

具有自适应的 DCT 隐写算法实现与性能分析 (草稿版)

1. Design Detail

(1) 编写 genNeedImg.py 用于生成做本次实验所需的灰度图/二值图:

```
genNeedImg.py > ...
1   from cv2 import cv2 as cv
2   import sys
3
4   def genNeedImg(imgPath,size=None,flag='binary'):
5       """
6           用于生成指定大小的灰度图或二值图, imgPath为图像路径
7           size为tuple类型, 用于指定生成图像的尺寸, 如: (512,512), 默认为None表示输出原图像尺寸
8           flag为标志转换类型, 默认为binary, 可选的值为binary或gray
9       """
10      imgRow = cv.imread(imgPath)
11      if size != None: # 调整图像尺寸
12          imgRow= cv.resize(imgRow,size)
13      imgGray = cv.cvtColor(imgRow,cv.COLOR_RGB2GRAY) # 转换颜色空间为灰度
14      imgName = imgPath[9:].split('.')[0] # 获取图像原始名称
15      if flag == 'gray': # 生成灰度图
16          cv.imwrite('./images/{}_gray.bmp'.format(imgName),imgGray)
17          print('Gray image generated!')
18      else: # 生成二值图
19          ret, imgBinary = cv.threshold(imgGray,127,255,cv.THRESH_BINARY)
20          prop = int(size[0]*size[1]/(512*512)*100) # 以载体图像为512x512, 算生成的水印大小占载体图的百分比
21          cv.imwrite('./images/{}_binary{}.bmp'.format(imgName,prop),imgBinary)
22          print('Binary image generated!')
23          print('threshold:{} '.format(ret)) # 输出转换与之
24
25      if __name__ == "__main__":
26          imgName = sys.argv[1]
27          size =[int(sys.argv[2]),int(sys.argv[3])]
28          flag = sys.argv[4]
29          genNeedImg(imgName,tuple(size),flag=flag)
```

这是我编写的代码, 详细介绍看里面的注释

(2) 测试 genNeedImg.py, 并生成所需要的灰度图作为载体图像, 生成二值图作为嵌入的水印:



lena.bmp



dog.jpg

左边的 lena.bmp 已经是灰度图了, 直接拿它用作载体图像

右边的 dog.jpg 用于生成嵌入的二值水印图

```
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [21:09:40]
$ python3 genNeedImg.py ./images/dog.jpg 150 150 gray
Gray image generated!

# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [21:17:57]
$
```

使用 genNeedImg.py 生成 dog 的灰度图，大小为 150x150



dog_gray.bmp

此图片当作水印，嵌入 lena.bmp 中

由于 DCT 是分块嵌入，我设计的 DCT 算法是每 8x8 的分块中，

嵌入对角线 8 个水印像素信息

因此，针对 512x512 大小的 lena.bmp 而言

其总过可以嵌入的量为 $\frac{512 \times 512}{8 \times 8} \times 8 = 32768$ 个像素信息

而 150x150 的 dog_gray.bmp 其像素量为 22500

则其嵌入量为 $\frac{22500}{32768} \approx 0.6866 \approx 70\%$

为了对比不同的嵌入量，我在生成一张图，并将 dog_gray 重命名为 dog_gray70

```
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [21:32:32]
$ python3 genNeedImg.py ./images/dog.jpg 100 100 gray
Gray image generated!
```



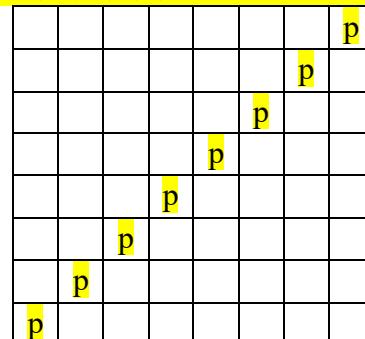
dog_gray30.bmp

同样的，计算出这个图的嵌入量约为 30%，因此命名为 dog_gray30.bmp

(3) 编写 DCT 嵌入代码 DCT_embed.py:

首先，需要阐明的是，我设计的 DCT 算法是在每 8x8 的分块中，

嵌入对角线 8 个水印像素信息，也就是在大致中频区域



格子中的 p 表示嵌入的信息

以下是嵌入算法的 python 代码：

```

from cv2 import cv2 as cv
import numpy as np
import matplotlib
import random,json

# 每个block的嵌入区，选最中间的中频8个位置
embedZone = [(0,7),(1,6),(2,5),(3,4),(4,3),(5,2),(6,1),(7,0)]

def resize2octa(imgPath):
    """
    为了之后能够整划分8x8矩阵，此函数将图像resize成8x8的倍数
    """
    imgRaw = cv.imread(imgPath, cv.IMREAD_UNCHANGED)
    rowSize = imgRaw.shape[0]
    columnSize = imgRaw.shape[1]
    # 要变成一个方阵
    adjust = rowSize if rowSize > columnSize else columnSize
    if adjust%8: # 等价于adjust%8 != 0
        adjust = (1+adjust//8)*8
    size = (adjust,adjust)
    imgResized = cv.resize(imgRaw, size)
    return imgResized

def divideBlock(img):
    """
    将图像分割成8x8的块，返回一个list存储了所有的块信息
    """
    scale = img.shape[0] # resize过 rowSize 和 columnSize 一样，这里用scale表示
    columnSliceList = [ [column[i*8:(i+1)*8] for i in range(scale//8)] for column in img]
    blockList = []
    for i in range(scale//8):
        rowBlockList = columnSliceList[8*i:8*(i+1)]
        for x in range(scale//8):
            blockList.append([rowBlockList[y][x] for y in range(8)])
    return blockList

```

载体图尺寸规范函数、分块函数

```

def dctBlock(blockList):
    """
    对每一个块进行dct变换
    """
    # dct计算的矩阵是一个二维矩阵，所以直接把三维图像塞进来的话容易报错如上，
    # 如果要处理三维图像（三通道）可以把rgb三个维度一个一个丢进来dct，最后合成一个三维矩阵保存成彩色图像。
    # 本实验不用考虑以上这点，灰度图本来就是二维的
    # 同时，需要注意的是，要用np.float32把矩阵转换成32位浮点精度，这才是dct能处理的精度。所以必不可少。
    return [cv.dct(np.float32(block)) for block in blockList]

def idctBlock(dctBlockList):
    """
    对每一个块进行逆dct变换
    """
    return [cv.idct(block) for block in dctBlockList]

```

块的 dct 变换函数和逆变换函数

```

def mergeBlock(blockList): # 注意此函数传入的必须是list不能是np的array, 要先把array转成list
    scale = int(pow(len(blockList), 0.5))*8
    imgMatrix = []
    for i in range(scale//8):
        for y in range(8):
            column = []
            for x in range(scale//8):
                column += blockList[(scale//8)*i+x][y]
            imgMatrix.append(column)
    return imgMatrix

# https://blog.csdn.net/missingui1314/article/details/8677703
def getEmbedPixel(imgEmbed, factor=0.05): # 缩放因子默认为0.05
    ...
    此函数用于将嵌入的水印图片的像素值, 使用加法准则公式调节成嵌入的值,
    默认的缩放因子factor=0.05, 关于此部分的介绍, 详见实验报告
    ...
    rowScale = imgEmbed.shape[0]
    columnScale = imgEmbed.shape[1]
    pixelStream = []
    for i in range(rowScale):
        for j in range(columnScale): # 加法准则
            pixelStream.append(factor*imgEmbed.item(i,j))
    if len(pixelStream)%8 != 0:
        zeroAmount = 8 - len(pixelStream)%8
        for i in range(zeroAmount):
            pixelStream.append(0)
    return pixelStream

```

合并块为图片的函数、嵌入水印像素值操作函数

关于 getEmbedPixel 函数的具体解释:

由于考虑到直接将要嵌入的灰度图像素值嵌入载体图像的 DCT 系数矩阵中, 可能效果会不好, 因为考虑到灰度图的像素值在 0~255 之间, 一般都是很大的整数, 而 DCT 系数矩阵一般是一个浮点矩阵, 数值不会特别大

因此我参考了 Cox 等人于 1995 年提出了基于图像全局变换的水印方法, 称之为扩频法。这也是目前大部分变换域水印算法中所用到的技术。它将满足正态分布的伪随机序列加入到图像的 DCT 变换后视觉最重要系数中, 利用了序列扩频技术 (SS) 和人类视觉特性 (HVS)。算法原理为先选定视觉重要系数, 再进行修改, 最常用的嵌入规则如下:

$$V1 = V + \alpha \cdot Wi \text{ (加法准则)}$$

$$V1 = V \cdot (1 + \alpha \cdot Wi) \text{ (乘法准则)}$$

其中 V 、 $V1$ 分别是修改前和修改后的频域系数, α 是缩放因子, 是第 Wi 个信息位水印。此处, 我设计的 getEmbedPixel 函数使用了加法准则, 具体而言:

要嵌入的信息 = 载体DCT系数 + factor × 水印像素值

其中, factor 就是缩放因子

```

def dctEmbedding(imgCoverPath, imgEmbedPath, embedZone, factor=0.05):
    """
    DCT 嵌入主函数
    """

    imgCover = cv.imread(imgCoverPath, cv.IMREAD_GRAYSCALE) # 以灰度图方式读取载体图像
    imgEmbed = cv.imread(imgEmbedPath, cv.IMREAD_GRAYSCALE) # 以灰度图方式读取嵌入图像
    pixelStream = getEmbedPixel(imgEmbed, factor=factor)
    blockList = divideBlock(imgCover)
    dctBlockList = dctBlock(blockList)
    embedBlock = getRandEmbedBlock(blockList, pixelStream)
    with open('./KeyFile/embedblock.json', 'w') as fp:
        json.dump(embedBlock, fp)
    rowVlist = []
    pos = 0
    for i in embedBlock:
        pixelList = pixelStream[8*pos:(pos+1)*8]
        pos+=1
        for zone, pixel in zip(embedZone, pixelList):
            # rowVlist.append(float(dctBlockList[i][zone[0]][zone[1]]))
            rowVlist.append(round(float(dctBlockList[i][zone[0]][zone[1]]), 7))
            dctBlockList[i][zone[0]][zone[1]] += pixel
    with open('./KeyFile/Vlist.json', 'w') as fp:
        json.dump(rowVlist, fp)
    idctBlockList = np.uint8(np.rint(idctBlock(dctBlockList)))
    imgStego = mergeBlock(idctBlockList.tolist())
    imgStego = np.array(imgStego, dtype=np.uint8)
    cv.imwrite('./img/imgStego.bmp', imgStego)
    print('Done!')

```

DCT 嵌入主处理函数

(4) 测试 DCT_embed.py，并生成嵌入率 30% 和 70% 的 stego 图像：

```

# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [10:39:40] C:1
$ python3 DCT_embed.py
cover path:./images/lena.bmp
embed path:./images/dog_gray30.bmp
factor:0.06
Done!
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [10:40:46]
$ []

缩放因子 0.06, 嵌入率 30%

```



```

# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [10:40:09]
$ python3 DCT_embed.py
cover path:./images/lena.bmp
embed path:./images/dog_gray70.bmp
factor:0.06
Done!
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [10:40:46]
$ []

缩放因子 0.06, 嵌入率 70%

```

缩放因子 0.06，生成嵌入率为 30% 和 70% 的 stego 图像



可以看到生成了对应的 stego 图像和密钥文件（用于提取时解密水印）

生成密钥文件是因为我在嵌入时对嵌入信息做了随机加密

(5) 编写 DCT 提取代码 DCT_extract.py:

```
from cv2 import cv2 as cv
import numpy as np
import matplotlib
import random,json

# 每个block的嵌入区，选最中间的中频8个位置
embedZone = [(0,7),(1,6),(2,5),(3,4),(4,3),(5,2),(6,1),(7,0)]

def divideBlock(img):
    '''将图像分割成8x8的块，返回一个list存储了所有的块信息'''
    scale = img.shape[0] # resize过 rowSize 和 columnSize 一样，这里用scale表示
    columnSliceList = [ [column[i*8:(i+1)*8] for i in range(scale//8)] for column in img]
    blockList = []
    for i in range(scale//8):
        rowBlockList = columnSliceList[8*i:8*(i+1)]
        for x in range(scale//8):
            blockList.append([ rowBlockList[y][x] for y in range(8)])
    return blockList

def dctBlock(blockList):
    '''对每一个块进行dct变换'''
    # dct计算的矩阵是一个二维矩阵，所以直接把三维图像塞进来的话容易报错如上,
    # 如果要处理三维图像（三通道）可以把rgb三个维度一个一个丢进来dct，最后合成一个三维矩阵保存成彩色图像。
    # 本实验不用考虑以上这点，灰度图本来就是二维的
    # 同时，需要注意的是，要用np.float32把矩阵转换成32位浮点精度，这才是dct能处理的精度。所以必不可少。
    return [cv.dct(np.float32(block)) for block in blockList]

def idctBlock(dctBlockList):
    ...
    对每一个块进行逆dct变换
    ...
    return [cv.idct(block) for block in dctBlockList]
```

分块函数、块的 dct 变换函数和逆变换函数

```
def mergeBlock(blockList): # 注意此函数传入的必须是list不能是np.array, 要先把array转成list
    scale = int(pow(len(blockList),0.5))*8
    imgMatrix = []
    for i in range(scale//8):
        for y in range(8):
            column = []
            for x in range(scale//8):
                column += blockList[(scale//8)*i+x][y]
            imgMatrix.append(column)
    return imgMatrix

def pixelStream2Img(pixelStream):
    '''将像素流转换为图像矩阵'''
    scale = int(pow(len(pixelStream),0.5))
    return [ pixelStream[i*scale:(i+1)*scale] for i in range(scale)]
```

块合并函数和像素流转换函数

```
def dctExtracting(imgStegoPath,embedBlock,Vlist,embedZone,factor=0.05):
    '''DCT提取主函数'''
    imgStego = cv.imread(imgStegoPath, cv.IMREAD_UNCHANGED)
    blockList = divideBlock(imgStego)
    dctBlockList = dctBlock(blockList)
    pos = 0
    pixelStream = []
    for i in embedBlock:
        vlistInUsing = Vlist[pos*8:(pos+1)*8]
        pos+=1 # 这里之前没写。。。
        for zone,v in zip(embedZone,vlistInUsing):
            tmp = dctBlockList[i][zone[0]][zone[1]]
            p = int((tmp - v)/factor)
            pixelStream.append(p)
    imgExtract = pixelStream2Img(pixelStream)
    # print(imgExtract)
    imgExtract = np.array(imgExtract,dtype=np.uint8)
    cv.imwrite('./img/imgExtract.bmp',imgExtract)
```

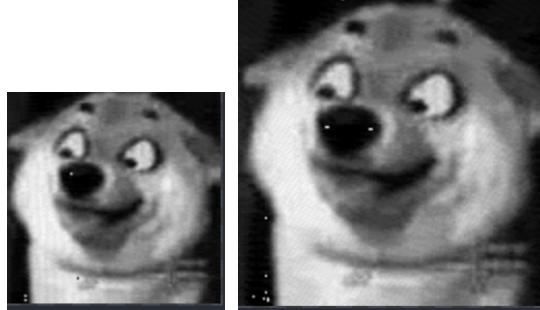
DCT 提取主处理函数

(6) 测试 DCT_extract.py，并测试对嵌入率为 30% 和 70% 的 stego 图像的水印提取：

```
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:07:41]
$ python3 DCT_extract.py
stego path: ./img/stego30.bmp
embedblock file path: ./KeyFile/embedblock30.json
Vlist file path: ./KeyFile/Vlist30.json
factor:0.06
Extracting Done!
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:08:24]
$ 
缩放因子 0.06, 嵌入率 30% 的水印提取

# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:05:53]
$ python3 DCT_extract.py
stego path: ./img/stego70.bmp
embedblock file path: ./KeyFile/embedblock70.json
Vlist file path: ./KeyFile/Vlist70.json
factor:0.06
Extracting Done!
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:09:17]
$ 
缩放因子 0.06, 嵌入率 70% 的水印提取
```

缩放因子 0.06，对嵌入率为 30% 和 70% 的 stego 图像的水印进行提取



从左到右依次为，从嵌入率 30% 和 70% 的 stego 图像提取出的水印

(7) 关于 PSNR、错误率和鲁棒性见下一部分

2. Experimental Results Analysis

1) Imperceptibility

Compare the original image with the watermarked image by PSNR.

Analyze the results with different images and different payload/capacity.

2) False Rate

Compare the original watermark with the extracted watermark to calculate the error false.

3) Robustness

Apply some attacks (image noise, image compression, content modification et al.) to the watermarked image, then extract the watermark and compare with the original one to calculate the error false.

(1) 对于不同嵌入率情况下对嵌入后图像，进行不可感性分析（PSNR）

```
import numpy as np
from cv2 import cv2 as cv
import math

def psnr(img_1, img_2):
    mse = np.mean((img_1 / 1.0 - img_2 / 1.0) ** 2)
    if mse < 1.0e-10:
        return 100
    return 10 * math.log10(255.0 ** 2 / mse)

if __name__ == "__main__":
    imgCoverPath = input('cover image path:')
    imgStegoPath = input('stego image path:')
    imgCover = cv.imread(imgCoverPath, cv.IMREAD_UNCHANGED)
    imgStego = cv.imread(imgStegoPath, cv.IMREAD_UNCHANGED)
    print('「载体图」和「嵌入水印图」对比算出的psnr: {}'.format(psnr(imgCover, imgStego)))
```

编写 psnr.py 用于计算 psnr

```

问题 输出 调试控制台 终端 1: zsh, zsh + □ ■

# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:20:15]
$ python3 psnr.py
cover image path:./images/lena.bmp
stego image path:./img/stego30.bmp
「载体图」和「嵌入水印图」对比算出的psnr: 44.51911264713804
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:21:06]
$ 

缩放因子0.06, 嵌入率30%计算出来的psnr

# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:20:17]
$ python3 psnr.py
cover image path:./images/lena.bmp
stego image path:./img/stego70.bmp
「载体图」和「嵌入水印图」对比算出的psnr: 40.99980128802507
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:21:38]
$ 

缩放因子0.06, 嵌入率70%计算出来的psnr

```

从左到右分别是嵌入率 30% 和 70% 的 psnr 值，

依次为 44.5191、40.9998，均在 40 以上，

同时也可以看到，随着嵌入率提升，PSNR 值会随之下降

PSNR 高于 40dB 说明图像质量极好（即非常接近原始图像），

在 30—40dB 通常表示图像质量是好的（即失真可以察觉但可以接受），

在 20—30dB 说明图像质量差；而 PSNR 低于 20dB 图像不可接受

因此，我写的 dct 嵌入算法嵌入效果极其出色

(考虑到我一个块嵌入了 8 个信息，并且嵌入在中频区，因此效果十分出色了)

嵌入后的 stego 图像的不可感知性极为出色

(2) 对提取出来的水印对比原始水印，进行错误率分析

我设计的错误率计算公式：

$$falseRate = \frac{\sum_{i=0}^n |p_i - p'_i| / 256}{n}$$

其中 n 为图片总像素个数， p_i 是原始水印第 i 个像素的像素值，

p'_i 为提取出来的水印的第 i 个像素的像素值

```

import numpy as np
from cv2 import cv2 as cv
import math

def falseRate(img1,img2):
    img1_float = np.array(img1,dtype=np.float)
    img2_float = np.array(img2,dtype=np.float)
    return np.mean(abs(img1_float-img2_float))/256

if __name__ == "__main__":
    imgEmbedPath = input('embed img path(original watermark):')
    imgExtractPath = input('extract img path(extracted watermark):')
    imgEmbed = cv.imread(imgEmbedPath, cv.IMREAD_UNCHANGED)
    imgExtract = cv.imread(imgExtractPath, cv.IMREAD_UNCHANGED)
    print('『提取出来的水印』的错误率: {}'.format(falseRate(imgEmbed, imgExtract)))

```

依据该公式，编写 falseRate.py 用于计算错误率

```

# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:27:52]
$ python3 falseRate.py
embed img path(original watermark):./images/dog_gray30.bmp
extract img path(extracted watermark):./img/imgExtract30.bmp
「提取出来的水印」的错误率: 0.01853203125
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:28:45]
$ 

缩放因子0.06, 嵌入率30%, 提取出来水印的错误率

```

```

# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:27:59]
$ python3 falseRate.py
embed img path(original watermark):./images/dog_gray70.bmp
extract img path(extracted watermark):./img/imgExtract70.bmp
「提取出来的水印」的错误率: 0.01820642361111112
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:29:38]
$ 

缩放因子0.06, 嵌入率30%, 提取出来水印的错误率

```

从左到右分别是在缩放因子为 0.06，嵌入率 30% 和 70% 计算出来的 falseRate 值

可以看到依次为 1.853%、1.821%，提取出来的水印质量还是极其不错的

(注：由于错误率公式是我自己定义的，这里的百分比不是直观的效果，

我测出来，当错误率达到 10% 时，提取出来的水印就基本失真了)

因此设错误率阈值为 10%，大于则提取的效果不好（阈值的来源请看额外创新部分）

(3) 对图像做加噪攻击，分析 LSB 算法的鲁棒性

```
# noise.py > ...
1   from cv2 import cv2 as cv
2   import numpy as np
3   import random
4
5   def sp_noise(image, prob):
6       """
7           添加椒盐噪声
8           prob: 噪声比例
9       """
10      output = np.zeros(image.shape, np.uint8)
11      thres = 1 - prob
12      for i in range(image.shape[0]):
13          for j in range(image.shape[1]):
14              rdn = random.random()
15              if rdn < prob:
16                  output[i][j] = 0
17              elif rdn > thres:
18                  output[i][j] = 255
19              else:
20                  output[i][j] = image[i][j]
21
22      return output
```

编写 noise.py 用于给图像进行椒盐加噪

```
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:38:51] C:1
$ python3 noise.py
image path:./img/stego30.bmp
Noised image generated!
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:39:11]
$
```

对缩放因子 0.06，嵌入率 30% 的 stego 加噪

```
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:39:20]
$ python3 noise.py
image path:./img/stego70.bmp
Noised image generated!
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:39:32]
$
```

对缩放因子 0.06，嵌入率 70% 的 stego 加噪

对之前生成的 stego 进行加噪



stego30_noised.bmp

stego70_noised.bmp

从左到右分别是对嵌入率 30% 和 70% 的 stego 进行椒盐加噪生成的图片

```
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:45:36] C:127
$ python3 DCT_extract.py
stego path:./img/stego30_noised.bmp
embedblock file path:./Keyfile/embedblock30.json
vlist file path:./KeyFile/Vlist30.json
factor:0.06
Extracting Done!
```

rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:46:32]
\$

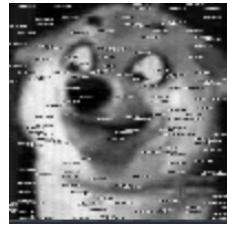
对嵌入率 30% 加噪后的水印提取

```
# rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:47:04]
$ python3 DCT_extract.py
stego path:./img/stego70_noised.bmp
embedblock file path:./KeyFile/embedblock70.json
vlist file path:./KeyFile/Vlist70.json
factor:0.06
Extracting Done!
```

rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:47:57]
\$

对嵌入率 70% 加噪后的水印提取

分别对加噪的嵌入率 30% 和 70% 的 stego 进行水印提取



noisedExtract30.bmp



noisedExtract70.bmp

以上是加噪后提取出的水印，可以看到有许多椒盐噪点

| | |
|--|---|
| # rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:52:42] \$ python3 falseRate.py embed img path(original watermark):./images/dog_gray30.bmp extract img path(extracted watermark):./img/noisedExtract30.bmp 「提取出来的水印」的错误率：0.059889625 | # rgmax @ RGMac in ~/Desktop/信息隐藏/RG_DCT [11:51:59] \$ python3 falseRate.py embed img path(original watermark):./images/dog_gray70.bmp extract img path(extracted watermark):./img/noisedExtract70.bmp 「提取出来的水印」的错误率：0.0573506944444445 |
| 对加噪后嵌入率 30% 提取出的水印进行错误率分析 | 对加噪后嵌入率 70% 提取出的水印进行错误率分析 |

使用 falseRate.py 计算错误率

从左到右依次是 5.9889%、5.7351%

可以看到受到椒盐加噪攻击之后，提取出的水印错误率提高了

但是，受到攻击后提取出的水印错误率并不是很高，没有超出错误率阈值

说明我的 LSB 算法具有一定的鲁棒性

不过可能和椒盐系数有关，下面部分会有针对 70% 嵌入的图进行加噪系数分析

(4) 关于我设计的 DCT 算法中，引入缩放因子 factor 的分析（额外的创新点）

此部份的分析研究是针对 70% 嵌入率进行的，其他嵌入了结论应该类似

| factor | psnr | false rate | false rate*100 |
|--------|----------|------------|--------------------------------|
| 1.000 | 16.68709 | 0.2052722 | 20.52722222 |
| 0.500 | 20.55290 | 0.0820757 | 8.207569444 |
| 0.250 | 27.57784 | 0.0186399 | 1.863993056 |
| 0.200 | 29.95717 | 0.0109160 | 1.091597222 |
| 0.175 | 31.59008 | 0.0077813 | 0.778125 水印质量极好，但是嵌入的图片质量不佳 |
| 0.150 | 32.99889 | 0.0082375 | 0.82375 |
| 0.125 | 34.63253 | 0.0094024 | 0.940243056 |
| 0.100 | 36.56894 | 0.0115611 | 1.156111111 |
| 0.090 | 37.48520 | 0.0126010 | 1.260104167 |
| 0.080 | 38.50463 | 0.0140851 | 1.408506944 |
| 0.070 | 39.66602 | 0.0156352 | 1.563524306 |
| 0.060 | 40.99980 | 0.0182049 | 1.820486111 嵌入图像质量极好，提取的水印质量很好 |
| 0.050 | 42.56097 | 0.0217481 | 2.174809028 |
| 0.040 | 44.51241 | 0.0267083 | 2.670833333 |
| 0.030 | 46.93299 | 0.0328010 | 3.280104167 |
| 0.020 | 50.42697 | 0.0485585 | 4.855850694 水印质量较好 |
| 0.010 | 56.14620 | 0.0815455 | 8.154548611 |

factor 越大被嵌图像质量越好，但水印质量越差；事实上，低于 10% 的错误率，提取的水印都清晰可辨

此处，调节缩放因子 factor 的取值，得到了 factor 对 psnr 和 false rate 的影响关系：

factor 越大被嵌图像质量越好，但水印质量越差；

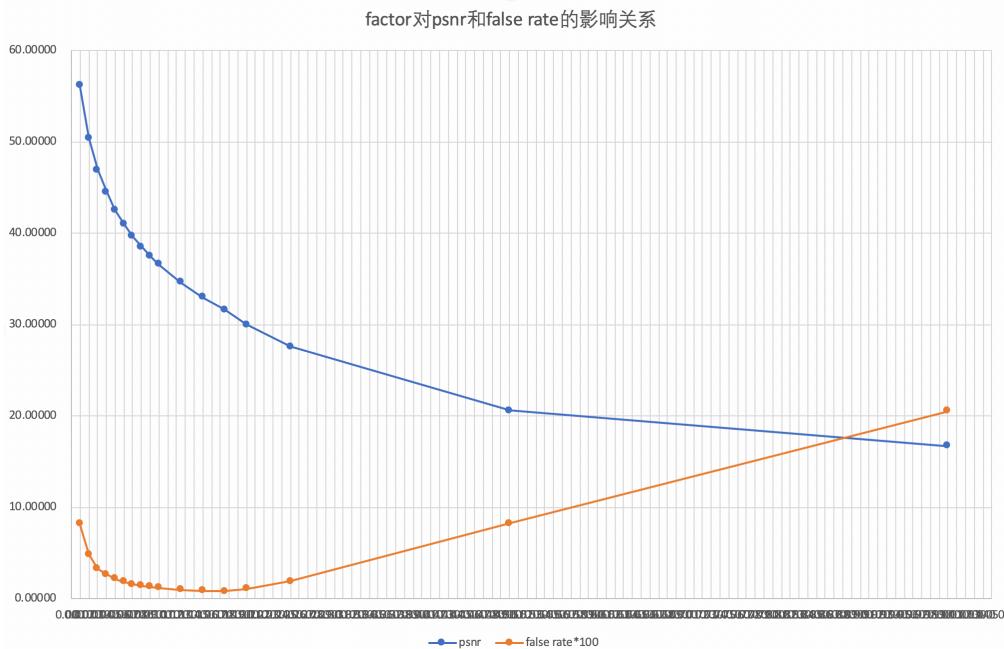
事实上，低于 10% 的错误率，提取的水印都清晰可辨

总体而言，为了达到不可感知性和鲁棒性的平衡，使两者均处于较好的水平

针对 dog_gray.bmp 这张图，选取的最佳 factor 为 0.06

经过测试，针对不同的水印图片，其最佳的 factor 也有所不同

可以说，我设计的 DCT 嵌入算法，还实现了针对不同嵌入水印的自适应



这是对上表整理得到的 factor 对 psnr 和 false rate 的影响关系曲线

关于椒盐加噪系数 prob 对提取水印错误率的影响

| 加噪（椒盐）分析 | | |
|----------|------------|--------------------|
| prob | false rate | 提取的水印描述 |
| 0.001 | 0.0614951 | 轮廓清晰，细节清晰，存在极少数噪点 |
| 0.002 | 0.0969894 | 轮廓清晰，细节较为清晰，噪点稍微增多 |
| 0.003 | 0.1248385 | 轮廓清晰，细节略微丢失，存在大量噪点 |
| 0.004 | 0.1573474 | 轮廓略微模糊，细节丢失较多，噪点极多 |
| 0.005 | 0.1741882 | 较为模糊的轮廓，细节可以辨识 |
| 0.006 | 0.1950806 | 模糊的轮廓，细节基本可以辨识 |
| 0.007 | 0.2133950 | 非常模糊的轮廓，细节勉强可以辨识 |
| 0.008 | 0.2284424 | 极其模糊的轮廓，细节丢失严重 |
| 0.009 | 0.2444998 | 轮廓勉强辨识，细节难以辨识 |
| 0.01 | 0.2473035 | 大致只能看出个轮廓 |

经过不断调节椒盐加噪系数 prob，并计算提取出的水印错误率，

以及肉眼评价水印质量，可以得到上表的结论

最终，取错误率 10%作为评价水印提取质量的阈值

3. Conclusion

Highlight your own contribution in the group.

Summarize your own understanding and interpretation of the results.

通过本次试验，我巩固复习了 DCT 隐写算法的整体流程，同时使用 python 而不是 matlab 实现了自己的 DCT 算法。通过对自选取的图片，进行灰度图水印的嵌入和提取，测试了我设计的 DCT 算法的性能：

- 拥有极好的不可感知性，肉眼观察嵌入图与原图毫无差别，在嵌入率 30% 和 70% 的 psnr 值，依次为 44.5191、40.9998，均在 40 以上，同时也可以看到，随着嵌入率提升，PSNR 值会随之下降

- 在没有攻击情况下，缩放因子为 0.06，嵌入率 30% 和 70% 提取出来的水印的错误率值依次为 1.853%、1.821%，提取出来的水印质量还是极其不错的
 (注：由于错误率公式是我自己定义的，这里的百分比不是直观的效果，我测出来，当错误率达到 10% 时，提取出来的水印就基本失真了)
 因此设错误率阈值为 10%，大于则提取的效果不好（阈值的来源请看额外创新部分）
- 在椒盐加噪系数 0.01（常用），缩放因子为 0.06 时，嵌入率 30% 会让 70% 提取出来的水印错误率依次是 5.9889%、5.7351%。可以看到受到椒盐加噪攻击之后，提取出的水印错误率提高了。但是，受到攻击后提取出的水印错误率并不是很高，没有超出错误率阈值 10%，说明我的 DCT 算法还是有很好的鲁棒性的

本次实验我的原创之处：

- 使用 python 而不是 matlab 实现，所有算法所有函数均为我亲自独立完成
- 自定义了错误率计算公式，并测试得出在该公式下的水印质量评价标准阈值（通过多次实验测试，定义错误率 10% 为提取出的水印好坏阈值），公式如下：

我设计的错误率计算公式：

$$falseRate = \frac{\sum_{i=0}^n |p_i - p'_i| / 256}{n}$$

其中 n 为图片总像素个数， p_i 是原始水印第 i 个像素的像素值， p'_i 为提取出来的水印的第 i 个像素的像素值

- 对于 DCT 嵌入的时候，我对水印做了处理（可以理解为加密），效果是使得嵌入的 01 均匀且随机分布
- 在嵌入的时候做了随机位置乱序嵌入，使得用位平面分析我的嵌入后的图根本分析不出来，藏得效果极好
- 同时，我设计的 DCT 在嵌入时，一个块嵌入了 8 个信息，并且嵌入在中频区，嵌入效果依旧十分出色了，嵌入后的 stego 图像的不可感知性极为出色，也具有较好的鲁棒性
- 最重要的是，由于考虑到直接将要嵌入的灰度图像素值嵌入载体图像的 DCT 系数矩阵中，可能效果会不好，因为考虑到灰度图的像素值在 0~255 之间，一般都是很大的整数，而 DCT 系数矩阵一般是一个浮点矩阵，数值不会特别大。因此我参考了 Cox 等人于 1995 年提出了基于图像全局变换的水印方法，称之为扩频法。这也是目前大部分变换域水印算法中所用到的技术。它将满足正态分布的伪随机序列加入到图像的 DCT 变换后视觉最重要系数中，利用了序列扩频技术(SS)和人类视觉特性(HVS)。
 算法原理为先选定视觉重要系数，再进行修改，最常用的嵌入规则如下：

$$V1 = V + \alpha \cdot Wi \text{ (加法准则)}$$

$$V1 = V \cdot (1 + \alpha \cdot Wi) \text{ (乘法准则)}$$

其中 V、V1 分别是修改前和修改后的频域系数， α 是缩放因子，是第 Wi 个信息位水印。此处，我设计的 getEmbedPixel 函数使用了加法准则，具体而言：

$$\text{要嵌入的信息} = \text{载体DCT系数} + factor \times \text{水印像素值}$$

其中，factor 就是缩放因子

- 通过对缩放因子的测试研究，我得出如下结论：factor 越大被嵌图像质量越好，但水印质量越差；事实上，低于 10% 的错误率，提取的水印都清晰可辨。总体而言，为了达到不可感知性和鲁棒性的平衡，使两者均处于较好的水平。针对 dog_gray.bmp 这张图，选取的最佳 factor 为 0.06。经过测试，针对不同的水印图片，其最佳的 factor 也有所不同。可以说，我设计的 DCT 嵌入算法，还实现了针对不同嵌入水印的自适应