MATLAB 图像处理大作业

2019010562　无 97　潘修睿

一.

1. 略。

2. image1_2a.m, image1_2b.m

（a）使用参数方程生成一个包含圆的数组，将图片上这些点改为[255, 0, 0]即可实现。

（b）图片长宽都是 24 的倍数，因此分别切分成 length/24 和 width/24，交替将图片上的点设置为[0, 0, 0]即可。

二.

1. image2_1.m

将先预处理和后预处理后的两矩阵相减，求误差的最大值，发现只有 7.2475e-13。说明可以在变换域进行。

2. image2_2.m

编程实现二维 dct，自己实现的 dct 为 ans1，matlab 的 dct 为 ans2。将两矩阵相减，求误差的最大值，发现只有 3.2372e-11。说明实现是正确的。

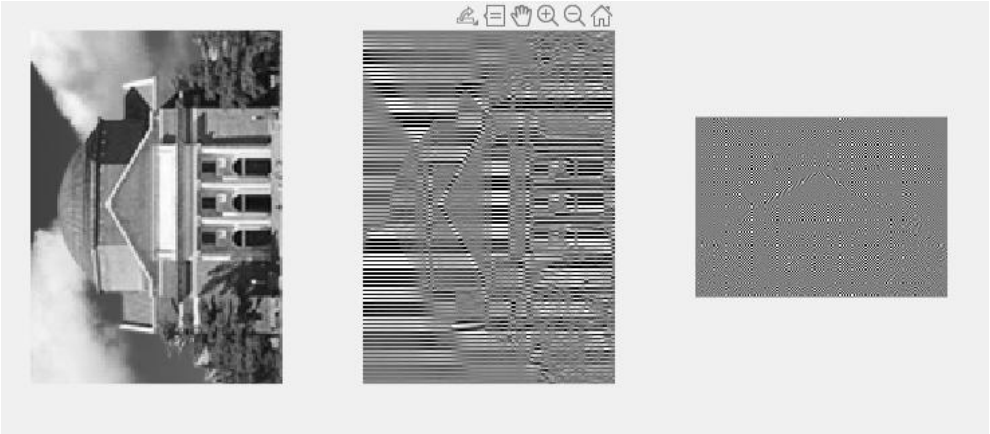
3. image2_3.m

从左到右分别为原图、右侧置零、左侧置零。



可以看出，将右侧置零几乎对画质无影响，而左侧置零失真严重。这是因为右侧是高频分量而左侧为低频分量系数，人眼对高频分量不敏感，因此难以察觉。

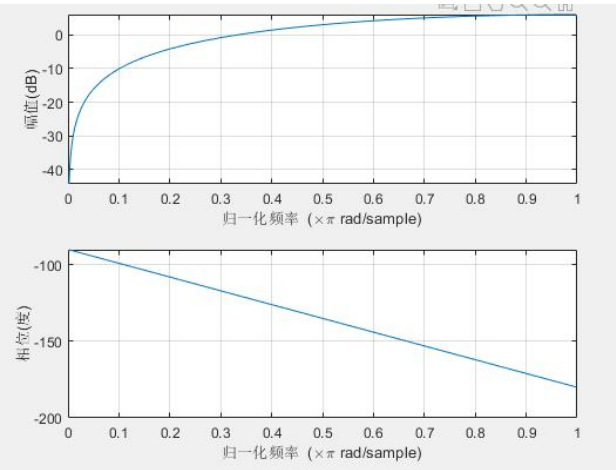
4. image2_4.m



从左至右分别为转置、旋转 90°和旋转 180°。

转置系数后、旋转 90°后，原图像都旋转了 90 度。而旋转 90 度和 180 度后，图像都发生了奇怪的失真模糊。

5. image2_5.m



频响如上，这是一个高通滤波器，DC 系数高频分量更多。

6. image2_6.m

若 DC 预测误差 DCerr 为 0，则 Category 为 0.

否则，`category = floor(log2(abs(DCerr))) + 1;`

7. image2_7.m

matlab 适合矩阵操作，因此我事先将 zigzag 扫描的顺序保存在一个数组中作为数组索引，利用 matlab 的 reshape 函数将原矩阵转换为列向量后，直接通过索引对该向量重排列，即可实现 zigzag 扫描。

8. image2_8.m

按照步骤进行分块、DCT、量化并 zigzag 扫描即可。此处截取部分 DCT 系数矩阵以说明正确性。

DCT

64x315 double

	1	2	3	4	5	6	7	8	9
1	57	51	9	-28	-32	-32	-32	-32	-32
2	1	3	22	5	0	0	0	0	0
3	1	-2	-40	-6	0	0	0	0	0
4	1	0	0	3	0	0	0	0	0
5	0	4	-2	-6	0	0	0	0	0
6	-1	-1	0	3	0	0	0	0	0
7	0	2	1	1	0	0	0	0	0
8	0	-2	5	-3	0	0	0	0	0
9	-1	2	-12	4	0	0	0	0	0
10	0	-2	0	-2	0	0	0	0	0
11	0	-1	0	1	0	0	0	0	0
12	0	1	0	-2	0	0	0	0	0
13	0	-1	0	2	0	0	0	0	0
14	0	1	0	-1	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0

9. images2_9.m

本题实现 jpeg 编码。

首先，通过与 2_8 同样的方法进行 DCT 变换，得到 DCT 系数矩阵。

然后，先进行 DC 编码。先差分编码后，将结果存储到 diff_coding 矩阵中，然后循环进行编码。

最后，进行 AC 编码。然后将 DCcode、ACcode、height、width 写入 jpegcodes.mat 中，具体实现细节见代码。

10.

计算压缩率

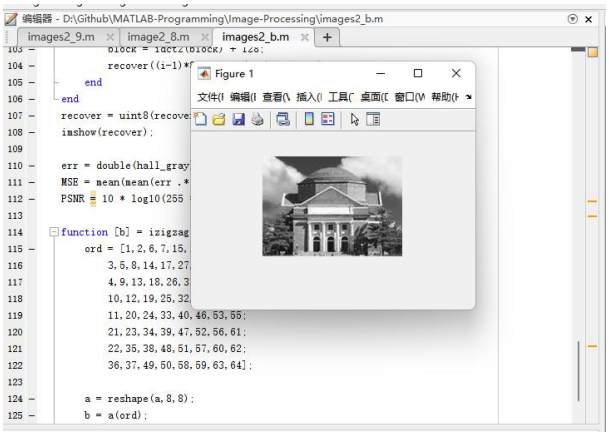
原文件大小为 120*168*8 = 161280 bit

压缩后大小为 2054+23072 = 25126 bit

压缩比 161280/25126 = 6.419

11. images2_b.m

本题实现 jpeg 解码和复原。将编码过程的逆过程实现即可，先进行 DC 解码，再进行 AC 解码。最后进行逆 zigzag 扫描并将分块合并，复原的图像保存到 recover 变量中。

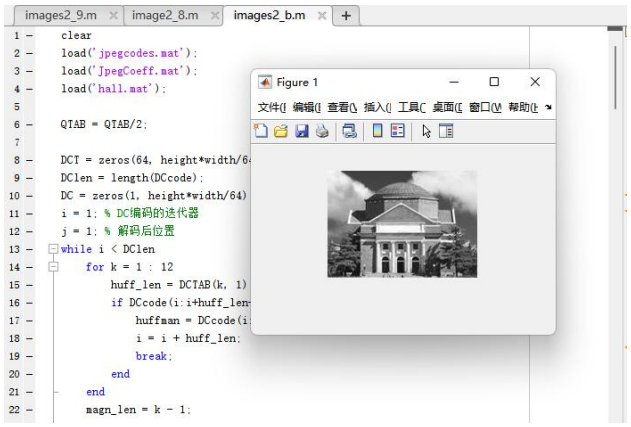


完美复原出了图像！主观上评价，和原图几乎没有差别。

最后计算 PSNR = 31.1874，看来还是稍有差别，但人眼已经很难分辨了。

12.

只需将 images2_9.m 和 images2_b.m 中的 QTAB 改为 QTAB/2 即可。不重复贴代码。

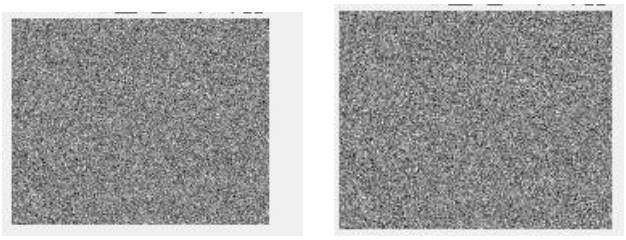


肉眼依然很难分辨，但 PSNR 变为了 34.2067 有所上升，说明失真程度变小。

压缩比变为 161280/(34164+2423)=4.4081，有所下降。

13.

只需将 load 的文件换为 snow.mat，将图像变量改为 snow 即可。不重复贴代码。



肉眼仍然难以分辨。

PSNR = 22.9244

压缩比为 128*160*8 / (43546+1456) = 3.6407

因为使用了原 QTAB 步长，与第 11 题相比，压缩比下降，图像质量也下降了。

这是因为雪花图形的像素与像素之间变化非常剧烈，因此 DC 分量较小，AC 分量较大。AC 编码时，大部分 AC 系数非零，因此压缩编码效果差；而 JPEG 编码主要通过舍弃高频 AC 分量进行压缩，导致压缩比较小，且失真较严重。

三.

1. images3_1.m

直接将信息“xinhaoyuxitonghenyouqu”转换成二进制编码，依次替换每个像素亮度的最低位即可。下图是原图与替换后的图的对比，肉眼几乎难以分辨。

```
msg_decoded =

  22×1 char 数组

    '`'
    '9'
    'L'
    '←'
    ' '
    ' '
    '6'
    ' '
    '-'
    ' '
    'C'
    'F'
    ' '
    ' '
    'F'
    'C'
    ' '
    ' '
    'z'
    's'
    'G'
    '#'
    'A'
```

但完全无法抗 JPEG 编码。

2. images3_2a.m, images3_2b.m, images

(a). 将信息编码后，嵌入 zigzag 扫描之前、量化之后的 DCT 系数中，然后按照 2_11 的代码进行编解码，最后成功恢复出被嵌入的信息。

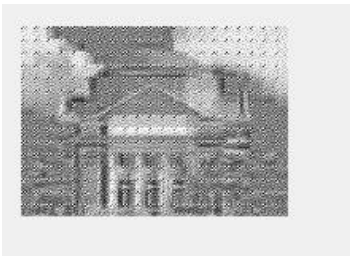
```
msg_decoded =

  22×1 char 数组

    'x'
    'i'
    'n'
    'h'
    'a'
    'o'
    'y'
    'u'
    'x'
    'i'
    't'
    'o'
    'n'
    'g'
    'h'
    'e'
    'n'
    'y'
    'o'
    'u'
    'q'
    'u'


PSNR =

    3.9828
```



但可以肉眼观察看出图片失真严重，PSNR 仅为 3.9828，隐蔽性非常差。
压缩比 161280/(54492+1649)=2.8728，压缩比也很低。

(b) 仅替换小部分 DCT 系数，重新进行编解码，成功恢复出被嵌入的信息。

```
>> images3_2b

msg_decoded =

  22×1 char 数组

    'x'
    'i'
    'n'
    'h'
    'a'
    'o'
    'y'
    'u'
    'x'
    'i'
    't'
    'o'
    'n'
    'g'
    'h'
    'e'
    'n'
    'y'
    'o'
    'u'
    'q'
    'u'


PSNR =

    4.6891
```

PSNR 也有所上升，但明显可以看出图片左上角出现失真。这种方法隐蔽性一般。
压缩比 161280/(23419+2053)=6.3317，压缩比与原图相近了。

(c) 在 DCT 系数的最后一个非零位后添加一位编码；若最后一位是非零位则直接替换。这种方法也成功恢复，且图像肉眼难以看出差别，隐蔽性非常好。
压缩比 161280/(24017+2054)=6.1862，略有下降但仍然很好。

```
>> images3_2c

msg_decoded =

  22×1 char 数组

    'x'
    'i'
    'n'
    'h'
    'a'
    'o'
    'y'
    'u'
    'x'
    'i'
    't'
    'o'
    'n'
    'g'
    'h'
    'e'
    'n'
    'y'
    'o'
    'u'
    'q'
    'u'
```

四.

1. images4_1.m

依次读入每张图片并操作即可。由于 L 不为 8，需要将图片的色域量化到 2^L 后再操作。

a. 不需要。我们只需要统计每种颜色在图片中的出现概率即可，与图片大小无关。

b. L 每增大 1，v 的长度增大 8 倍。

2. images4_2_pre.m, images4_2.m

按照实验指导书上的说明，先运行 images4_1.m，预训练基准向量 v，并将其与 L 保存到 v.mat 中。由于程序运行需要花费大量时间，为了方便调试，减小时间复杂度，我在 images4_2_pre.m 对图像进行预处理。注意到耗时最长的部分是重复统计待检测框内的颜色数量，我预处理了整张图片颜色数的二维前缀和，保存在 pres 矩阵中。然后在 images4_2.m 文件里只需读取该数组，就可以在常数时间内获取任意框内的 u 向量了。声明非原创内容，在去除重复人脸识别框的过程中，参考了网上的非极大值抑制算法及代码，其余部分均为独立完成。

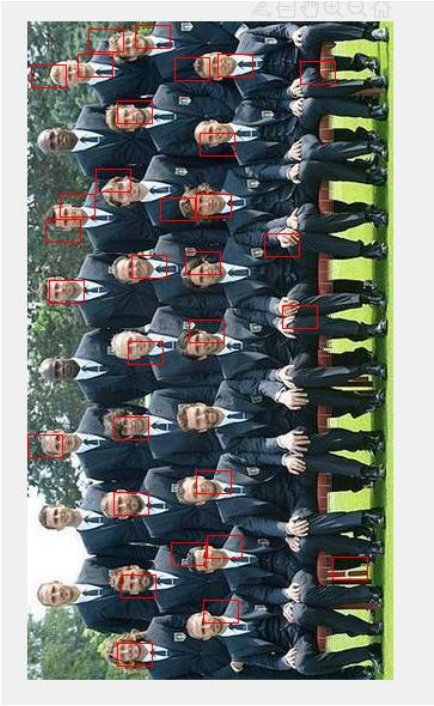
下面三张图片分别为 L=3、L=4、L=5 的识别结果。可以看出，随着 L 的增大，识别效果也变好了。

可以看出，白人和黑人因为肤色问题无法识别，其他人脸基本上都被成功检测。另外还

有一些非人脸的部分被检测到，猜测是颜色压缩后这些色块与人脸色块相近，因此被错误检测了；还有一些人脸被多个框框住，这是因为两个框没有重叠，因此 NMS 算法无法消除，这也说明该算法精度还不够，只能粗略检测人脸。猜想加大训练集数量，或提高 L 的值会有所帮助。

3.

（a）只需将 images4_2_pre.m 和 images4_2.m 中添加一行 img0 = rot90(img0);，并将 images4_2.m 中人脸检测框大小的 height 和 width 调转即可。



使用了 L=3 的参数。可以看出，检测结果与上一题的第一张图还是有区别的，不如旋转前效果好，但仍然不错。

（b）拉伸

在 images4_2_pre.m 与 image4_2.m 中添加代码 `img0 = imresize(img0, [612 550]);` 同时修改 image4_2.m 中 height 为原来的两倍。运行后发现效果不好，重新调整识别阈值后，得到了较好的结果。

可以看出仍有较好的识别效果，但与原图有一些差距。猜测仍然是参数设置的原因。

（c）调整颜色

在 images4_2_pre.m 与 image4_2.m 中添加代码 `img0 = imadjust(img0, [.2 .3 0; .7 .8 1],[]);` 得到下面的识别结果

可以发现识别效果明显差了许多，说明这个算法确实是颜色敏感的。

4. 我认为应该选取肤色与训练样本相近的（黄皮肤），背景色与皮肤颜色相差较大的，且其他皮肤裸露较少的图片。由于这是基于颜色检测的算法，任何相近的颜色都可能对监测效果造成干扰。