实验4-有损图像压缩系统设计

1 实验目的

（1）熟悉传统图像有损压缩的基本原理和开发流程；

（2）了解和掌握变换编码方法，如DCT变换等对信号熵的影响；

（3）了解量化的原理，以及量化等级与信号失真程度之间的关系；

（4）掌握常见的图像压缩性能评价指标，如PSNR和SSIM。

2 实验环境

（1）硬件环境：PC；

（2）软件环境：Windows 10、python3.6

3 实验内容

3.1 离散余弦变换

离散余弦变换（DCT）能够以数据无关的方式去除输入信息之间的相关性，在图像压缩标准中得到了广泛应用。在图像压缩中，通常是对固定的的$M \times N$图像块做二维DCT变换，然后获得$M \times N$大小的DCT变换系数矩阵F。对应的公式定义如下:

$$F(u,v) = \frac{2C(u)C(v)}{\sqrt{MN}} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} B(u,v) f(i,j) \tag{1}$$

其中$i,u = 0,\ldots,M-1, j,v = 0,\ldots,N-1$。常数$C(u)$和$C(v)$由下式得出:

$$C(\xi) = \begin{cases} \frac{\sqrt{2}}{2}, & \xi = 0 \\ 1, & \xi \neq 0 \end{cases} \tag{2}$$

对应的二维DCT变换的基函数$B(u,v)$定义为:

$$B(u,v) = cos\frac{(2i+1)\cdot u\pi}{2M} \cdot cos\frac{(2j+1)\cdot v\pi}{2N} \tag{3}$$

需要注意的是，基函数$B(u,v)$是一个尺寸为$M \times N$的矩阵。

下面给出了python语言实现的 plotdct2base 代码，该代码块可完成指定基函数$B(u,v)$的定义和可视化。

```
# 每一个uv对应的dct基，本质是一个变换矩阵
def plotdct2base(u, v, M, N, flag):
    A = u*np.pi/(2*M)
    B = v*np.pi/(2*N)
    row = np.arange(M)
    col = np.arange(N)

    row = np.cos((2*row+1)*A)
    col = np.cos((2*col+1)*B)

    dct2base = np.matmul(row.reshape(M, 1), col.reshape(1, N))
```

```
    dct2baseplot = None
    if flag:
        dct2baseplot = np.kron(dct2base, np.ones((32, 32)))
    return dct2base, dct2baseplot
```

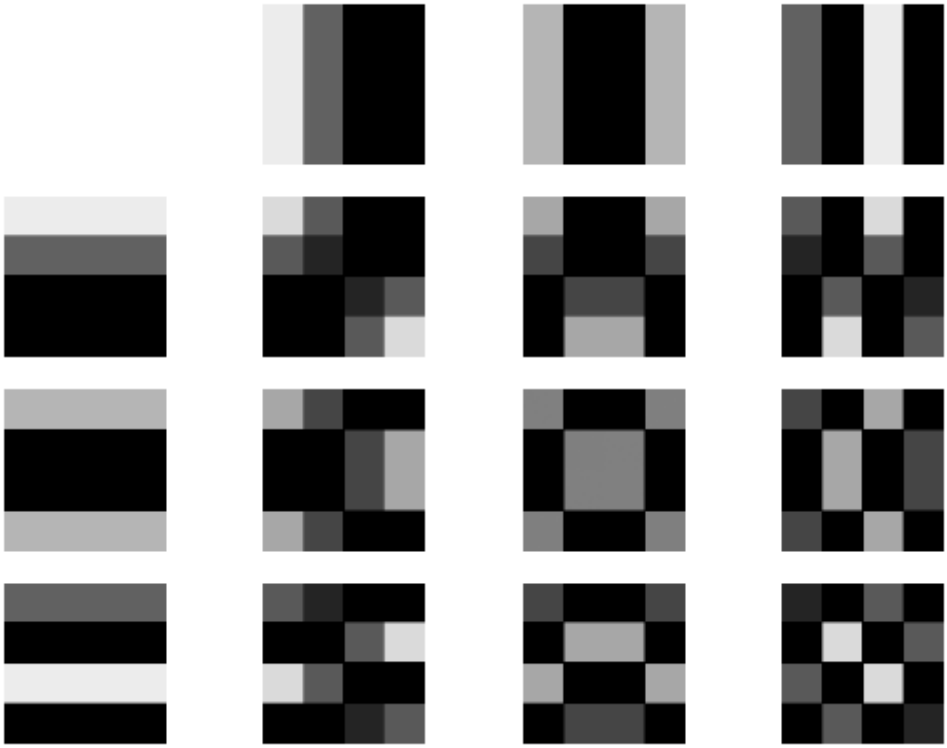
下面给出了python语言实现的 plotidct2base 代码，该代码块可完成指定逆变换中基函数$B(u,v)$的定义和可视化。

```
# 每一个uv对应的idct基，本质是一个变换矩阵
def plotidct2base(i, j, M, N, flag):
    C = np.ones((np.max([M, N]), 1))
    C[0] = np.sqrt(2)/2

    A = (2*i+1)*np.pi/(2*M)
    B = (2*j+1)*np.pi/(2*N)
    row = np.arange(M)
    col = np.arange(N)

    scale = 2*np.matmul(C.reshape(M, 1), C.reshape(1, N))/np.sqrt(M*N)

    row = np.cos(row*A)
    col = np.cos(col*B)
    idct2base = np.matmul(row.reshape(M, 1), col.reshape(1, N))*scale

    return idct2base
```

假设$M=N=4$,我们可通过执行下述的示例代码 Exp1 ， 获得相应的基函数的值，以及可视化的效果。

```
if __name__ == '__main__':
    M=4
    N=4
    flag=1
    plt.figure()
    for u in range(M):
        for v in range(N):
            dct2base,dct2baseplot = plotdct2base(u,v,M,N,flag)
            plt.subplot(M,N,u*N+v+1)
            plt.imshow(dct2baseplot,'gray',vmin=0,vmax=1) # 仅关系show
            plt.axis('off')
    plt.show()
```

以下为代码执行后的输出结果：



给定$M = N = 8$,对8×8大小的图像块，每次2D DCT/IDCT变换，需要做64次迭代计算。执行下 述的示例代码 Exp2 ， 不难看出DCT变换是可逆变换。

```
    data = np.random.randint(low=0,high=255,size=(8,8))

    M = 8
    N = 8
    flag = 0

    # 2D dct  # 复杂度M^2N^2
    C = np.ones((np.max([M, N]), 1))
    C[0] = np.sqrt(2)/2
    scale = 2*np.matmul(C.reshape(M, 1), C.reshape(1, N))/np.sqrt(M*N)

    st = time.time()
    dct_2D_coef = np.zeros((M, N))
    for u in range(M):
        for v in range(N):
            dct2base, _ = plotdct2base(u, v, M, N, flag)
            dct_2D_coef[u, v] = (data*dct2base).sum()  # 一个变换之后，能量集中在一点
    dct_2D_coef = dct_2D_coef * scale
    end = time.time()

    # 2D idct
    st1 = time.time()
    data_rec_2D = np.zeros((M, N))
    for i in range(M):
        for j in range(N):
            idct2base = plotidct2base(i, j, M, N, flag)
            data_rec_2D[i, j] = (dct_2D_coef*idct2base).sum()  # 一个逆变换，从所有
坐标的能量中提取恰当分量，来还原
    end1 = time.time()
```

以下为代码执行后的输出结果：

```
data=
 [[113 146 228 129  24  80 252  76]
 [ 46 123 240 185  14 247  35  54]
 [114 210  93 170  47   4  80 111]
 [130  49  92 231 249 183 138 116]
 [ 16 135 197 142  89 168   5  21]
 [ 35 115 100 124  94  53 200  80]
 [ 14 210 211 254  29 132  75 185]
 [ 42 129 222 113 221 154 213  97]]

The reconstructed version of original data for 2D DCT/IDCT Transform:
[[113. 146. 228. 129.  24.  80. 252.  76.]
 [ 46. 123. 240. 185.  14. 247.  35.  54.]
 [114. 210.  93. 170.  47.   4.  80. 111.]
 [130.  49.  92. 231. 249. 183. 138. 116.]
 [ 16. 135. 197. 142.  89. 168.   5.  21.]
 [ 35. 115. 100. 124.  94.  53. 200.  80.]
 [ 14. 210. 211. 254.  29. 132.  75. 185.]
 [ 42. 129. 222. 113. 221. 154. 213.  97.]]

Runtime is 0.00201583 s for 8 x 8 2D-DCT/IDCT Transform
```

如果利用前面获得二维DCT基函数对图像块进行DCT变换，需要大量重复的计算。由于二维基函数中关于u和v的两个分量相互独立，故二维DCT变换可以转化为两个一维DCT变换。该变换过程可表示为矩阵乘法：

$$F = TfT', \tag{4}$$

其对应的逆变换矩阵实现为

$$f = T'FT. \tag{5}$$

T为$N \times N$大小的DCT矩阵，其定义为：

$$T[i,j] = \begin{cases} \frac{1}{\sqrt{N}}, & i = 0 \\ \sqrt{\frac{2}{N}} \cdot \frac{cos(2j+1) \cdot i\pi}{2N}, & i > 0 \end{cases} \tag{6}$$

以$M = N = 8$为例，执行案例 Exp2 ，对固定的矩阵 data 进行DCT变换和反变换。不难发 现，DCT变换是可逆无损的。且随着迭代次数的减少，运算速度得到了显著提升。

```
    st2 = time.time()
    row = np.arange(M)
    col = 2*np.arange(N)+1
    T = np.sqrt(2/N)*np.cos(np.matmul(row.reshape(M, 1), col.reshape(1,
N))*np.pi/(2*N))
    T[0,:]=1/np.sqrt(N)


    dct_coef = np.matmul(np.matmul(T,data),T.T)
    data_rec_matrix = np.floor(np.matmul(np.matmul(T.T, dct_coef), T) +0.5)
    end2 = time.time()

    # print("data=\n", data)
```

```
    # print()
    print("The reconstructed version of original data for 2D DCT/IDCT
Transform:")
    print(np.floor(data_rec_matrix+0.5))

    print()
    print(f'Runtime is {end2-st2:.6f} s for {M} x {N} 2D-DCT/IDCT Transform')
```

以下为代码执行后的输出结果（original数据与在前面一致）：

```
The reconstructed version of original data for 2D DCT/IDCT Transform:
[[113. 146. 228. 129.  24.  80. 252.  76.]
 [ 46. 123. 240. 185.  14. 247.  35.  54.]
 [114. 210.  93. 170.  47.   4.  80. 111.]
 [130.  49.  92. 231. 249. 183. 138. 116.]
 [ 16. 135. 197. 142.  89. 168.   5.  21.]
 [ 35. 115. 100. 124.  94.  53. 200.  80.]
 [ 14. 210. 211. 254.  29. 132.  75. 185.]
 [ 42. 129. 222. 113. 221. 154. 213.  97.]]

Runtime is 0.000998 s for 8 x 8 2D-DCT/IDCT Transform
```

3.2 DCT变换系数的信息熵

3.2.1 中宽量化器的影响

对DCT变换系数直接进行量化（中宽），可以重建信号的误差为代价，显著的降低信源的信息熵，达到图像信号压缩的目的。下面的python函数 dct_entropy_demo 提供了简单的中宽量化的演示功能。可以通过改变2D-DCT变换核的尺寸和中宽量化器的量化步长，在图像信号的信息熵和重建质量两个指标之间取得平衡。

```
def dct_entropy_demo(filename, blksize, q):

    print(f'The block size of DCT is {blksize[0]}')
    print(f'The step size of midtread quantizer is {q}')

    img = Image.open(filename)
    img_yuv = img.convert('YCbCr')

    img_yuv_offset = np.array(img_yuv, dtype='float') - 128
    img_yuv_rec = np.array(img)

    for i in range(3):
        img_yuv_offset_pad, pad1, pad2 = padding(
            img_yuv_offset[:, :, i], blksize)
        x = blockproc(img_yuv_offset_pad, blksize, dct)
        x_h = np.floor(x/q+0.5)
        [Height, Width] = x_h.shape
        hist, bins = np.histogram(x_h, bins=256, range=(0, 255))
        p = hist/(Height*Width)
        entropy = (-p*np.log2(p+1e-08)).sum()
        print(f'Entropy of input image for channel {i} = {entropy}')

        y_h = x_h*q
        img_yuv_rec[:, :, i] = unpadding(
            blockproc(y_h, blksize, idct), pad1, pad2)+128
```

```
    img_rec = Image.fromarray(
        np.array(img_yuv_rec, dtype='uint8'), 'YCbCr').convert('RGB')

    img = np.array(img)
    img_rec = np.array(img_rec)

    Image.fromarray(np.array(img_rec, dtype='uint8')).save('rec_lenna.png')
    ssim = cal_ssim(img_rec, img, data_range=255, multichannel=True)
    # ssim=calculate_ssim(img,img_rec)
    mse = cal_mse(img, img_rec)
    psnr = cal_psnr(mse)

    print(f'SSIM:{ssim:.4f},MSE:{mse:.4f},PSNR:{psnr:.4f} dB')

    plt.figure()

    plt.subplot(1,2,1)
    plt.imshow(img) # 仅关系show
    plt.title('original image')
    plt.axis('off')

    plt.subplot(1,2,2)
    plt.imshow(img_rec) # 仅关系show
    plt.title('reconstructed image')
    plt.axis('off')

    plt.show()
```

将2D-DCT的变换核尺寸设为4×4,同时中宽量化器的步长设为50。通过下面的代码块 Exp3 调 用 dct_entropy_demo ， 通过代码执行中的打印信息不难看出，即使对DCT系数直接做比较粗的量化， 重建后的画质仍然可以让人接受。同时，Y、Cb和Cr几个通道的信息熵下降明显。

```
if __name__ == '__main__':
    filename = 'Lenna.png'
    blksize = [4, 4]
    q = 50
    dct_entropy_demo(filename, blksize, q)
```

以下为代码执行后的输出结果：

```
The block size of DCT is 4
The step size of midtread quantizer is 50
Entropy of input image for channel 0 = 0.40890689335517705
Entropy of input image for channel 1 = 0.15272272862740138
Entropy of input image for channel 2 = 0.4662324499035637
SSIM:0.7370,MSE:54.0765,PSNR:30.8007 dB
```

将2D-DCT的变换核尺寸设为8×8,同时中宽量化器的步长保持不变。可以发现变换核的尺寸增大可以限制降低测试图像的信息熵，同时重建后的画质亦显著提高。

```
if __name__ == '__main__':
    filename = 'Lenna.png'
    blksize = [8,8]
    q = 50
    dct_entropy_demo(filename, blksize, q)
```

以下为代码执行后的输出结果:

```
The block size of DCT is 8
The step size of midtread quantizer is 50
Entropy of input image for channel 0 = 0.26191883875393096
Entropy of input image for channel 1 = 0.07153736996115548
Entropy of input image for channel 2 = 0.18150330960234534
SSIM:0.7786,MSE:41.9980,PSNR:31.8985 dB
```

继续将2D-DCT的变换核尺寸增大为16×16,同时中宽量化器的步长保持不变。可以发现变换核的尺寸增大仍然可以限制降低测试图像的信息熵，同时重建后的画质亦显著提高，但增幅相对变缓。

```
if __name__ == '__main__':
    filename = 'Lenna.png'
    blksize = [16, 16]
    q = 50
    dct_entropy_demo(filename, blksize, q)
```

以下为代码执行后的输出结果：

```
The block size of DCT is 16
The step size of midtread quantizer is 50
Entropy of input image for channel 0 = 0.2165962270048971
Entropy of input image for channel 1 = 0.048453528725270305
Entropy of input image for channel 2 = 0.08423446985370696
SSIM:0.7896,MSE:40.0087,PSNR:32.1093 dB
```

3.2.2 JPEG压缩标准中的量化矩阵

JPEG压缩标准根据心理学研究结果，为亮度和色度图分别提供了不同的量化矩阵。不同的频率对应不同的量化矩阵系数值。下面的python函数 dct_jpeg_entropy_demo 给出了相应的JPEG量化矩阵的实现。可以通过改变量化步长，在图像信号的信息熵和重建质量两个指标之间取得平衡。

```
def dct_jpeg_entropy_demo(filename, q):
    blksize=[8,8]
    print(f'The block size of DCT is {blksize[0]}')
    print(f'The step size of midtread quantizer is {q}')

    jpgQstepsY = [[16, 11, 10, 16, 24, 40, 51, 61],
                  [12, 12, 14, 19, 26, 58, 60, 55],
                  [14, 13, 16, 24, 40, 57, 69, 56],
                  [14, 17, 22, 29, 51, 87, 80, 62],
                  [18, 22, 37, 56, 68, 109, 103, 77],
                  [24, 35, 55, 64, 81, 104, 113, 92],
                  [49, 64, 78, 87, 103, 121, 120, 101],
                  [72, 92, 95, 98, 112, 100, 103, 99]]
    jpgQstepsY = np.array(jpgQstepsY)*q
    jpgQstepsC = [[17, 18, 24, 47, 66, 99, 99, 99],
                  [18, 21, 26, 66, 99, 99, 99, 99],
                  [24, 26, 56, 99, 99, 99, 99, 99],
```

```
                  [47, 66, 99, 99, 99, 99, 99, 99],
                  [99, 99, 99, 99, 99, 99, 99, 99],
                  [99, 99, 99, 99, 99, 99, 99, 99],
                  [99, 99, 99, 99, 99, 99, 99, 99],
                  [99, 99, 99, 99, 99, 99, 99, 99]]
    jpgQstepsC = np.array(jpgQstepsC)*q

    img = Image.open(filename)
    img_yuv = img.convert('YCbCr')

    img_yuv_offset = np.array(img_yuv, dtype='float') - 128
    img_yuv_rec = np.array(img)

    for i in range(3):
        img_yuv_offset_pad, pad1, pad2 = padding(
            img_yuv_offset[:, :, i], blksize)
        x = blockproc(img_yuv_offset_pad, blksize, dct)



        #x_h = np.floor(x/q+0.5)
        if i > 0:
            x_h = np.floor(blockproc(x, blksize, div(jpgQstepsC))+0.5)
        else:
            x_h = np.floor(blockproc(x, blksize, div(jpgQstepsY))+0.5)

        [Height, Width] = x_h.shape
        hist, bins = np.histogram(x_h,bins=int(x_h.max()-x_h.min()+1) ,range=
(x_h.min(),x_h.max()))
        p = hist/(Height*Width)
        entropy = (-p*np.log2(p+1e-08)).sum()
        print(f'Entropy of input image for channel {i} = {entropy}')

        # y_h = x_h*q
        if i > 0:
            y_h = blockproc(x_h, blksize, mul(jpgQstepsC))+0.5
        else:
            y_h = blockproc(x_h, blksize, mul(jpgQstepsY))+0.5

        img_yuv_rec[:, :, i] = unpadding(
            blockproc(y_h, blksize, idct), pad1, pad2)+128

    img_rec = Image.fromarray(
        np.array(img_yuv_rec, dtype='uint8'), 'YCbCr').convert('RGB')

    img = np.array(img)
    img_rec = np.array(img_rec)

    Image.fromarray(np.array(img_rec, dtype='uint8')).save('rec_lenna.png')
    ssim = cal_ssim(img_rec, img, data_range=255, multichannel=True)
    # ssim=calculate_ssim(img,img_rec)
    mse = cal_mse(img, img_rec)
    psnr = cal_psnr(mse)

    print(f'SSIM:{ssim:.4f},MSE:{mse:.4f},PSNR:{psnr:.4f} dB')

    plt.figure()

    plt.subplot(1,2,1)
```

```
    plt.imshow(img) # 仅关系show
    plt.title('original image')
    plt.axis('off')

    plt.subplot(1,2,2)
    plt.imshow(img_rec) # 仅关系show
    plt.title('reconstructed image')
    plt.axis('off')

    plt.show()
```

将量化矩阵的等级设为3。调用 dct_jpeg_entropy_demo ， 通过代码执行后的结果不难看出，JPEG量化矩阵能够取得不错的压缩效果。

```
if __name__ == '__main__':
    blksize = [8,8]
    filename = 'Lenna.png'
    dct_jpeg_entropy_demo(filename,q=3) # exp4
```

以下为代码执行后的输出结果：

```
The block size of DCT is 8
The step size of midtread quantizer is 3
Entropy of input image for channel 0 = 0.4800495044258819
Entropy of input image for channel 1 = 0.17425947667266486
Entropy of input image for channel 2 = 0.1872648295589554
SSIM:0.7771,MSE:41.7537,PSNR:31.9239 dB
```

3.3 图像有损压缩系统设计

在3.1和3.2的基础上，我们可以考虑针对图像数据，基于2D-DCT变换和JPEG量化矩阵，进行有损压缩系统的设计。 ImgLossyCodec_Encoder_Demo 为一个简单的图像压缩系统的函数实现示例，具体步骤包括：

Step 1 预处理：读入原始图像数据，并将图像数据从RGB颜色空间转换到YCbCr颜色空间。完 成参数设置，如码流文件的存放文件夹检查等。

step 2 对图像数据的每个通道，利用python写的函数函数 blockproc 、 dct ， 完成图像信 号的分块二维DCT变换和JPEG量化，调用 Lec4 中的图像单通道熵编码函数 encode_channel , 完成二进制码流的生成和保存。并打印编码时间和压缩倍数。

step 3 对图像数据的每个通道，调用 Lec4 中的图像单通道熵解码函数 decode_channel ,再 通过反量化和分块逆二维DCT变换，完成图像数据的重建，再重新将图像数据转换到RGB颜色 空间。并打印解码时间和图像质量。

```
def dct_jpeg_entropy_demo(filename, q):
    blksize=[8,8]
    print(f'The block size of DCT is {blksize[0]}')
    print(f'The step size of midtread quantizer is {q}')

    jpgQstepsY = [[16, 11, 10, 16, 24, 40, 51, 61],
                  [12, 12, 14, 19, 26, 58, 60, 55],
                  [14, 13, 16, 24, 40, 57, 69, 56],
                  [14, 17, 22, 29, 51, 87, 80, 62],
                  [18, 22, 37, 56, 68, 109, 103, 77],
                  [24, 35, 55, 64, 81, 104, 113, 92],
                  [49, 64, 78, 87, 103, 121, 120, 101],
                  [72, 92, 95, 98, 112, 100, 103, 99]]
    jpgQstepsY = np.array(jpgQstepsY)*q
    jpgQstepsC = [[17, 18, 24, 47, 66, 99, 99, 99],
                  [18, 21, 26, 66, 99, 99, 99, 99],
                  [24, 26, 56, 99, 99, 99, 99, 99],
                  [47, 66, 99, 99, 99, 99, 99, 99],
                  [99, 99, 99, 99, 99, 99, 99, 99],
                  [99, 99, 99, 99, 99, 99, 99, 99],
                  [99, 99, 99, 99, 99, 99, 99, 99],
                  [99, 99, 99, 99, 99, 99, 99, 99]]
    jpgQstepsC = np.array(jpgQstepsC)*q

    img = Image.open(filename)
    img_yuv = img.convert('YCbCr')

    img_yuv_offset = np.array(img_yuv, dtype='float') - 128
    img_yuv_rec = np.array(img)


    compress_time = 0
    decompress_time = 0
    size=0
    for i in range(3):
        img_yuv_offset_pad, pad1, pad2 = padding(
            img_yuv_offset[:, :, i], blksize)
        x = blockproc(img_yuv_offset_pad, blksize, dct)


        #x_h = np.floor(x/q+0.5)
        if i > 0:
            x_h = np.floor(blockproc(x, blksize, div(jpgQstepsC))+0.5)
        else:
            x_h = np.floor(blockproc(x, blksize, div(jpgQstepsY))+0.5)


        # 压缩
        s,ct = Compresser.main_compress(np.array(x_h,dtype='int'))
```

```
        compress_time+=ct
        size+=s
        # 解压缩
        dt,x_h = Compresser.main_decompress()
        decompress_time+=dt
        x_h = np.squeeze(x_h)



        [Height, Width] = x_h.shape
        hist, bins = np.histogram(x_h,bins=int(x_h.max()-x_h.min()+1) ,range=
(x_h.min(),x_h.max()))
        p = hist/(Height*Width)
        entropy = (-p*np.log2(p+1e-08)).sum()
        print(f'Entropy of input image for channel {i} = {entropy}')

        # y_h = x_h*q
        if i > 0:
            y_h = blockproc(x_h, blksize, mul(jpgQstepsC))+0.5
        else:
            y_h = blockproc(x_h, blksize, mul(jpgQstepsY))+0.5

        img_yuv_rec[:, :, i] = unpadding(
            blockproc(y_h, blksize, idct), pad1, pad2)+128

    img_rec = Image.fromarray(
        np.array(img_yuv_rec, dtype='uint8'), 'YCbCr').convert('RGB')

    img = np.array(img)
    img_rec = np.array(img_rec)

    Image.fromarray(np.array(img_rec, dtype='uint8')).save('rec_lenna.png')
    ssim = cal_ssim(img_rec, img, data_range=255, multichannel=True)
    # ssim=calculate_ssim(img,img_rec)
    mse = cal_mse(img, img_rec)
    psnr = cal_psnr(mse)

    H, W, C = img.shape
    original = H*W
    numel = original * C
    bpp = size*8/original

    print()
    print(f'quality:{q}')
    print(f'encoded time:{compress_time:.4f} s')
    print(f'decoded time:{decompress_time:.4f} s')
    print(f'original size:{numel} bytes')
    print(f'compressed size:{size} bytes')
    print(f'compressed ratio:{numel/size:.4f} ')
    print(f'bpp:{bpp:.4f}, SSIM:{ssim:.4f},MSE:{mse:.4f},PSNR:{psnr:.4f} dB')
```

调用下面的示例代码 Exp5

```
if __name__ == '__main__':
    filename = 'kodim12.png'
    blksize = [8, 8]
    dct_jpeg_entropy_demo(filename,q=3)  # exp5
```

以下为代码执行后的输出结果：

```
quality:3
encoded time:2.794187 s
decoded time:2.119039 s
original size:1179648 bytes
compressed size:162023 bytes
compressed ratio:7.2807
SSIM:0.7890,MSE:35.1307,PSNR:32.6739 dB
```

3.4 图像有损压缩系统性能评估

系统性能的评价指标包含多种判断标准，如计算复杂度、压缩率等。本实验中，我们主要考虑压缩率。并将所设计的有损压缩系统与matlab中默认的JPEG的编解码器进行性能比较。具体的，对每种编解码器，我们通过调节量化步长或者质量因子等控制参数，来获取对应的<码率，图像质量>数据。其中码率用位率（Bit Per Pixel，bpp）表示。图像质量用所有RGB分量的SSIM，或者均方差的平均值计算得到的PSNR来表示。最后，绘制出相应的图像质量与位率的关系曲线。通过曲线可以很容易判断出，不同编解码器的性能优劣。

首先执行下面的代码块 Exp6 中的第一部分，通过调节量化步长 的取值，完成对所提出的编 解码器的性能测试。为后续的率失真性能对比提供数据支撑。

```
if __name__ == '__main__':
    filename = 'kodim12.png'
    blksize = [8, 8]
    qfactor = [1,2,4,6]
    bpp = []
    psnr = []
    ssim = []
    for q in qfactor:
        ssim1,bpp1,psnr1=dct_jpeg_entropy_demo(filename,q=q)  # exp5
        bpp.append(bpp1)
        psnr.append(psnr1)
        ssim.append(ssim1)
```

以下为代码执行后的部分输出结果：

```
quality:1
encoded time:2.8169 s
decoded time:2.2274 s
original size:1179648 bytes
compressed size:177014 bytes
```

```
compressed ratio:6.6642
SSIM:0.8838,MSE:20.0319,PSNR:35.1136 dB

quality:2
encoded time:2.7577 s
decoded time:2.1580 s
original size:1179648 bytes
compressed size:166491 bytes
compressed ratio:7.0854
SSIM:0.8300,MSE:28.9144,PSNR:33.5197 dB

quality:4
encoded time:2.7066 s
decoded time:2.1506 s
original size:1179648 bytes
compressed size:159605 bytes
compressed ratio:7.3910
SSIM:0.7650,MSE:40.9140,PSNR:32.0121 dB

quality:6
encoded time:2.6726 s
decoded time:2.0321 s
original size:1179648 bytes
compressed size:157300 bytes
compressed ratio:7.4994
SSIM:0.7105,MSE:54.9151,PSNR:30.7339 dB
```

继续执行以下代码，完成python中自带的JPEG编解码器的性能测试：

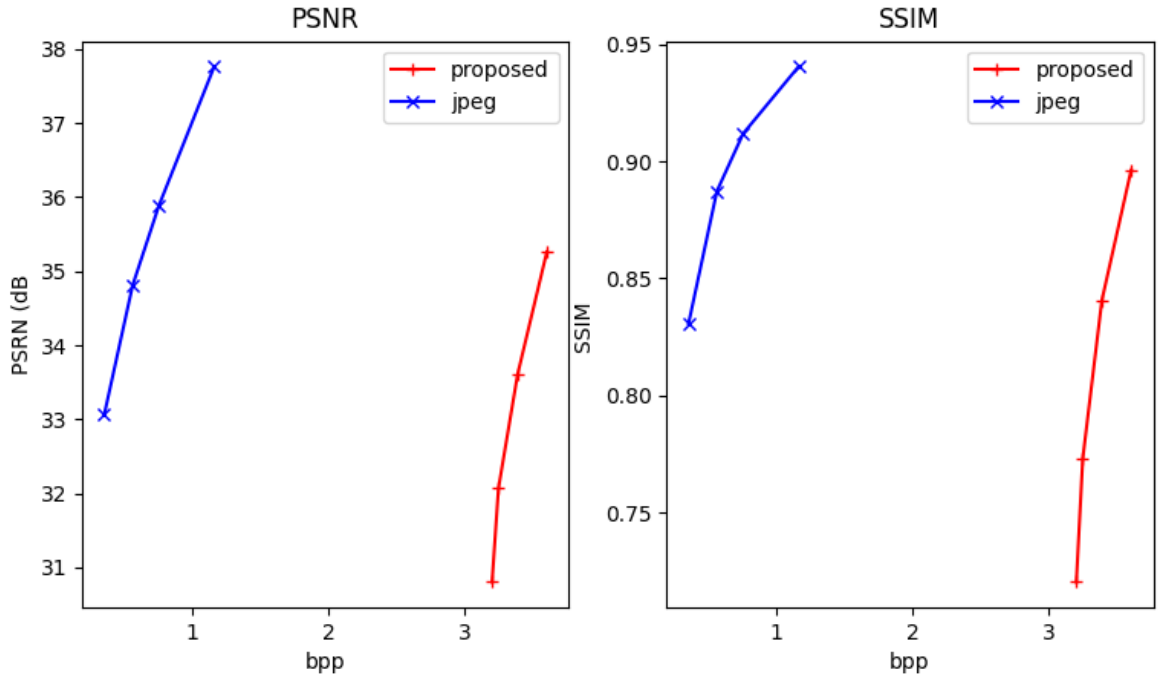
```
    qfactor = [20,40,60,80]
    img = np.array(Image.open(filename))
    H,W,C=img.shape
    numel = H*W
    bpp_PIL = []
    psnr_PIL = []
    ssim_PIL = []
    for q in qfactor:
        path = f'filename_{q}.jpg'
        rec_img = Image.fromarray(img).save(path, format='jpeg', quality=q)
        rec_img = np.array(Image.open(path))
        bpp_PIL.append(os.path.getsize(path)*8/numel)
        psnr_PIL.append(cal_psnr(((rec_img-img)**2).mean()))
        ssim_PIL.append(cal_ssim(rec_img,img,multichannel=True))
```

执行以下代码，画出所设计的编解码器，以及PIL中自带的JPEG编解码器的图像质量-码率曲线：

```
  plt.figure()
    plt.subplot(1,2,1)
    plt.title('PSNR')
    # plt.grid('off')
    plt.xlabel('bpp')
    plt.ylabel('PSRN (dB')
    plt.plot(bpp, psnr,c='red',marker='+')
    plt.plot(bpp_PIL, psnr_PIL,c='blue',marker='x')
    plt.legend(['proposed','jpeg'])
```

```
    plt.subplot(1,2,2)
    plt.title('SSIM')
    # plt.grid('off')
    plt.xlabel('bpp')
    plt.ylabel('SSIM ')
    plt.plot(bpp, ssim,c='red',marker='+')
    plt.plot(bpp_PIL, ssim_PIL,c='blue',marker='x')
    plt.legend(['proposed','jpeg'])

    plt.show()
```

以下为代码执行后的输出结果：



从上面的性能比较图中可以看出，PIL中自带的JPEG编解码器的性能比我们所提出的编解码系统，在率失真性能上有明显的优势。比如相同的PSNR/SSIM值下，JPEG编解码器需要更少的位率。

4 课后习题

1）3.3.节中的 ImgLossyCodec_Encoder_Demo 中还存在很多需要优化的地方,比如量化后的 DCT系数的熵编码的过程。JPEG编码标准中针对DC和AC系数的特点，分别采用了DPCM，Z 字形扫描和游程编码（Run Length Coding, RLC）等技术以消除频率系数间的空间相关性。 再次基础上，为了减少赫夫曼编解码中的符号表过大的问题，设计了赫夫曼编码的变体方案， 进一步降低了统计冗余，提高了压缩效率。请尝试在代码基础上，去复现JPEG编码标准中的 相应的技术。

这里展示部分关键函数，具体代码参考附件。

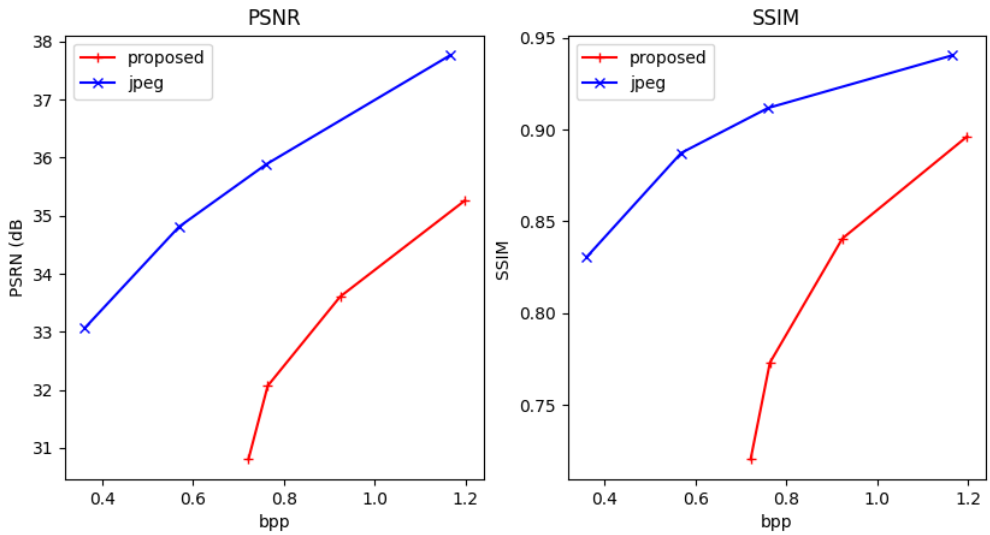
```
def getsize(val):
    # return np.ceil(np.log2(np.abs(val)+1)+0.5)
    if val == 0: return 0
    return np.ceil(np.log2(np.abs(val)+0.5))

# 返回acindex
def getindex(runlength,dcsize):
    if runlength == 0 and dcsize == 0 :
        return 160
    if runlength == 15 and dcsize == 0:
        return 161
```

```
    index = runlength*10+dcsize-1
    return index

# 针对序对 (SIZE, AMPLITUDE) 中的SIZE，会采用哈夫曼编码，同时产生一张哈夫曼编码表。而AMPLITUDE则直接用二进制bit串表示。
def dpcm(ac_coef,bitout,dctable):
    def diff(img):
        img = np.array(img, dtype='int')
        img[1:] = img[1:]-img[:-1]
        return img

    ac_coef = diff(ac_coef)
    size = 0
    for ac in ac_coef:
        acsize = int(getsize(ac))
        symbol = dctable[acsize]
        size+=write_symbol(symbol,bitout)    # size固定哈夫曼表编码
        size+=write_int(ac,acsize+1,bitout)  # amplitude直接编码
    # ac.write(256)  # 结束标识符（需要超过最大范围）  # 可以通过编码长宽来解决
    return size

def write_symbol(symbol,bitout):
    size = 0
    for s in symbol:
        size+=bitout.write(int(s))
    return size

def read_symbol(inp,table):
    hashtable = set(table)
    s=''
    while True:
        s+=str(inp.read_no_eof())
        if s in hashtable:
            for i,x in enumerate(table):
                if x == s:
                    return i,s # size_category

def idpcm(bitin,table,numOfDC):  # 可以通过记录一个数量来标记结束，但可能并非如此简单
    def reverse_diff(img):
        for i in range(1,img.shape[0]):
            img[i]=img[i]+img[i-1]
        return img

    dc_coef = []

    for i in range(numOfDC):
        size_category,s = read_symbol(bitin,table)

        dc = read_int(size_category+1,bitin)
        if dc>=2**size_category:  # -6 26 (4+1) -4 12 (3+1)
            dc = -(2**(size_category+1)-dc)
        dc_coef.append(dc)

    dc_coef = reverse_diff(np.array(dc_coef))
    return dc_coef

# 针对序对 ((RunLength/SIZE), AMPLITUDE) 中的(RunLength/SIZE)，会采用哈夫曼编码，同时产生一张哈夫曼编码表。而AMPLITUDE则直接用二进制bit串表示。
```

```
def rle(dc_coef,bitout,table):
    size = 0
    # with contextlib.closing(BitOutputStream(open(outfile, "wb"))) as bitout:
    num=0
    for dc in dc_coef:
        if dc[1]==0:
            dcsize = 0
        else:
            dcsize = int(getsize(dc[1]))
        runlength = dc[0]
        index = getindex(runlength,dcsize)
        symbol = table[index]
        # if num>15900:
        #     print('rle:',num,index,symbol)

        num+=1

        size+=write_symbol(symbol,bitout)
        if index >= 160: # 160 （0，0） 161 （15，0） 直接进行解析
            continue

        index = index+1
        if index % 10 == 10:
            size_category = 10
        else:
            size_category = index%10

        size+=write_int(dc[1],size_category+1,bitout)
    print(size)
    return size

def write_int(val,bit,bitout):
    size = 0
    for j in reversed(range(bit)):
        size+=bitout.write((val >> j) & 1)
    return size

def read_int(n,inp):
    result = 0
    for _ in range(n):
        bit = inp.read_no_eof()
        result = (result << 1) | bit  # Big endian
    return result


def irle(bitin,table,numOfAC):
    dc_coef = []

    size = 0
    for i in range(numOfAC):
        index,s = read_symbol(bitin,table)  # 跟dpcm不同，需要解析一下index
        size+=len(s)
        # print(size//8)
        # if i>15900:
        # print(i,index,s)
        if index == 160:
            # 遇到结束标志，后面的
            dc_coef.append([0,0])
```

```
            continue
        if index == 161:
            dc_coef.append([15,0])
            continue

        index = index+1
        if index % 10 == 10:
            size_category = 10
            dc1 = index //10-1
        else:
            size_category = index%10
            dc1 = index //10

        dc2 = read_int(size_category+1,bitin)
        if dc2>=2**size_category:  # -6 26 (4+1) -4 12 (3+1)
            dc2 = -(2**(size_category+1)-dc2)
        dc_coef.append([dc1,dc2])
        size+=size_category+1
    return dc_coef

def reverseZigZag(block,zigzag):
    new_block = np.zeros(block.shape)
    for i in range(64):
        new_block[i]=block[zigzag[i]]
    return np.reshape(new_block,[8,8])
```

以下对`kodim12.png`进行jpeg压缩后的输出结果：



2）3.4节中采用了PSNR/SSIM两个指标来评估压缩重建后的图像画质，请查阅文献，探索一 下图像质量评价领域中的其他平均指标。另外，3.4节中只给出了定性的分析方法，如何定量分析不同的编解码器性能的优劣呢？请以BD-PSNR或者BD-Rate为关键词来寻找相应的解决方案。

BD-PSNR（Bjøntegaard Delta Peak Signal-to-Noise Ratio）和BD-Rate（Bjøntegaard Delta Rate）是在视频编码领域用来衡量编解码性能的两个重要指标。这两个指标是由Gisle Bjøntegaard提出的，用于比较不同编码设置或编解码器之间的性能差异。

BD-PSNR：BD-PSNR衡量的是在相同比特率下，编码视频质量（以PSNR衡量）的改进或下降。一个更高的BD-PSNR值意味着在相同的比特率下视频质量有所提高。BD-PSNR是通过比较两个编码方案在不同比特率下的PSNR值，通过插值计算得到整体的平均PSNR差值。

BD-Rate：BD-Rate衡量的是在达到相同视频质量时，所需比特率的增加或减少百分比。如果一个编码方案的BD-Rate值为-10%，这意味着与参考编码方案相比，它在达到同样的视频质量时所需的比特率减少了10%。因此，BD-Rate越低，表示编码效率越高。

```
def BD_PSNR(R1, PSNR1, R2, PSNR2, piecewise=0, base_first=True):
    if base_first:
        R1, PSNR1, R2, PSNR2 = R2, PSNR2, R1, PSNR1
    lR1 = np.log(R1)
    lR2 = np.log(R2)

    PSNR1 = np.array(PSNR1)
    PSNR2 = np.array(PSNR2)

    p1 = np.polyfit(lR1, PSNR1, 3)
    p2 = np.polyfit(lR2, PSNR2, 3)

    # integration interval
    min_int = max(min(lR1), min(lR2))
    max_int = min(max(lR1), max(lR2))

    # find integral
    if piecewise == 0:
        p_int1 = np.polyint(p1)
        p_int2 = np.polyint(p2)

        int1 = np.polyval(p_int1, max_int) - np.polyval(p_int1, min_int)
        int2 = np.polyval(p_int2, max_int) - np.polyval(p_int2, min_int)
    else:
        # See https://chromium.googlesource.com/webm/contributor-
guide/+/master/scripts/visual_metrics.py
        lin = np.linspace(min_int, max_int, num=100, retstep=True)
        interval = lin[1]
        samples = lin[0]
        v1 = scipy.interpolate.pchip_interpolate(np.sort(lR1),
PSNR1[np.argsort(lR1)], samples)
        v2 = scipy.interpolate.pchip_interpolate(np.sort(lR2),
PSNR2[np.argsort(lR2)], samples)
        # Calculate the integral using the trapezoid method on the samples.
        int1 = np.trapz(v1, dx=interval)
        int2 = np.trapz(v2, dx=interval)

    # find avg diff
    avg_diff = (int2 - int1) / (max_int - min_int)

    return avg_diff


def BD_RATE(R1, PSNR1, R2, PSNR2, piecewise=1, base_first=True):
    if base_first:
        R1, PSNR1, R2, PSNR2 = R2, PSNR2, R1, PSNR1
    lR1 = np.log(R1)
    lR2 = np.log(R2)

    # rate method
    p1 = np.polyfit(PSNR1, lR1, 3)
    p2 = np.polyfit(PSNR2, lR2, 3)
```

```
    # integration interval
    min_int = max(min(PSNR1), min(PSNR2))
    max_int = min(max(PSNR1), max(PSNR2))

    # find integral
    if piecewise == 0:
        p_int1 = np.polyint(p1)
        p_int2 = np.polyint(p2)

        int1 = np.polyval(p_int1, max_int) - np.polyval(p_int1, min_int)
        int2 = np.polyval(p_int2, max_int) - np.polyval(p_int2, min_int)
    else:
        lin = np.linspace(min_int, max_int, num=100, retstep=True)
        interval = lin[1]
        samples = lin[0]
        v1 = scipy.interpolate.pchip_interpolate(np.sort(PSNR1),
lR1[np.argsort(PSNR1)], samples)
        v2 = scipy.interpolate.pchip_interpolate(np.sort(PSNR2),
lR2[np.argsort(PSNR2)], samples)
        # Calculate the integral using the trapezoid method on the samples.
        int1 = np.trapz(v1, dx=interval)
        int2 = np.trapz(v2, dx=interval)

    # find avg diff
    avg_exp_diff = (int2 - int1) / (max_int - min_int)
    avg_diff = (np.exp(avg_exp_diff) - 1) * 100
    return avg_diff
```