(EOB) - 1 <= length(ac\_stream) && sum(~(ac\_stream(i : i + length(EOB) - 1) == EOB')) == 0

block\_cnt = block\_cnt + 1;

inner\_block\_cnt = 1;

i = i + length(EOB);

else

for j = 1 : 1 : size(ACTAB, 1)

if i + ACTAB(j, 3) - 1 > length(ac\_stream)

continue;

end

if ACTAB(j, 4 : ACTAB(j, 3) + 3) == ac\_stream(i : i + ACTAB(j, 3) - 1)'

inner\_block\_cnt = inner\_block\_cnt + ACTAB(j, 1);

i = i + ACTAB(j, 3);

Amplitude = ac\_stream(i : i + ACTAB(j, 2) - 1);

if Amplitude(1) == 1

ac\_decoding\_res(inner\_block\_cnt, block\_cnt) = bin2dec(char(Amplitude' + '0'));

else

ac\_decoding\_res(inner\_block\_cnt, block\_cnt) = -bin2dec(char(~Amplitude' + '0'));

end

i = i + ACTAB(j, 2);

inner\_block\_cnt = inner\_block\_cnt + 1;

break;

end

end

end

end

然后将DC和AC解码得到的结果拼合成矩阵：

decoding\_res = [dc\_cum'; ac\_decoding\_res];

然后将结果按逐行重新排列为图片的行、列分布：

decoding\_C = zeros(64 \* hp, wp);

for i = 1 : 1 : hp

decoding\_C((i - 1) \* 64 + 1 : i \* 64, :) = decoding\_res(:, (i - 1) \* wp + 1 : i \* wp);

end

接下来要将其分块进行反Zig-Zag。类似于Zig-Zag，我们先编写一个用于对长度为64的序列进行反Zig-Zag的函数i\_zig\_zag\_8：

function y = zig\_zag\_8(x)

order = [ ...

1, 2, 6, 7, 15, 16, 28, 29; ...

3, 5, 8, 14, 17, 27, 30, 43; ...

4, 9, 13, 18, 26, 31, 42, 44; ...

10, 12, 19, 25, 32, 41, 45, 54; ...

11, 20, 24, 33, 40, 46, 53, 55; ...

21, 23, 34, 39, 47, 52, 56, 61; ...

22, 35, 38, 48, 51, 57, 60, 62; ...

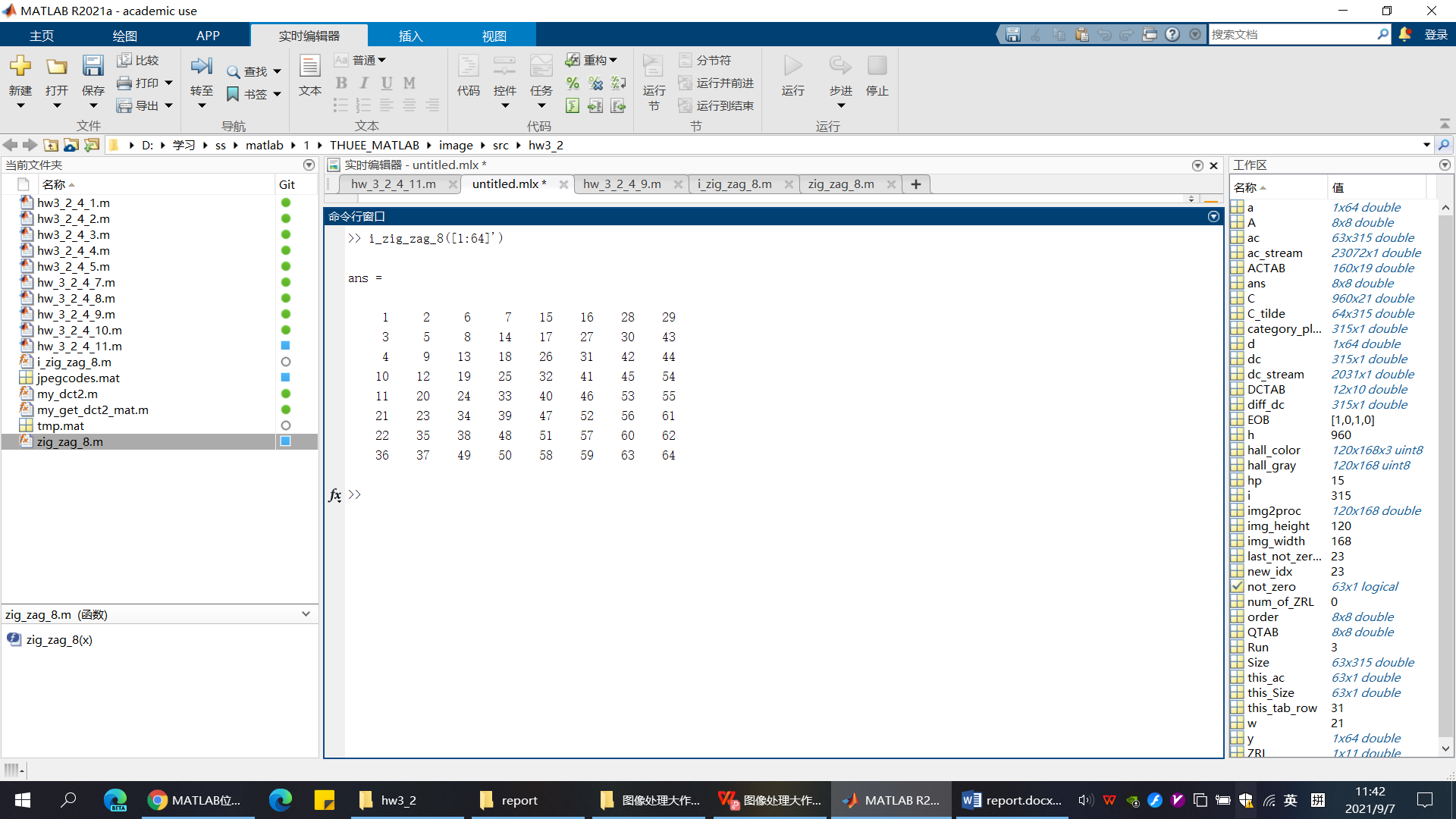
36, 37, 49, 50, 58, 59, 63, 64];

idx(order) = reshape([1 : 64], 8, 8);

y = x(idx)';

end

对其进行测试：若输入为1~64的连续数字，则输出应该与上面的order数组相同：



结果符合预期。

然后依次分块进行反Zig-Zag、反量化、离散余弦逆变换：

decoding = blockproc(decoding\_C, [64, 1], @(blk) idct2(i\_zig\_zag\_8(blk.data) .\* QTAB));

最终得到结果decoding。将其加上亮度128得到图片：

decoding\_img = uint8(decoding + 128);

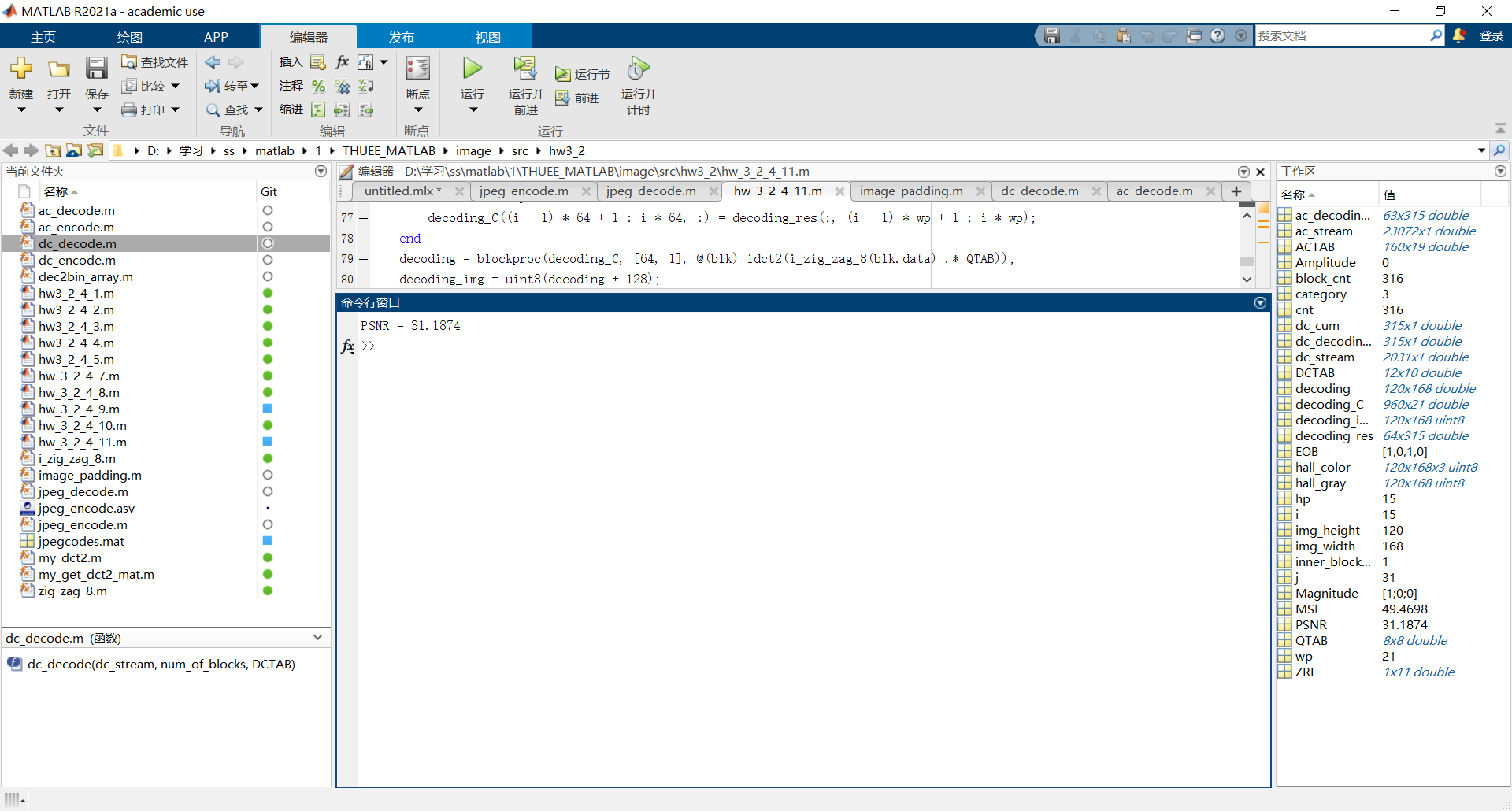
根据公式计算结果的PSNR：

MSE = sum((double(decoding\_img) - double(hall\_gray)).^2, 'all') / (img\_height \* img\_width);

PSNR = 10 \* log10(255 \* 255 / MSE);

disp("PSNR = " + PSNR);

输出结果为：



因此PSNR的值为31.1874 dB。

绘制图片，并与原图形进行比对：

subplot(1, 2, 1);

imshow(hall\_gray);

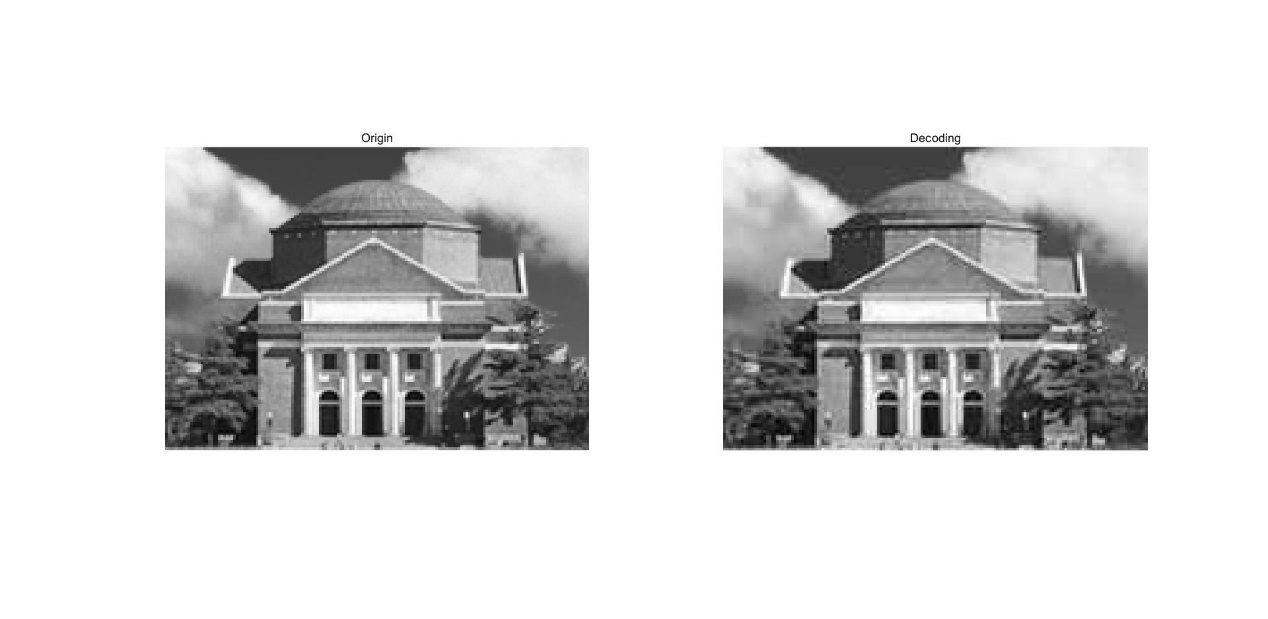
title('Origin');

subplot(1, 2, 2);

imshow(decoding\_img);

title('Decoding');

得到下图：



其中左侧是原图片，右侧是解码后的图片。总体来看，两图差别不大，都能比较清晰地看出原本的情景。但是仔细看会发现，解码后的图较原图稍模糊，尤其是色彩的剧烈变化处变化变缓，这是滤掉高频分量的结果。此外，在解码后的图中，尤其是白云内部，可以看到较为明显的方块状，这可能是按8个像素分块 量化的结果。

本问题完整代码位于文件hw3\_2\_4\_11.m中。

### 减小量化步长

将上述编码、解码功能分别封装成函数jpeg\_encode和jpeg\_decode，并且增加了边长不足8的倍数时进行用零进行边界扩展的功能；并将计算PSNR封装为函数mypsnr，将量化表除以2进行编码和解码：

QTAB = QTAB / 2;

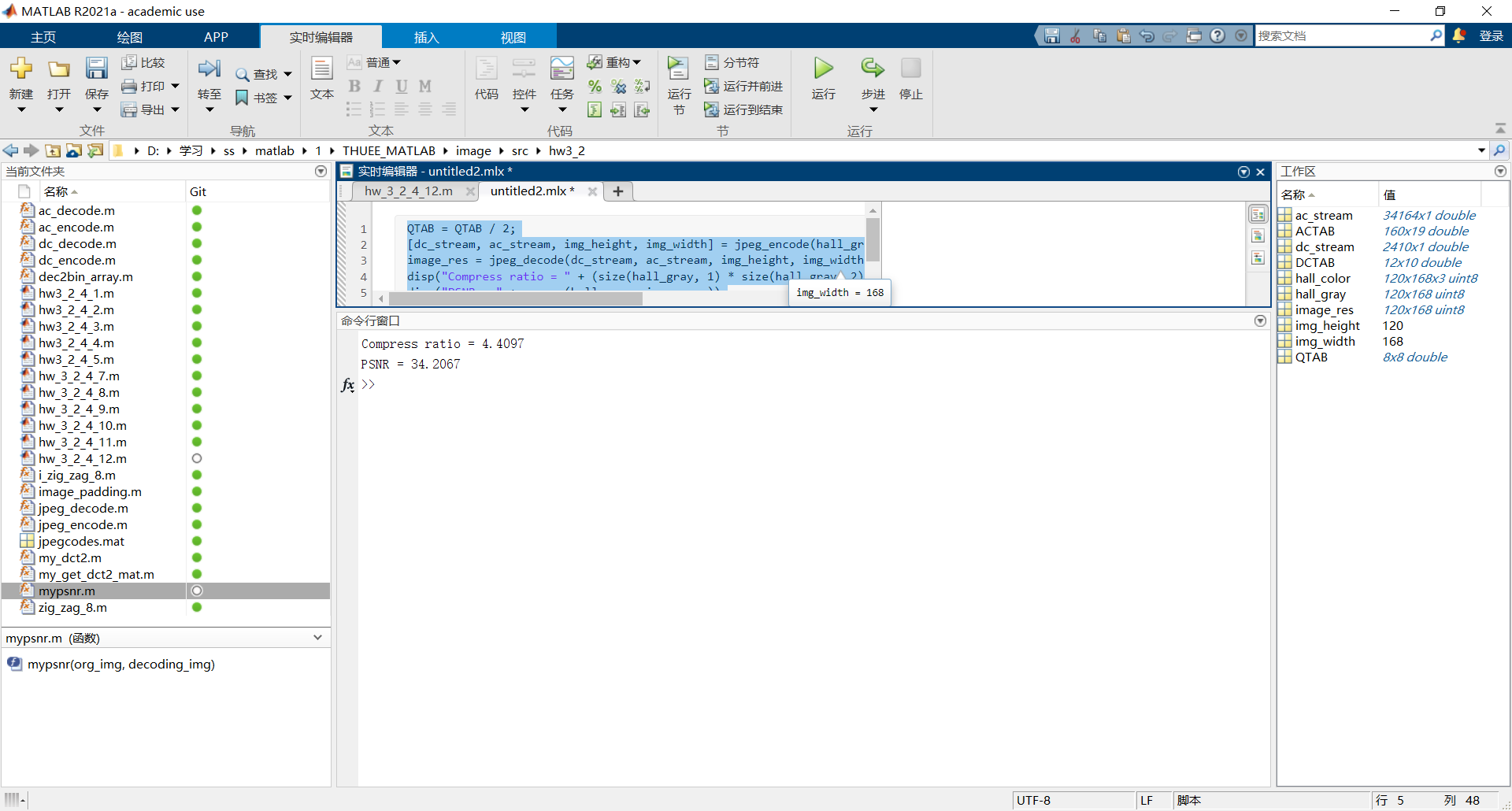
[dc\_stream, ac\_stream, img\_height, img\_width] = jpeg\_encode(hall\_gray, DCTAB, ACTAB, QTAB);

image\_res = jpeg\_decode(dc\_stream, ac\_stream, img\_height, img\_width, DCTAB, ACTAB, QTAB);

disp("Compress ratio = " + (size(hall\_gray, 1) \* size(hall\_gray, 2)) / ((length(dc\_stream) + length(ac\_stream)) / 8));

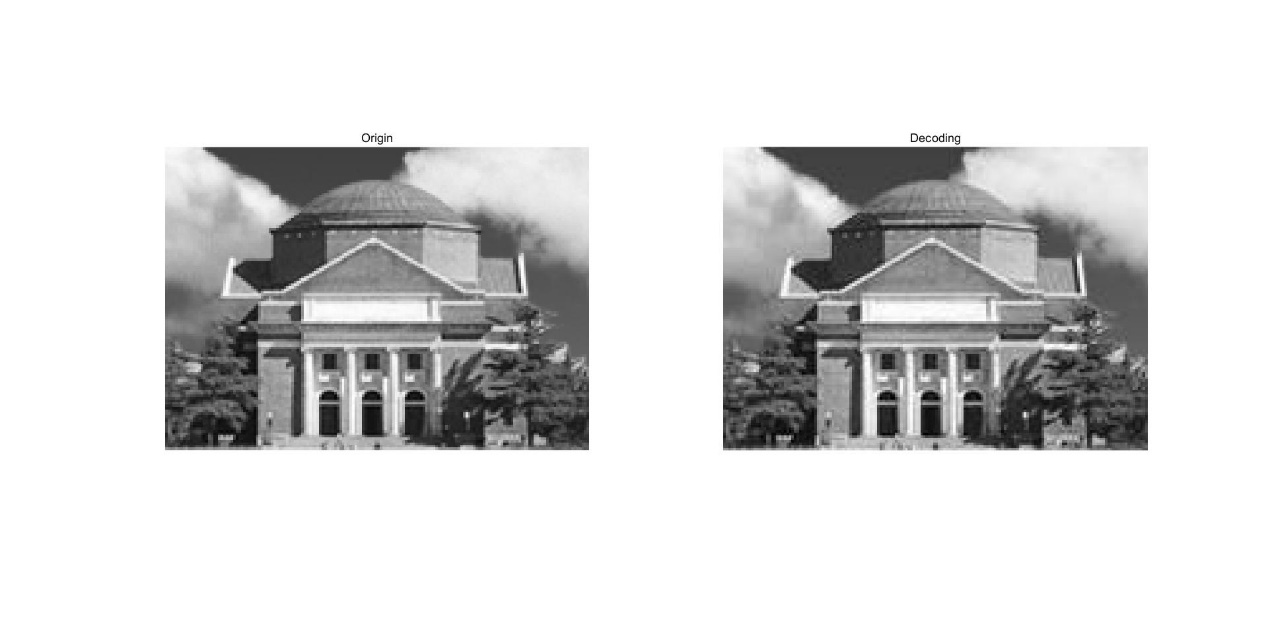
disp("PSNR = " + mypsnr(hall\_gray, image\_res));

得到结果：



因此可以看到，压缩比约为4.4097，略有降低，而PSNR为34.2067 dB，略有升高。推测其原因，因为量化步长缩短，使得高频信息损失更少，因此PSNR升高；而量化步长缩短导致得到的量化值增大，因此编码结果的长度变长，减低压缩比。

绘制图片得到：

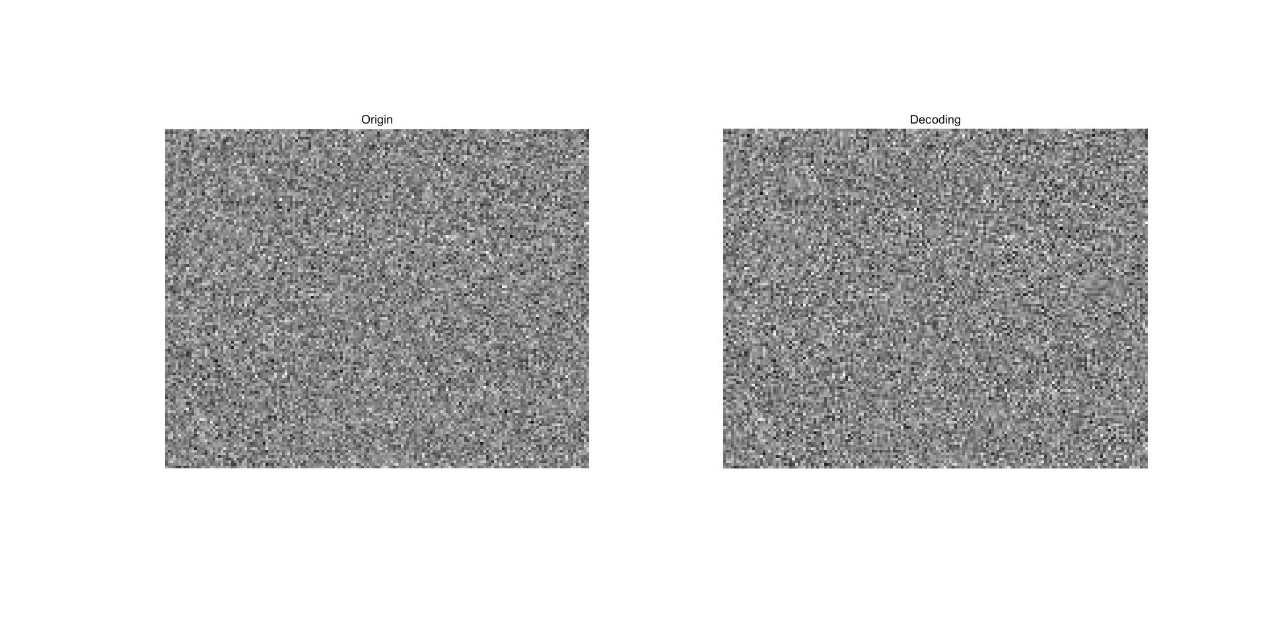


可以看到，图片质量与没有将量化步长除以2的时候相比，可能略有提高，但是变化不是很大，和之前的效果难以用肉眼分辨。

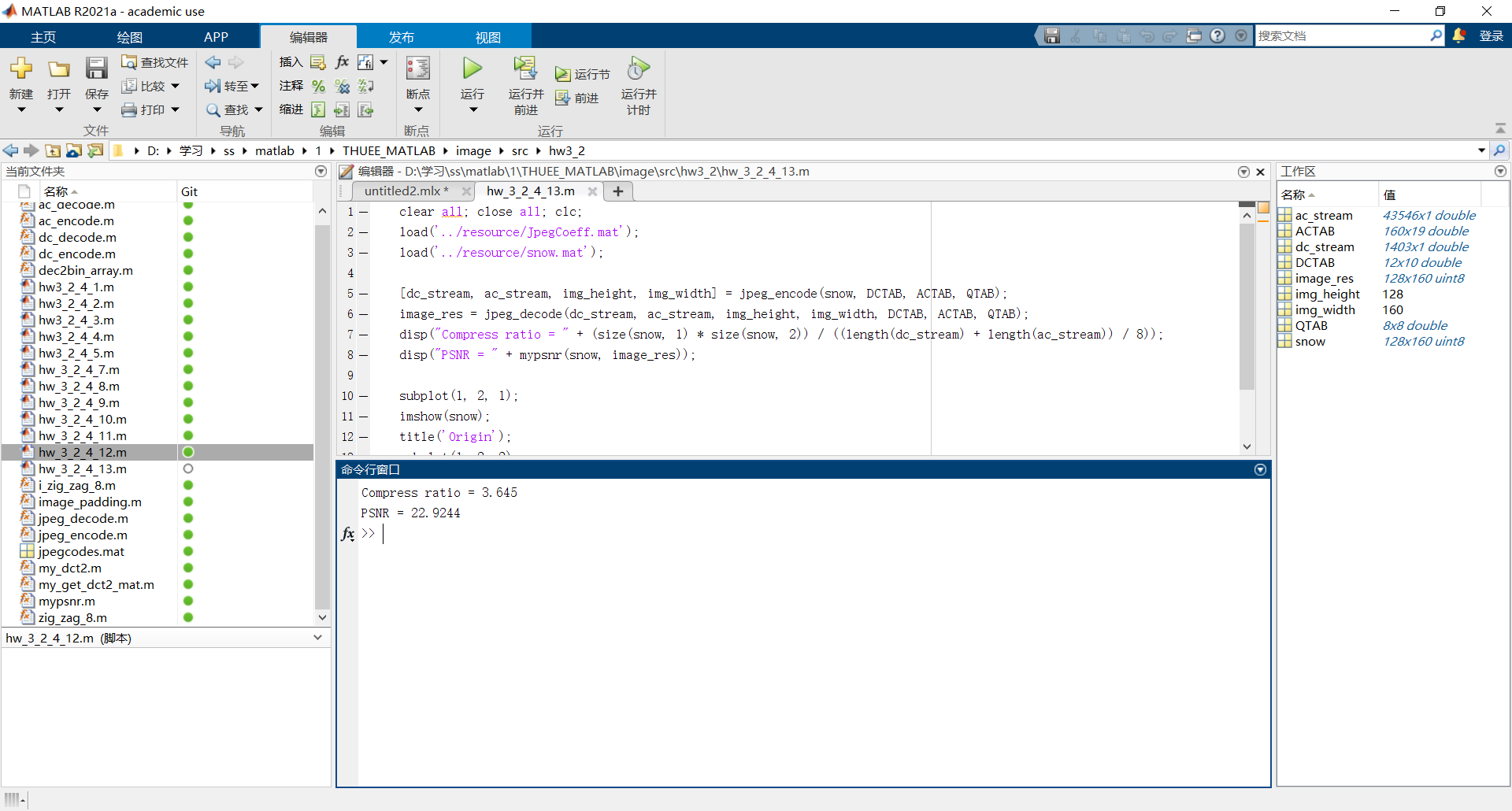
本问题的完整代码位于文件hw3\_2\_4\_12.m中。

### 处理雪花图像

将输入图片变成雪花图像，得到的图片如下图：



压缩比和PSNR分别为：



即压缩比为3.645，PSNR为22.9244 dB。相比来看，雪花图像的压缩比较之前更小，而且PSNR值也更低。仔细对比两幅图，压缩并解码后的图片也失真较多，与原图片相似度降低。

究其原因，电视上雪花图的雪花是在电视屏幕上随机出现的，并且深浅度不一，因此图片的高频分量很大，故高频分量不容易被量化至0或其他较低的值。因此压缩比相对较低；而高频分量被量化导致高频分量存在失真，而雪花图高频分量较多，因此图片的失真比较严重，PSNR较低。

## 信息隐藏

### 空域隐藏

将图片进行空域信息隐藏。我们先用随机01序列当作信息进行隐藏，然后检查其正确率。为了让结果更准确，我们重复10次实验。关键代码如下：

test\_time = 10;

[h, w] = size(hall\_gray);

correct\_ratio = zeros([test\_time, 1]);

for i = 1 : 1 : test\_time

seq = uint8(randi([0, 1], h, w));

secret\_image = bitset(hall\_gray, 1, seq);

[dc, ac, imh, imw] = jpeg\_encode(secret\_image, DCTAB, ACTAB, QTAB);

decoding\_image = jpeg\_decode(dc, ac, imh, imw, DCTAB, ACTAB, QTAB);

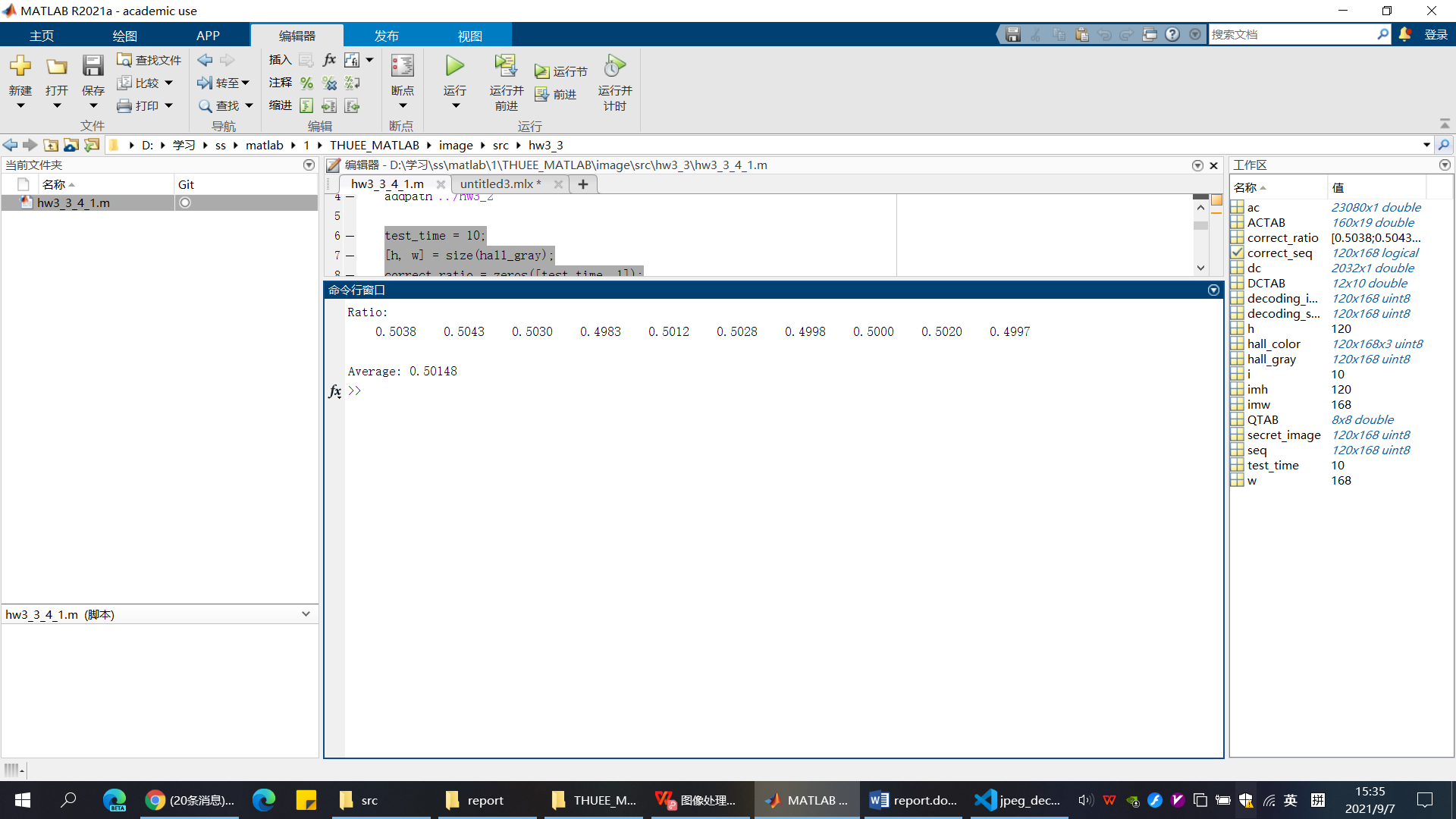
decoding\_seq = bitand(decoding\_image, uint8(ones([h, w])));

correct\_seq = ~xor(seq, decoding\_seq);

correct\_ratio(i) = sum(correct\_seq, 'all') / (h \* w);

end

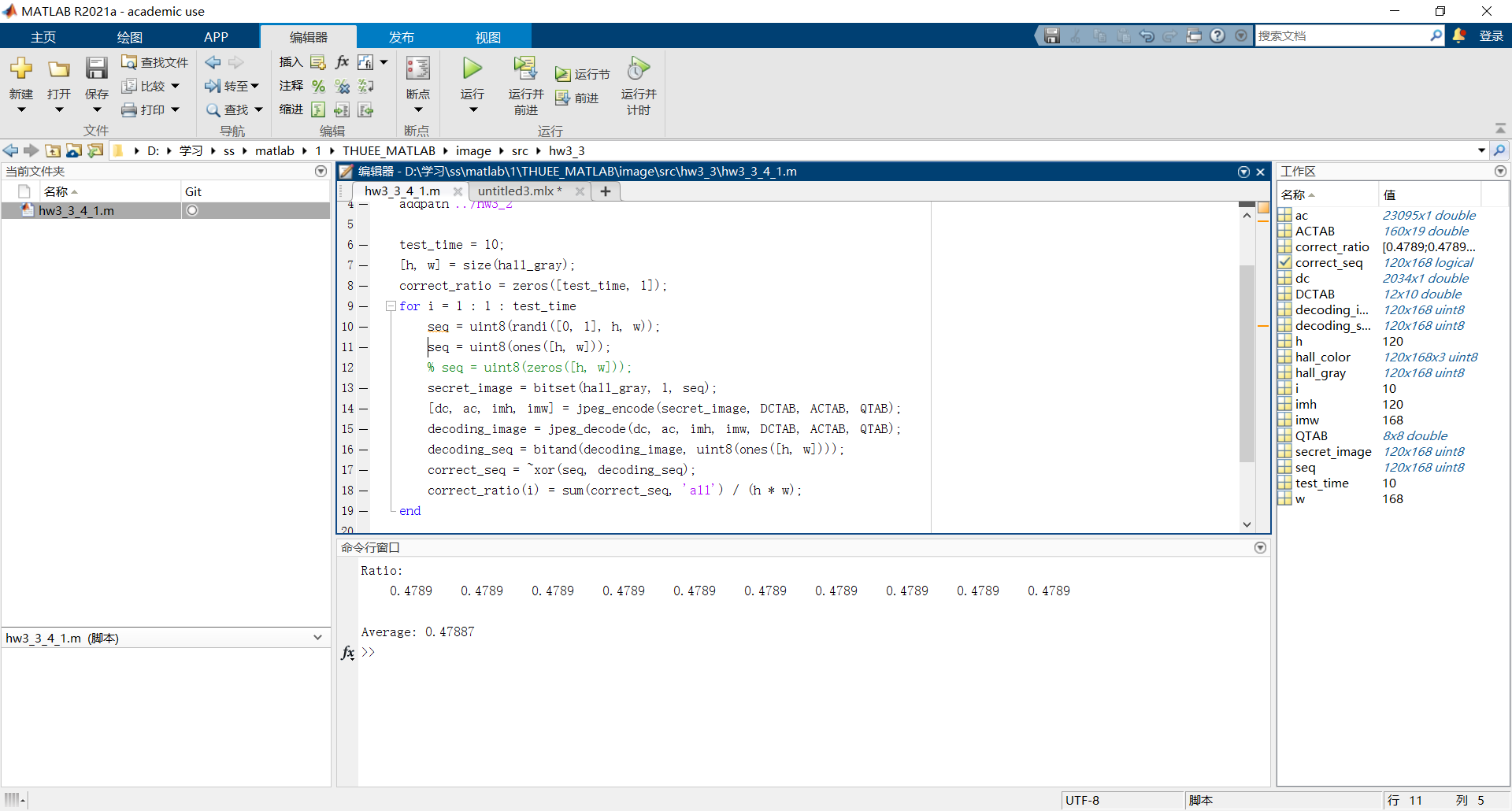
运行结果：



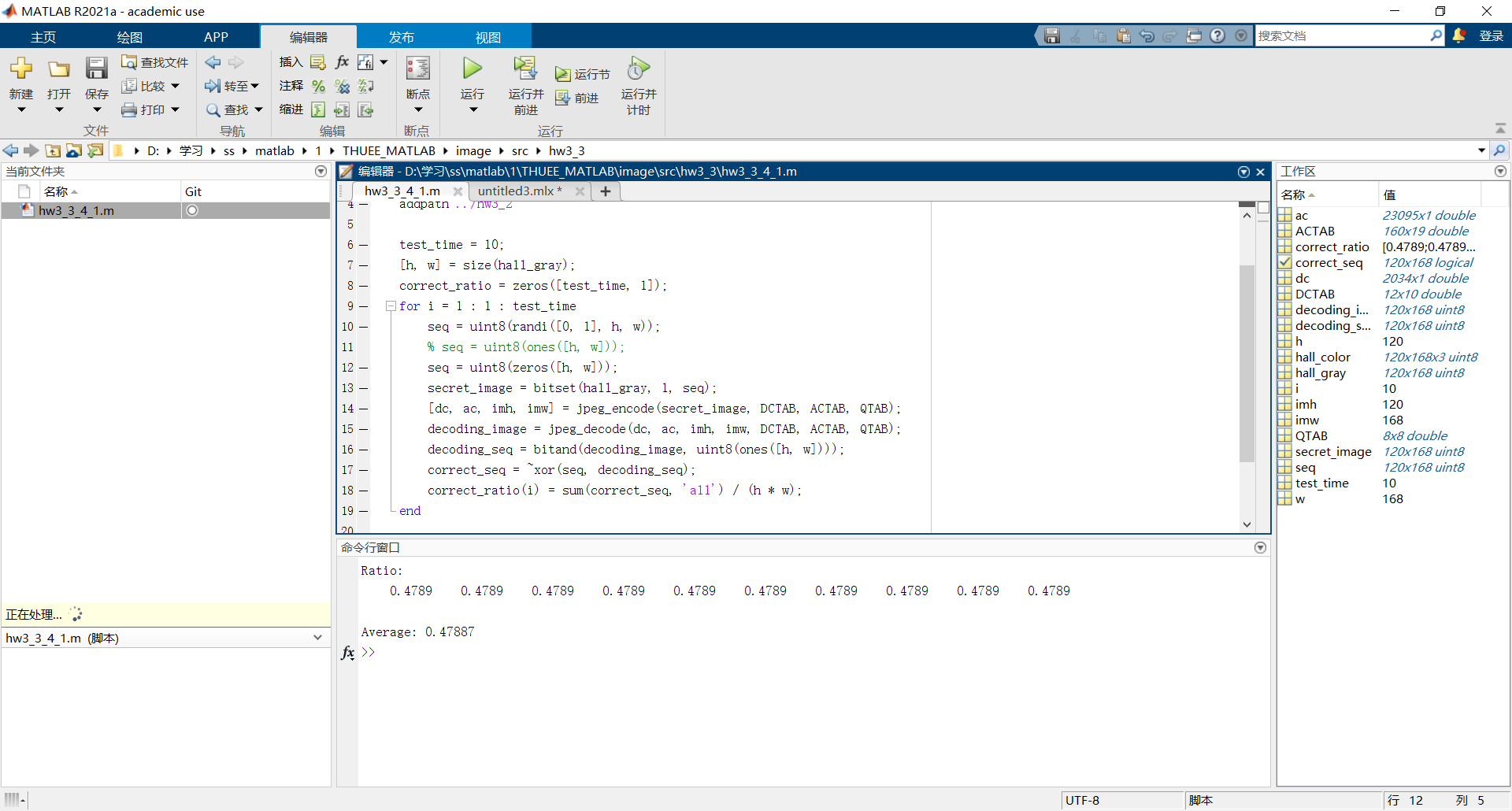
可以看到正确率几乎都是0.5上下——这代表信息根本没有被获取到——有因为我们都是01序列，所以即使是随便猜测信息都回平均有50% 的正确率。因此信息完全被丢失！

据此我本来存在一个猜想：随机的序列不能代表真实信息，真实信息可能有一定的规律，而随机的信息代表高频，高频分量很容易被过滤掉，因此我决定使用全0信息和全1信息进行测试。

全1信息的代码与结果如下：



全0信息的代码与结果如下：



可以看到它们的正确率也还都是50%上下，也基本上没有保留任何信息。因此，问题不在于高频分量上。我认为，问题出在量化过程。由于量化过程会将比较接近的数字映射到同一个数字上，因此相接近的数字是无法做区分的，即在量化过程中损失掉了。而我们在空域进行信息隐藏时，由于只修改最低的一位，因此对数字影响可能不大，故信息在量化过程中丢失。

可以看到，空域隐藏的抗JPEG编码能力极低。

本问题的全部代码位于文件hw3\_3\_4\_1.m中。

### DCT域隐藏

下面对量化后的量化系数进行处理以隐藏信息，下面采用三种方法进行隐藏。直接的操作对象是进行量化并且进行了Zig-Zag且逐行排列后的矩阵。

1. 信息隐藏在每个系数的最低位

将每个系数的最低为均置为要隐藏的信息，该功能封装为freqdom\_hide1函数，位于freqdom\_hide1.m函数文件中，关键代码如下：

function Out = freqdom\_hide1(In, info)

Out = double(bitset(int64(round(In)), 1, info));

end

对应的解密算法封装为freqdom\_find1函数，位于freqdom\_find1.m函数文件中：

function info = freqdom\_find1(In)

info = uint8(bitget(int64(round(In)), 1));

end

1. 信息隐藏在部分系数的最低位

另外一种方法是只挑选部分系数隐藏信息。为了尽量减小对图像的影响，我决定选取量化矩阵QTAB中最小元素的位置用于隐藏信息。这是因为量化矩阵的元素越小，对应位置的量化系数每一个单位代表的值越小，能最大程度上减小图像的失真。基于这个思路，设计的加密代码封装为freqdom\_hide2函数，位于freqdom\_hide2.m函数文件中。关键代码如下：

function Out = freqdom\_hide2(In, info, QTAB)

[~, min\_idx] = min(zig\_zag\_8(QTAB));

In(min\_idx, :) = double(bitset(int64(round(In(min\_idx, :))), 1, info));

Out = In;

end

对应的解密算法封装为freqdom\_find2函数，位于freqdom\_find2.m函数文件中：

function info = freqdom\_find2(In, QTAB)

[~, min\_idx] = min(zig\_zag\_8(QTAB));

info = uint8(bitget(int64(round(In(min\_idx, :))), 1));

end

1. 信息隐藏在最后一个非零位之后

基于此思路设计的加密代码封装为freqdom\_hide3函数，位于freqdom\_hide3.m函数文件中，关键代码如下：

function Out = freqdom\_hide3(In, info)

info = double(info);

info(info == 0) = -1;

Out = zeros(size(In));

for i = 1 : 1 : size(In, 2)

this\_seq = In(:, i);

if this\_seq == 0

Out(:, i) = [info(i); In(2 : end, i)];

else

not\_zero = flipud(this\_seq ~= 0);

this\_seq = flipud(this\_seq);

[~, idx] = max(not\_zero);

if idx == 1

this\_seq(1) = info(i);

else

this\_seq(idx - 1) = info(i);

end

Out(:, i) = flipud(this\_seq);

end

end

end

对应的解密算法封装为freqdom\_find3函数，位于freqdom\_find3.m函数文件中：

function info = freqdom\_find3(In)

info = uint8(zeros([1, size(In, 2)]));

for i = 1 : 1 : size(In, 2)

this\_seq = flipud(In(:, i));

if this\_seq == 0

info(i) = 0; % error

disp('Warning: Get message error: All is zero!');

else

is\_zero = this\_seq ~= 0;

[~, idx] = max(is\_zero);

if this\_seq(idx) == 1

info(i) = 1;

elseif this\_seq(idx) == -1

info(i) = 0;

else

info(i) = 0; % error

disp('Warning: Get message error: Not 1 or -1!');

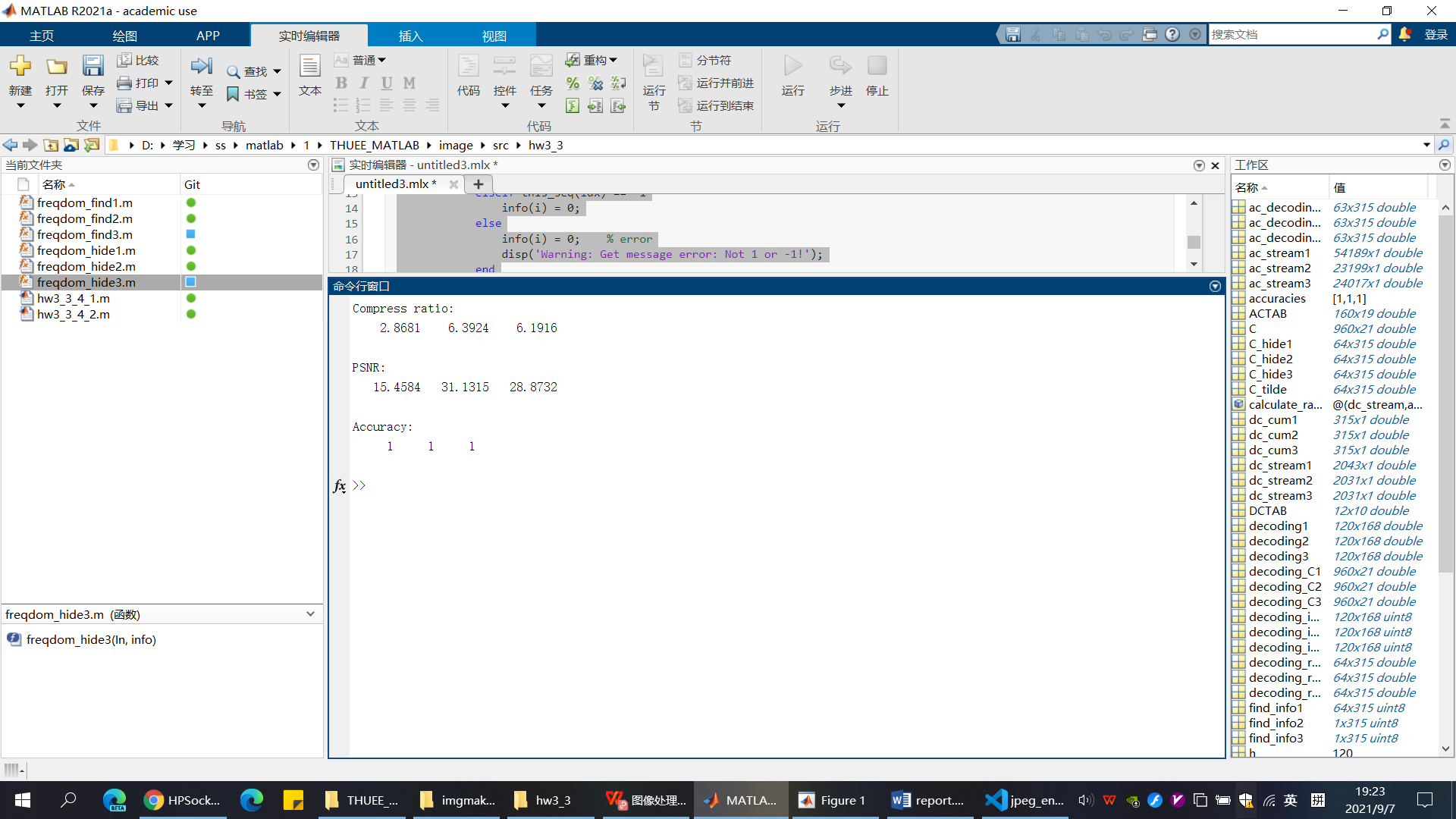
end

end

end

end

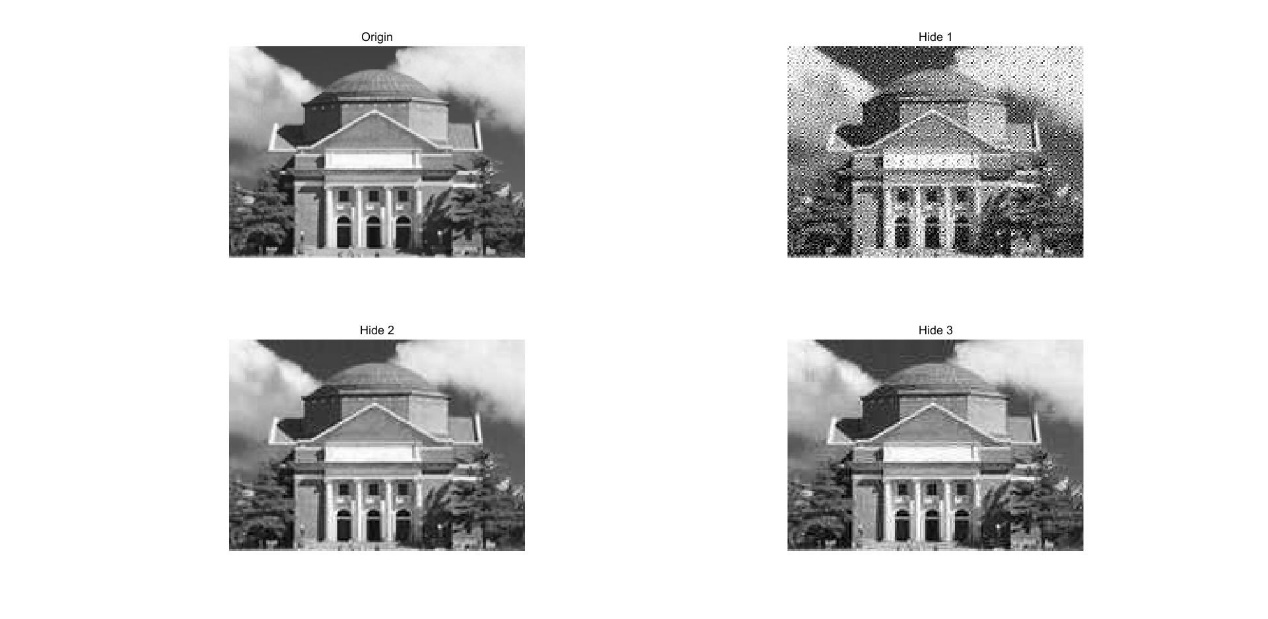
下面进行测试，信息仍然采用随机的01数值，运行MATLAB代码，得到的信息准确率、压缩比以及PSNR：



可以看到，第一种方法的压缩比和PSNR最低；而第二种方法和第三种方法的压缩比和失真率都和没有隐藏信息是更接近。初步推测这是因为第一种方法的信息量大，对每个系数都有所改变，因此图像失真也比较严重，影响了PSNR；并且每个系数都改变导致了编码中0的数量较少以及码字变长等，影响了压缩率。

此外，第二种方法比第三种方法还要优一些，压缩率和PSNR略高。推测这是因为第二种方法我是选取了QTAB最小的元素进行隐藏编码，因此对图像的影响较小，对PSNR影响小；并且，第三种方法在最后附加信息，本身便会增加码长，况且为零的值大多是高频分量，因此对图像影响也会较大。

将几幅图分别画出，如下：



可以看到，图片给人的感觉与上面理论分析相一致，第一种方法失真严重，第二种方法和第三种方法和原图片差别均不大，但第三种方法比第二种方法稍模糊。原因在上面已经进行了分析。

本问题的启动脚本为文件hw3\_3\_4\_2.m。