Task 3 Report

1. Task requirement

Use Block Vector Quantization Coding (BVQC) to encode an image and then decode the encoded file to reconstruct the image.

The main program should do the following things:

1. **Get the input from the user:**
   Get the file: Prompt the user to input the file name of an image file. Give hints to help the user to input a valid file and make sure the exceptions can be handled in case that the specified file does not exist.
   Get the block size d: Prompt the user to input the block size d. Give hints to help the user to input a valid value of block size which must be an integer power of 2. If null string is input, the default block size (4) will be used.
2. **Encode**
   1) Get the input image size and normalize its intensity values such that each pixel value is bounded in [0,255]. Display the image.

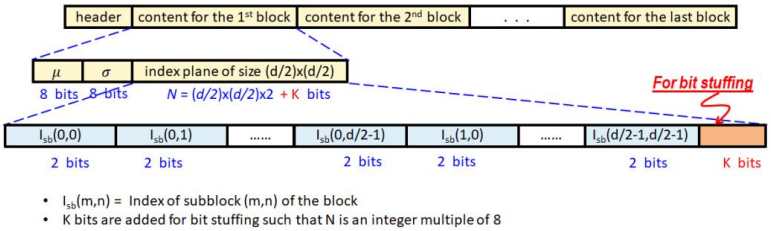
   2) Partition the image into block of size d*d. For each block, compute the mean and standard deviation.
      Then compute the g0=max(0,mean-standard deviation) and g1=min(255,mean+standard deviation) and construct a code book based on the table below.

index	0	1	2	3
codeword	$c_0 = \begin{bmatrix} g_0 & g_0 \\ g_1 & g_1 \end{bmatrix}$	$c_1 = \begin{bmatrix} g_1 & g_1 \\ g_0 & g_0 \end{bmatrix}$	$c_2 = \begin{bmatrix} g_0 & g_1 \\ g_0 & g_1 \end{bmatrix}$	$c_3 = \begin{bmatrix} g_1 & g_0 \\ g_1 & g_0 \end{bmatrix}$

   3) After that, divide the block into subblocks of size 2x2 and approximate each of them as the closest codeword based on their distances from the subblock according to the distance formula below.(When there are more than one closest codewords, the one with the minimum index value is picked)

$$J = \sum_{m=0}^{1} \sum_{n=0}^{1} (x(m,n) - c_i(m,n))^2$$

4) Next, save the encoding result in a file in the format shown below.



- $I_{sb}(m,n)$ = Index of subblock (m,n) of the block
- K bits are added for bit stuffing such that N is an integer multiple of 8

Figure 1 The structure of the file format

Byte(s)	No. of bits	Information
1	8	It specifies the header length in terms of number of bytes
2	8	It specifies the value of d, where $d \times d$ is the block size
3-4	16	It specifies the number of columns of blocks in the image
5-6	16	It specifies the number of rows of blocks in the image

Table 2 structure of the header

5) Finally, return the data structure that carries the information of the encoding output which contains the mean and standard deviation of each block and the index of a specific subblock.

3. Decode

1) Read the file to get the data structure that carries the encoding result.
2) Decode the file, display the reconstructed image and finally save the reconstructed image as a conventional image file.

4. Evaluation of the result

Based on the formula, calculate the mean square error (mse) and peak-to-peak signal-to-noise ratio (PPSNR) of the reconstructed image.

2. Explanation of the code

1. Get the input from the user

1) Get the image file to be processed.

The function readfile() uses a dead loop to ask the user to input the file name continuously until the input file name is valid. Use try…except… to handle the case that the specified file does not exist.

```
def readfile():
    while(1):#The program will loop until a valid input is received.
        filename=input("Please input the file name of an image:\n")
        try:#Use try...except... to handle the exceptions in case that the file does not exist.
            img=mping.imread(filename)
        except IOError:
            print("<{}> does not exist!".format(filename))
        else:
            return filename
```

The result:

```
Please input the file name of an image:
asd
<asd> does not exist!

Please input the file name of an image:
asdas
<asdas> does not exist!

Please input the file name of an image:
myTimg.png

Please input the value of d:
```

2) Get the block size d

The function readd() uses a dead loop to ask the user to input the d until the d is valid. If a null string is input, return the default value of d which is 4.

```
def readd():
    while(1):#The program will loop until a valid input is received.
        d=input("Please input the value of d:\n")
        if(d==''):#If null string is input, return the default value 4 of d.
            return 4
        if(int(d)>1):#check whether d is an integer power of 2
            i=1
            while(i<=int(d)):
                if(i==int(d)):
                    return int(d)
                i*=2
        print("d must be an integer power of 2!\n")
```

```
if(int(d)>1):#check whether d is an integer power of 2
    i=1
    while(i<=int(d)):
        if(i==int(d)):
            return int(d)
        i*=2
```

This part is used to check whether d is an integer power of 2. The idea is use i to enumerate each value of 2 to the power of an integer and check whether the value of i can equal to d. If currently i is bigger than d and the program still doesn't return the value of d to the main program, which means that there is no chance for i to equal to d, so that we can end the loop.

The results:

```
Please input the value of d:
5
d must be an integer power of 2!


Please input the value of d:
7
d must be an integer power of 2!


Please input the value of d:
4
```

2. Encode

The whole encode function:

```
def BVQCencode(in_image_filename,out_encoding_result_filename,d):
    img=mping.imread(in_image_filename)#read the image
    plt.imshow(img,cmap='gray')
    plt.show()#show the image
    X=np.array(img)*255#Normalise the pixel values to (0,255).
    rows_of_blocks=int(X.shape[0]/d)#Calculate the number of rows for blocks
    cols_of_blocks=int(X.shape[1]/d)#Calculate the number of columns for blocks
    meanplane=np.zeros([rows_of_blocks,cols_of_blocks],dtype="uint8")#create an array to store the mean of each block
    stdplane=np.zeros([rows_of_blocks,cols_of_blocks],dtype="uint8")#create an array to store the standard deviation of each block
    codeplane=np.zeros([int(X.shape[0]/2),int(X.shape[1]/2)],"uint8")#create an array to store the code words
    for i in range(0,X.shape[0],d):
        for j in range(0,X.shape[1],d):#enumerate each block
            block=X[i:i+d,j:j+d]#get the block from the normalised image
            miu=np.mean(block)#calculate the mean of the current block
            std=np.std(block)#calculate the standard deviation of the current block
            meanplane[int(i/d),int(j/d)]=np.uint8(round(miu))#store the mean value of this block to the appropriate position of the meanplane
            stdplane[int(i/d),int(j/d)]=np.uint8(round(std))#store the standard deviation value of this block to the appropriate position of the stdplane
            g0=max(0,miu-std)#compute g0
            g1=min(255,miu+std)#compute g1
            for m in range(0,d,2):
                for n in range(0,d,2): #enumerate each subblock in the current block
                    code=codeword(X[i+m:i+m+2,j+n:j+n+2],g0,g1)#get the code for the current subblock
                    codeplane[int((i+m)/2),int((j+n)/2)]=np.uint8(code)##store the code of the subblock to the appropriate position of the codeplane
    file=open(out_encoding_result_filename,"wb")
    header=np.zeros([6],dtype='uint8')#write the header
    header[0]=np.uint8(6)
    header[1]=np.uint8(d)
    header[2]=np.uint8(cols_of_blocks%256)
    header[3]=np.uint8(cols_of_blocks/256)
    header[4]=np.uint8(rows_of_blocks%256)
    header[5]=np.uint8(rows_of_blocks/256)
    for byte in header:
        file.write(byte)
    for i in range(0,rows_of_blocks):
        for j in range(0,cols_of_blocks):#enumerate each block
            file.write(meanplane[i,j])#write the mean of this block into the file
            file.write(stdplane[i,j])#write the standard deviation of this block into the file
            cnt=0#used to count from 0 to 4 so that we can know when a byte will be generated
            now=0#used to store the current value of the byte
            for m in range(0,d,2):
                for n in range(0,d,2):#enumerate each subblock
                    cnt+=1
                    now=now*4+codeplane[int((i*d+m)/2),int((j*d+n)/2)]#change the current value of the byte
                    if cnt==4:#if 4 operations have been done, write the byte into the file and initialise the cnt and now
                        file.write(np.uint8(now))
                        cnt=0
                        now=0
            if cnt!=0:#if finally, the cnt is not 0, it means that we have to add K bits to form a complete byte,
                now*=(4**(4-cnt))
                file.write(np.uint8(now))
    file.close()
    return ({'M':meanplane,'Sd':stdplane,'Idx':codeplane})
```

1) Read and show the image

```
img=mping.imread(in_image_filename)#read the image
plt.imshow(img,cmap='gray')
plt.show()#show the image
```

2) Preprocessing

```
X=np.array(img)*255#Normalise the pixel values to (0,255).
rows_of_blocks=int(X.shape[0]/d)#Calculate the number of rows for blocks
cols_of_blocks=int(X.shape[1]/d)#Calculate the number of columns for blocks
meanplane=np.zeros([rows_of_blocks,cols_of_blocks],dtype="uint8")#create an array to store the mean of each block
stdplane=np.zeros([rows_of_blocks,cols_of_blocks],dtype="uint8")#create an array to store the standard deviation of each block
codeplane=np.zeros([int(X.shape[0]/2),int(X.shape[1]/2)],"uint8")#create an array to store the code words
```

Normalize the pixel values to (0,255).
Calculate the number of rows and columns of the blocks.
Create one array to store the mean value of each block.

Create one array to store the standard deviation of each block.

Create one array to store the code words for each 2*2 subblock.

3) Process each block

```
for i in range(0,X.shape[0],d):
    for j in range(0,X.shape[1],d):#enumerate each block
        block=X[i:i+d,j:j+d]#get the block from the normalised image
        miu=np.mean(block)#calculate the mean of the current block
        std=np.std(block)#calculate the standard deviation of the current block
        meanplane[int(i/d),int(j/d)]=np.uint8(round(miu))#store the mean value of this block to the appropriate position of the meanplane
        stdplane[int(i/d),int(j/d)]=np.uint8(round(std))#store the standard deviation value of this block to the appropriate position of the stdplane
        g0=max(0,miu-std)#compute g0
        g1=min(255,miu+std)#compute g1
        for m in range(0,d,2):
            for n in range(0,d,2): #enumerate each subblock in the current block
                code=codeword(X[i+m:i+m+2,j+n:j+n+2],g0,g1)#get the code for the current subblock
                codeplane[int((i+m)/2),int((j+n)/2)]=np.uint8(code)##store the code of the subblock to the appropriate position of the codeplane
```

For each block, calculate the mean and the standard deviation and store them in the appropriate position in the meanplane and the stdplane. Since i and j are enumerating each block, int(i/d) and int(j/d) represent the corresponding row and column these values should be stored in the meanplane and the stdplane.

Calculate g0 and g1. Then enumerate each subblock in the current block. Call function codeword() to calculate the code word for each subblock and store the code word in the appropriate position in the codeplane. Because (i,j) represent the coordinates of the top left corner of the current block in the original image X, and (m,n) represent the coordinates of the top left corner of the current subblock in the current block, (i+m,j+n) represent the coordinates of the top left corner of the current subblock in the original image X. Since each subblock is of size 2*2, ((i+m)/2, (j+n)/2) should be the proper position to store the current code word in the codeplane.
The codeword function:

```
def codeword(a,g0,g1):
    minn=(a[0,0]-g0)**2+(a[0,1]-g0)**2+(a[1,0]-g1)**2+(a[1,1]-g1)**2#Calculate the
    code=0#Initial the codeword to be chosen to codeword C0
    count1=(a[0,0]-g1)**2+(a[0,1]-g1)**2+(a[1,0]-g0)**2+(a[1,1]-g0)**2#Calculate th
    if(count1<minn):#If this distance is less than minn, change the minn and code a
        minn=count1
        code=1
    count2=(a[0,0]-g0)**2+(a[0,1]-g1)**2+(a[1,0]-g0)**2+(a[1,1]-g1)**2#Calculate th
    if(count2<minn):#If this distance is less than minn, change the minn and code a
        minn=count2
        code=2
    count3=(a[0,0]-g1)**2+(a[0,1]-g0)**2+(a[1,0]-g1)**2+(a[1,1]-g0)**2#Calculate th
    if(count3<minn):#If this distance is less than minn, change the minn and code a
        minn=count3
        code=3
    return code
```

Variable minn is used to store the min value of the 4 distances.
Variable code is used to store which code should be chosen.
The function will ask for 3 parameters a(a 2*2 subblock), g0 and g1, and will return the code word should be chosen.

4) Write the file

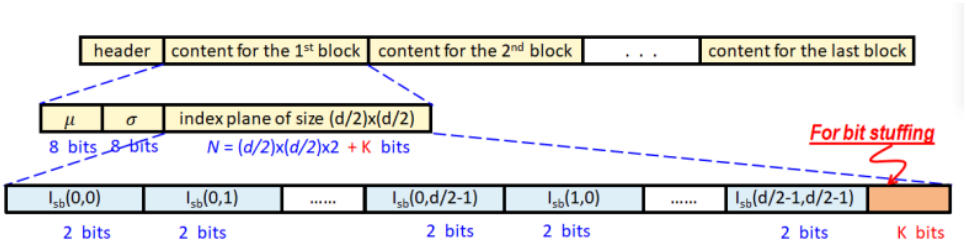
```
file=open(out_encoding_result_filename,"wb")
header=np.zeros([6],dtype='uint8')#write the header
header[0]=np.uint8(6)
header[1]=np.uint8(d)
header[2]=np.uint8(cols_of_blocks%256)
header[3]=np.uint8(cols_of_blocks/256)
header[4]=np.uint8(rows_of_blocks%256)
header[5]=np.uint8(rows_of_blocks/256)
for byte in header:
    file.write(byte)
for i in range(0,rows_of_blocks):
    for j in range(0,cols_of_blocks):#enumerate each block
        file.write(meanplane[i,j])#write the mean of this block into the file
        file.write(stdplane[i,j])#write the standard deviation of this block into the file
        cnt=0#used to count from 0 to 4 so that we can know when a byte will be generated
        now=0#used to store the current value of the byte
        for m in range(0,d,2):
            for n in range(0,d,2):#enumerate each subblock
                cnt+=1
                now=now*4+codeplane[int((i*d+m)/2),int((j*d+n)/2)]#change the current value of the byte
                if cnt==4:#if 4 operations have been done, write the byte into the file and initialise the cnt and now
                    file.write(np.uint8(now))
                    cnt=0
                    now=0
        if cnt!=0:#if finally, the cnt is not 0, it means that we have to add K bits to form a complete byte,
            now*=(4**(4-cnt))
            file.write(np.uint8(now))
file.close()
return ({'M':meanplane,'Sd':stdplane,'Idx':codeplane})
```

First, create an array header of size 6 to store the header of the file to fit the requirement below. Write the head into the file.

Byte(s)	No. of bits	Information
1	8	It specifies the header length in terms of number of bytes
2	8	It specifies the value of d, where $d \times d$ is the block size
3-4	16	It specifies the number of columns of blocks in the image
5-6	16	It specifies the number of rows of blocks in the image

Table 2 structure of the header

Next, enumerate each block and write the content of each block into the file. The content consists of three parts: The mean of the block, the standard deviation of the block and all the code words in this block. The format of the content should follow the instructions below.




```
file.write(meanplane[i,j])#write the mean of this block into the file
file.write(stdplane[i,j])#write the standard deviation of this block into the file
```

These two lines are used to write the mean and the standard deviation into the file.

```
cnt=0#used to count from 0 to 4 so that we can know when a byte will be generated
now=0#used to store the current value of the byte
for m in range(0,d,2):
    for n in range(0,d,2):#enumerate each subblock
        cnt+=1
        now=now*4+codeplane[int((i*d+m)/2),int((j*d+n)/2)]#change the current value of the byte
        if cnt==4:#if 4 operations have been done, write the byte into the file and initialise the cnt and now
            file.write(np.uint8(now))
            cnt=0
            now=0
if cnt!=0:#if finally, the cnt is not 0, it means that we have to add K bits to form a complete byte,
    now*=(4**(4-cnt))
    file.write(np.uint8(now))
```

This part is to write the code words into the file.

Due to the requirement, we have to store each code word in length of 2 bits. Hence, we have to convert every 4 code words into an 8-bit unsigned integer value. For example, if we have 4 code words 1,2,3,0 and we want to convert them into an 8-bit unsigned integer, what should we do? We can change them into binary to explain.

1(Decimal)=0000,0001(Binary)

2(Decimal)=0000,0010(Binary)

3(Decimal)=0000,0011(Binary)

0(Decimal)=0000,0000(Binary)

If we multiply the first code word 1 by 64. 1 will become 0100,0000 in binary. Then, multiply the second code word 2 by 16, 2 will become 0010,0000 in binary. Next, multiply the third code word 3 by 4, 3 will become 0000,1100 in binary. Leave the fourth code word 0 unchanged. Finally, add all the four Binary value together, we have 0110,1100. Then we successfully change the four integers into one 8-bit unsigned integer. In conclusion, multiply the first code word by 64, multiply the second code word by 16, multiply the third code word by 4 and leave the fourth code word unchanged. Add them all up and we will have a value in range [0,255]. Then we can realize use 2 bits to store a particular code word in the file.

Variable cnt is counting from 1 to 4 periodically. It indicates the position of the 2-bit code word in the whole byte.

Variable now is storing the calculation result.

```
now=now*4+codeplane[int((i*d+m)/2),int((j*d+n)/2)]#change the current value of the byte
```

Use this line to realize the above calculation.

```
if cnt==4:#if 4 operations have been done, write the byte into the file and initialise the cnt and now
    file.write(np.uint8(now))
    cnt=0
    now=0
```

This part in to check whether the 4 code words are all stored in the current byte. If so, write the current byte into the file and reset the variable cnt and now.

```
if cnt!=0:#if finally, the cnt is not 0, it means that we have to add K bits to form a complete byte,
    now*=(4**(4-cnt))
    file.write(np.uint8(now))
```

This part is to check, in the end, whether we should add k bits to fill the current byte. For example, after enumerating all subblocks, if cnt is 2, which means we have only two code words in the current byte, we have to add two code words 00 and 00 to fill the current byte.

5) Return the dictionary

As the requirement below, the decode function has to return a dictionary.

- The function should return a data structure that carries the information of the encoding output. The data structure is a dictionary containing 3 elements:
  - M : a 2D numpy array of size $\left(\frac{L_1}{d}\right) \times \left(\frac{L_2}{d}\right)$, each element of which provides the mean of a specific block.
  - Sd : a 2D numpy array of size $\left(\frac{L_1}{d}\right) \times \left(\frac{L_2}{d}\right)$, each element of which provides the standard deviation of a specific block.
  - Idx : a 2D numpy array of size $\left(\frac{L_1}{2}\right) \times \left(\frac{L_2}{2}\right)$, each element of which provides the index of a specific subblock.

This line is used to return the dictionary:

```
return ({'M':meanplane,'Sd':stdplane,'Idx':codeplane})
```

The dictionary:

```
{'M': array([[ 96,  64,  68, ...,  59,  60,  89],
       [ 75,  32,  40, ...,  28,  26,  62],
       [ 62,  37,  46, ...,  28,  28,  62],
       ...,
       [ 88,  61,  63, ...,   8,   8,  46],
       [ 91,  52,  64, ...,   7,   8,  51],
       [128, 103, 110, ...,  68,  70, 103]], dtype=uint8), 'Sd': array([[39, 34, 31, ..., 34, 34, 41],
       [35,  1,  4, ...,  1,  1, 34],
       [36,  6,  3, ...,  1,  2, 35],
       ...,
       [27,  5,  2, ...,  0,  0, 39],
       [33,  3,  4, ...,  0,  1, 39],
       [49, 50, 48, ..., 61, 61, 58]], dtype=uint8), 'Idx': array([[3, 1, 1, ..., 1, 1, 2],
       [3, 1, 2, ..., 0, 2, 2],
       [3, 2, 3, ..., 1, 0, 2],
       ...,
       [3, 1, 0, ..., 3, 0, 2],
       [3, 1, 3, ..., 0, 2, 2],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)}
```

3. Decode

The whole encode function:

```
def BVQCdecode(in_encoding_result_filename, out_reconstructed_image_filename):
    file=open(in_encoding_result_filename,"rb")
    header_len=file.read(1)[0]#read the header
    d=file.read(1)[0]
    no_of_block_cols=file.read(1)[0]+file.read(1)[0]*256
    no_of_block_rows=file.read(1)[0]+file.read(1)[0]*256
    OImg=np.zeros([no_of_block_rows*d,no_of_block_cols*d])#create an array to store the pixel values after decoding
    n=np.ceil(d/2*d/2*2/8)+2#n represents the number of bytes we have to read for each block
    for i in range(0,no_of_block_rows):
        for j in range(0,no_of_block_cols):#enumerate each block
            bfr=file.read(np.uint(n))#read one content for a block
            mean=bfr[0]#read the mean
            std=bfr[1]#read the standard deviation
            g0=max(0,mean-std)#calculate g0
            g1=min(255,mean+std)#calculate g1
            cnt=0#cnt is used to store how many indexes have been read
            for byte in bfr[2:]:
                temp=np.uint8(byte)
                for m in range(0,4):#deal with the four indexes in a byte
                    current=int (temp/(64/(4**m)))#get the (m+1)th index in a byte
                    temp-=64/(4**m)*current#modify twmp accordingly
                    now_row=int(cnt/(d/2))#based on the number of indexes that have been read, we can define the ir
                    now_col=int(cnt%(d/2))
                    #Based on the index, rebulid the block with according values in the code book
                    if(current==0):
                        OImg[i*d+2*now_row,j*d+2*now_col]=g0
                        OImg[i*d+2*now_row,j*d+2*now_col+1]=g0
                        OImg[i*d+2*now_row+1,j*d+2*now_col]=g1
                        OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g1
                    if(current==1):
                        OImg[i*d+2*now_row,j*d+2*now_col]=g1
                        OImg[i*d+2*now_row,j*d+2*now_col+1]=g1
                        OImg[i*d+2*now_row+1,j*d+2*now_col]=g0
                        OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g0
                    if(current==2):
                        OImg[i*d+2*now_row,j*d+2*now_col]=g0
                        OImg[i*d+2*now_row,j*d+2*now_col+1]=g1
                        OImg[i*d+2*now_row+1,j*d+2*now_col]=g0
                        OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g1
                    if(current==3):
                        OImg[i*d+2*now_row,j*d+2*now_col]=g1
                        OImg[i*d+2*now_row,j*d+2*now_col+1]=g0
                        OImg[i*d+2*now_row+1,j*d+2*now_col]=g1
                        OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g0
                    cnt+=1
                    if cnt==int(d/2*d/2):#This part is used to handle the case that stuffing bits exists.
                        break
    file.close()
    plt.imshow(OImg,cmap='gray')
    plt.show()#show the reconstructed image
    plt.imsave(out_reconstructed_image_filename,OImg,cmap='gray')  #save the image
    return OImg
```

1) Read the header

```
file=open(in_encoding_result_filename,"rb")
header_len=file.read(1)[0]#read the header
d=file.read(1)[0]
no_of_block_cols=file.read(1)[0]+file.read(1)[0]*256
no_of_block_rows=file.read(1)[0]+file.read(1)[0]*256
```

2) Preprocessing

```
OImg=np.zeros([no_of_block_rows*d,no_of_block_cols*d])#create an array
n=np.ceil(d/2*d/2*2/8)+2#n represents the number of bytes we have to re
```

Create an array OImg to store the pixel values of the reconstructed image.

Vairable n is used to store the number of bytes we have to read for each block. Firstly, 1 byte for mean and 1 byte for standard deviation. The rest bytes are for code words. In each block, we have (d/2)*(d/2) subblocks in total. Each subblock corresponds to a 2-bit code word in the file, hence, the bits for subblocks are (d/2)*(d/2)*2 in total and let (d/2)*(d/2)*2 divide by 8 to get the number of bytes for code words. Here, in case that the stuffing k bits occurs, we have to use np.ceil() to round up the result.

3) Decode the file

The whole code for this part:

```
def BVQCdecode(in_encoding_result_filename, out_reconstructed_image_filename):
    file=open(in_encoding_result_filename,"rb")
    header_len=file.read(1)[0]#read the header
    d=file.read(1)[0]
    no_of_block_cols=file.read(1)[0]+file.read(1)[0]*256
    no_of_block_rows=file.read(1)[0]+file.read(1)[0]*256
    OImg=np.zeros([no_of_block_rows*d,no_of_block_cols*d])#create an array to store the pixel valu
    n=np.ceil(d/2*d/2*2/8)+2#n represents the number of bytes we have to read for each block
    for i in range(0,no_of_block_rows):
        for j in range(0,no_of_block_cols):#enumerate each block
            bfr=file.read(np.uint(n))#read one content for a block
            mean=bfr[0]#read the mean
            std=bfr[1]#read the standard deviation
            g0=max(0,mean-std)#calculate g0
            g1=min(255,mean+std)#calculate g1
            cnt=0#cnt is used to store how many indexes have been read
            for byte in bfr[2:]:
                temp=np.uint8(byte)
                for m in range(0,4):#deal with the four indexes in a byte
                    current=int (temp/(64/(4**m)))#get the (m+1)th index in a byte
                    temp-=64/(4**m)*current#modify temp accordingly
                    now_row=int(cnt/(d/2))#based on the number of indexes that have been read, we
                    now_col=int(cnt%(d/2))
                    #Based on the index, rebulid the block with according values in the code book
                    if(current==0):
                        OImg[i*d+2*now_row,j*d+2*now_col]=g0
                        OImg[i*d+2*now_row,j*d+2*now_col+1]=g0
                        OImg[i*d+2*now_row+1,j*d+2*now_col]=g1
                        OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g1
                    if(current==1):
                        OImg[i*d+2*now_row,j*d+2*now_col]=g1
                        OImg[i*d+2*now_row,j*d+2*now_col+1]=g1
                        OImg[i*d+2*now_row+1,j*d+2*now_col]=g0
                        OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g0
                    if(current==2):
                        OImg[i*d+2*now_row,j*d+2*now_col]=g0
                        OImg[i*d+2*now_row,j*d+2*now_col+1]=g1
                        OImg[i*d+2*now_row+1,j*d+2*now_col]=g0
                        OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g1
                    if(current==3):
                        OImg[i*d+2*now_row,j*d+2*now_col]=g1
                        OImg[i*d+2*now_row,j*d+2*now_col+1]=g0
                        OImg[i*d+2*now_row+1,j*d+2*now_col]=g1
                        OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g0
                    cnt+=1
                    if cnt==int(d/2*d/2):#This part is used to handle the case that stuffing bits
                        break
    file.close()
    plt.imshow(OImg,cmap='gray')
    plt.show()#show the reconstructed image
    plt.imsave(out_reconstructed_image_filename,OImg,cmap='gray')  #save the image
    return OImg
```

```
bfr=file.read(np.uint(n))#read one content for a block
mean=bfr[0]#read the mean
std=bfr[1]#read the standard deviation
g0=max(0,mean-std)#calculate g0
g1=min(255,mean+std)#calculate g1
```

This part is to read a block's content and use the mean and standard deviation to calculate g0 and g1.

```
    cnt=0#cnt is used to store how many indexes have been read
    for byte in bfr[2:]:
        temp=np.uint8(byte)
        for m in range(0,4):#deal with the four indexes in a byte
            current=int (temp/(64/(4**m)))#get the (m+1)th index in a byte
            temp-=64/(4**m)*current#modify twmp accordingly
            now_row=int(cnt/(d/2))#based on the number of indexes that have bee
            now_col=int(cnt%(d/2))
            #Based on the index, rebulid the block with according values in the
            if(current==0):
                OImg[i*d+2*now_row,j*d+2*now_col]=g0
                OImg[i*d+2*now_row,j*d+2*now_col+1]=g0
                OImg[i*d+2*now_row+1,j*d+2*now_col]=g1
                OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g1
            if(current==1):
                OImg[i*d+2*now_row,j*d+2*now_col]=g1
                OImg[i*d+2*now_row,j*d+2*now_col+1]=g1
                OImg[i*d+2*now_row+1,j*d+2*now_col]=g0
                OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g0
            if(current==2):
                OImg[i*d+2*now_row,j*d+2*now_col]=g0
                OImg[i*d+2*now_row,j*d+2*now_col+1]=g1
                OImg[i*d+2*now_row+1,j*d+2*now_col]=g0
                OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g1
            if(current==3):
                OImg[i*d+2*now_row,j*d+2*now_col]=g1
                OImg[i*d+2*now_row,j*d+2*now_col+1]=g0
                OImg[i*d+2*now_row+1,j*d+2*now_col]=g1
                OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g0
            cnt+=1
            if cnt==int(d/2*d/2):#This part is used to handle the case that stu
                break
```

This part is used to read 2-bit code words in the file and based on the code word, rebuild the image.

Variable cnt is used to store how many code words have been read in order to calculate which pixels in the reconstructed image should be given the corresponding values.

For each byte storing the code words, use variable current to store the code word and variable temp to store the modified value of the byte.

Variables now_row and now_col represent which row and column the chosen code book should be in the block. Because for each row, there are d/2 subblocks, cnt/(d/2) should be the row number of the chosen code book and cnt%(d/2) should be the column number of the chosen code book.
In the following, based on the code word, reconstruct the image.

Finally, check whether cnt ==d/2*d/2. Because d/2*d/2 is the total number of subblocks in a block. We have to quit the loop for reading the code word as soon as we have read all the code words in this block to avoid reading the k stuffing bits.

4) **Display the reconstructed image and return the array storing the pixel values of the reconstructed image for evaluation.**

```
file.close()
plt.imshow(OImg,cmap='gray')
plt.show()#show the reconstructed image
plt.imsave(out_reconstructed_image_filename,OImg,cmap='gray')  #save the image
return OImg
```

4. Evaluation of the result

In the main program:

```
img=mping.imread(in_file)#This in_file stored the original pixel values of the image which is bounded in (0,1)
X=np.array(img)*255#Normalise the pixel values to (0,255) to calculate the error.
MSE,PPSNR=evaluate(X,OImg)    #Calculate the error.
print("MSE={}".format(MSE))
print("PPSNR={}".format(PPSNR))
```

Read the initial image and normalize the pixel values to [0,255].

The function called evaluate:

```
def evaluate(original,product):
    size=X.shape[0]*X.shape[1]
    ans=0
    for i in range(0,X.shape[0]):
        for j in range(0,X.shape[1]):
            ans+=(original[i][j]-product[i][j])**2#for each pixel in original image and the reconstructed image,calculate the error.
    mse=ans/size
    PPSNR=10*math.log(255*255/mse,10)
    return mse,PPSNR
```

Variable size is to calculate how many pixels need to be calculated.

When d=2, the results are as follows:

```
MSE=12.94866875
PPSNR=37.00855239882917
```

When d=4, the results are as follows:

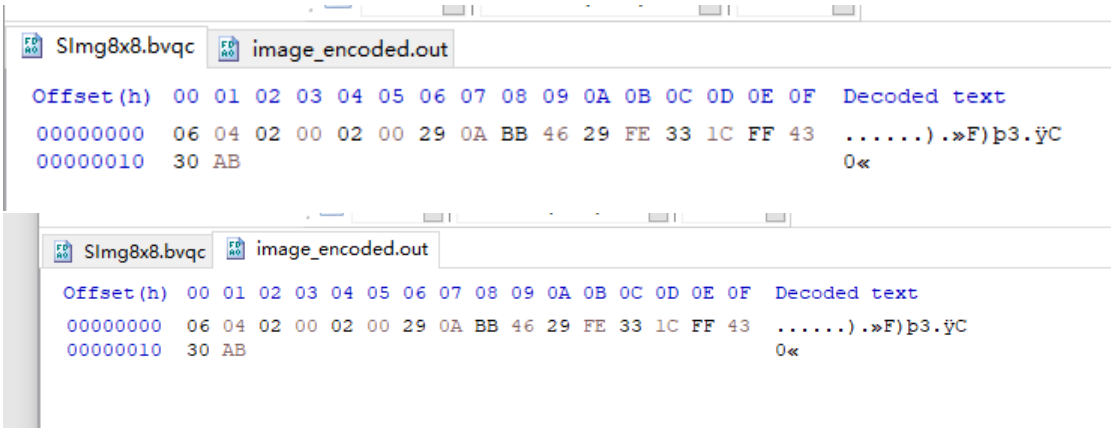
```
MSE=156.80988125
PPSNR=26.17706934935704
```

When d=8, the results are as follows:

```
MSE=402.32491875
PPSNR=22.085034286186193
```

3.Verification

For the encode results, use a hex editor to verify the result.



The encode result is the same as the one in the hints sheet.

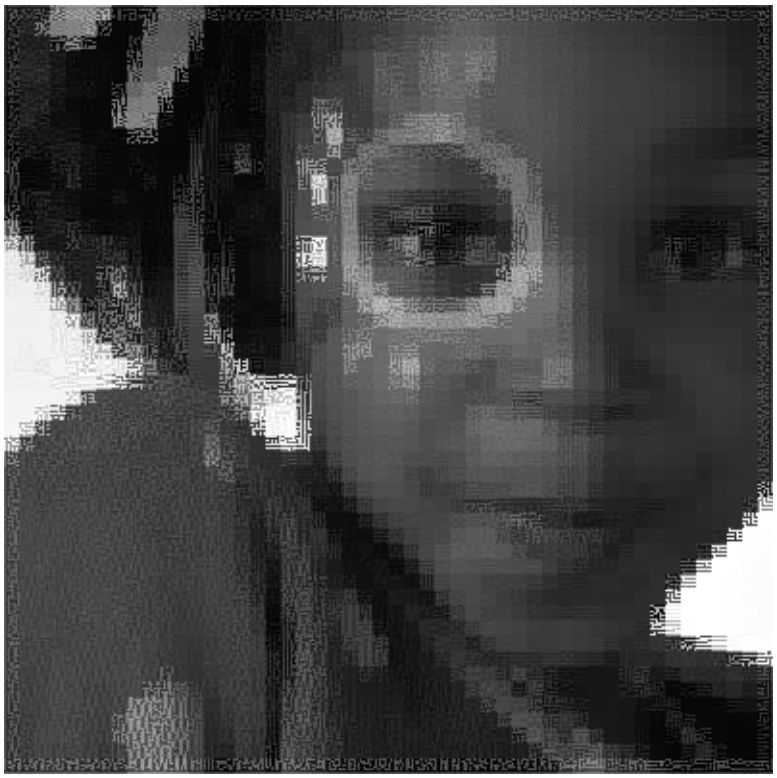
For the final result verification, compare the decode image to the original figure.

The initial image:

When d=2, the reconstructed image:

When d=4, the reconstructed image:

When d=8, the reconstructed image:



4.Code

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mping
import os
import math
def readfile():
    while(1):#The program will loop until a valid input is received.
        filename=input("Please input the file name of an image:\n")
        try:#Use try...except... to handle the exceptions in case that the file does not exist.
            img=mping.imread(filename)
        except IOError:
            print("<{}> does not exist!".format(filename))
        else:
            return filename
def readd():
    while(1):#The program will loop until a valid input is received.
        d=input("Please input the value of d:\n")
        if(d==''):#If null string is input, return the default value 4 of d.
```

```
        return 4
    if(int(d)>1):#check whether d is an integer power of 2
        i=1
        while(i<=int(d)):
            if(i==int(d)):
                return int(d)
            i*=2
    print("d must be an integer power of 2!\n")
def codeword(a,g0,g1):
  minn=(a[0,0]-g0)**2+(a[0,1]-g0)**2+(a[1,0]-g1)**2+(a[1,1]-g1)**2#Calculate the distance between the subblock and codewords C0, then give this value to the variable minn to record the current min value.
  code=0#Initial the codeword to be chosen to codeword C0
  count1=(a[0,0]-g1)**2+(a[0,1]-g1)**2+(a[1,0]-g0)**2+(a[1,1]-g0)**2#Calculate the distance between the subblock and codewords C1.
  if(count1<minn):#If this distance is less than minn, change the minn and code accordingly.
    minn=count1
    code=1
  count2=(a[0,0]-g0)**2+(a[0,1]-g1)**2+(a[1,0]-g0)**2+(a[1,1]-g1)**2#Calculate the distance between the subblock and codewords C2.
  if(count2<minn):#If this distance is less than minn, change the minn and code accordingly.
    minn=count2
    code=2
  count3=(a[0,0]-g1)**2+(a[0,1]-g0)**2+(a[1,0]-g1)**2+(a[1,1]-g0)**2#Calculate the distance between the subblock and codewords C3.
  if(count3<minn):#If this distance is less than minn, change the minn and code accordingly.
    minn=count3
    code=3
  return code

def BVQCencode(in_image_filename,out_encoding_result_filename,d):
  img=mping.imread(in_image_filename)#read the image
  plt.imshow(img,cmap='gray')
  plt.show()#show the image
  X=np.array(img)*255#Normalise the pixel values to (0,255).
  rows_of_blocks=int(X.shape[0]/d)#Calculate the number of rows for blocks
  cols_of_blocks=int(X.shape[1]/d)#Calculate the number of columns for blocks
  meanplane=np.zeros([rows_of_blocks,cols_of_blocks],dtype="uint8")#create an array to store the mean of each block
  stdplane=np.zeros([rows_of_blocks,cols_of_blocks],dtype="uint8")#create an array to store the standard deviation of each block
  codeplane=np.zeros([int(X.shape[0]/2),int(X.shape[1]/2)],"uint8")#create an array to store the code words
  for i in range(0,X.shape[0],d):
    for j in range(0,X.shape[1],d):#enumerate each block
```

```
        block=X[i:i+d,j:j+d]#get the block from the normalised image
        miu=np.mean(block)#calculate the mean of the current block
        std=np.std(block)#calculate the standard deviation of the current block
        meanplane[int(i/d),int(j/d)]=np.uint8(round(miu))#store the mean value of this block to the appropriate position of the meanplane
        stdplane[int(i/d),int(j/d)]=np.uint8(round(std))#store the standard deviation value of this block to the appropriate position of the stdplane
        g0=max(0,miu-std)#compute g0
        g1=min(255,miu+std)#compute g1
        for m in range(0,d,2):
          for n in range(0,d,2): #enumerate each subblock in the current block
            code=codeword(X[i+m:i+m+2,j+n:j+n+2],g0,g1)#get the code for the current subblock
            codeplane[int((i+m)/2),int((j+n)/2)]=np.uint8(code)##store the code of the subblock to the appropriate position of the codeplane
    file=open(out_encoding_result_filename,"wb")
    header=np.zeros([6],dtype='uint8')#write the header
    header[0]=np.uint8(6)
    header[1]=np.uint8(d)
    header[2]=np.uint8(cols_of_blocks%256)
    header[3]=np.uint8(cols_of_blocks/256)
    header[4]=np.uint8(rows_of_blocks%256)
    header[5]=np.uint8(rows_of_blocks/256)
    for byte in header:
      file.write(byte)
    for i in range(0,rows_of_blocks):
      for j in range(0,cols_of_blocks):#enumerate each block
        file.write(meanplane[i,j])#write the mean of this block into the file
        file.write(stdplane[i,j])#write the standard deviation of this block into the file
        cnt=0#used to count from 0 to 4 so that we can know when a byte will be generated
        now=0#used to store the current value of the byte
        for m in range(0,d,2):
          for n in range(0,d,2):#enumerate each subblock
            cnt+=1
            now=now*4+codeplane[int((i*d+m)/2),int((j*d+n)/2)]#change the current value of the byte
            if cnt==4:#if 4 operations have been done, write the byte into the file and initialise the cnt and now
              file.write(np.uint8(now))
              cnt=0
              now=0
        if cnt!=0:#if finally, the cnt is not 0, it means that we have to add K bits to form a complete byte,
          now*=(4**(4-cnt))
          file.write(np.uint8(now))
    file.close()
    return ({'M':meanplane,'Sd':stdplane,'Idx':codeplane})
```

```
def BVQCdecode(in_encoding_result_filename, out_reconstructed_image_filename):
  file=open(in_encoding_result_filename,"rb")
  header_len=file.read(1)[0]#read the header
  d=file.read(1)[0]
  no_of_block_cols=file.read(1)[0]+file.read(1)[0]*256
  no_of_block_rows=file.read(1)[0]+file.read(1)[0]*256
  OImg=np.zeros([no_of_block_rows*d,no_of_block_cols*d])#create an array to store the pixel values after decoding
  n=np.ceil(d/2*d/2*2/8)+2#n represents the number of bytes we have to read for each block
  for i in range(0,no_of_block_rows):
    for j in range(0,no_of_block_cols):#enumerate each block
      bfr=file.read(np.uint(n))#read one content for a block
      mean=bfr[0]#read the mean
      std=bfr[1]#read the standard deviation
      g0=max(0,mean-std)#calculate g0
      g1=min(255,mean+std)#calculate g1
      cnt=0#cnt is used to store how many indexes have been read
      for byte in bfr[2:]:
        temp=np.uint8(byte)
        for m in range(0,4):#deal with the four indexes in a byte
          current=int (temp/(64/(4**m)))#get the (m+1)th index in a byte
          temp-=64/(4**m)*current#modify twmp accordingly
          now_row=int(cnt/(d/2))#based on the number of indexes that have been read, we can define the index is in which row and column in the codeplane
          now_col=int(cnt%(d/2))
          #Based on the index, rebulid the block with according values in the code book
          if(current==0):
            OImg[i*d+2*now_row,j*d+2*now_col]=g0
            OImg[i*d+2*now_row,j*d+2*now_col+1]=g0
            OImg[i*d+2*now_row+1,j*d+2*now_col]=g1
            OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g1
          if(current==1):
            OImg[i*d+2*now_row,j*d+2*now_col]=g1
            OImg[i*d+2*now_row,j*d+2*now_col+1]=g1
            OImg[i*d+2*now_row+1,j*d+2*now_col]=g0
            OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g0
          if(current==2):
            OImg[i*d+2*now_row,j*d+2*now_col]=g0
            OImg[i*d+2*now_row,j*d+2*now_col+1]=g1
            OImg[i*d+2*now_row+1,j*d+2*now_col]=g0
            OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g1
          if(current==3):
            OImg[i*d+2*now_row,j*d+2*now_col]=g1
            OImg[i*d+2*now_row,j*d+2*now_col+1]=g0
            OImg[i*d+2*now_row+1,j*d+2*now_col]=g1
```

```
                OImg[i*d+2*now_row+1,j*d+2*now_col+1]=g0
            cnt+=1
            if cnt==int(d/2*d/2):#This part is used to handle the case that stuffing bits exists.
                break
    file.close()
    plt.imshow(OImg,cmap='gray')
    plt.show()#show the reconstructed image
    plt.imsave(out_reconstructed_image_filename,OImg,cmap='gray')      #save the image
    return OImg
def evaluate(original,product):
    size=X.shape[0]*X.shape[1]
    ans=0
    for i in range(0,X.shape[0]):
        for j in range(0,X.shape[1]):
            ans+=(original[i][j]-product[i][j])**2#for each pixel in original image and the reconstructed image,calculate the error.
    mse=ans/size
    PPSNR=10*math.log(255*255/mse,10)
    return mse,PPSNR



in_file=readfile()#Read the file.
out_file="image_encoded.out"#The encoded image is stored in this file.
d=readd()#Store d.
result=BVQCencode(in_file,out_file,d)#  Do the encode and the returned dictionary is stored in variable result.
print(result)
file_in="image_encoded.out"#Change the input file and output file name to do the decode.
file_out="image_decoded.png"
OImg=BVQCdecode(file_in,file_out) # OImg is an array store pixel values of the decoded image.
img=mping.imread(in_file)#This in_file stored the original pixel values of the image which is bounded in (0,1).
X=np.array(img)*255#Normalise the pixel values to (0,255) to calculate the error.
MSE,PPSNR=evaluate(X,OImg)    #Calculate the error.
print("MSE={}".format(MSE))
print("PPSNR={}".format(PPSNR))
```