Laboratory Exercise 3

Multimedia Communications over IP Networks

Multimedia and Video Communications (SSY150)

Prepared by: Irene Y.H. Gu, irenegu@chalmers.se
Department of Electrical Engineering,
Chalmers University of Technology,
41296, Göteborg, Sweden

Updated in April, 2018

1 Introduction: Basic Building Blocks in an IP-based Image/Video Communication System

In this laboratory exercise, we shall address the issue of multimedia communications through IP networks [1, 2, 3], the multimedia data under our study is a simple 2D raw image, and we shall make a case study on transporting compressed image over IP networks, where the network is prune to packet loss errors. Our aim here is to gain first-hand experience by a hands-on learning approach. To limit the scope of this laboratory work, we only consider (wired) IP networks rather than wireless or other wired networks, where the steps of modulation and demodulation are required. Further, we only consider erasure channels in the network layer that are mainly characterized by packet loss (e.g. using RTP/UDP). Other types of noise from channels connected to the physical layer in a communication system, e.g. AWGN (Additive White Gaussian Noise), Releigh and Racian fading channels in wireless communications, mainly cause bit errors. We shall try to build up a simplest communication system and study the erasure channel noise impact to the reconstructed multimedia data at the receiver side. The building blocks of the system to be studied in this laboratory exercise are depicted in Figure 4. The grey boxes "modulation" and "demodulation" in Figure 4 are not required in this laboratory work as we only consider IP networks. Further, the grey box "error concealment" is not included in this laboratory work, in order to limit the scope.

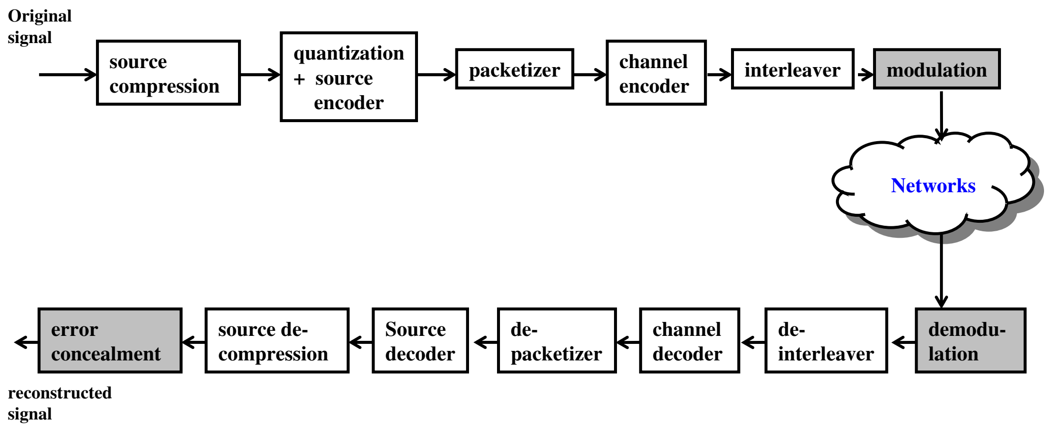

Figure 1: Block diagram of end-to-end video communications over IP networks

In the following, we shall briefly describe the function of each block in a more practical way, especially issues related to implementation work. Details of theories are beyond the scope of this laboratory handout material. Students should study the relevant course books and materials for more details [1, 2, 3].

Block-1: Block-based 2D DCTs and Lossy Image Compression

This issue has already been handled in the second Lab. exercise [4]. Therefore, only a brief description shall be given. More details can be found in other reading materials [4, 5, 6]. For a given 2D still image $I(x, y)$, it is first partitioned into non-overlapped blocks, each of size $M \times M$ (typical value of M is 8 or 16). Once each block of data is 2D DCT transformed, a zigzag scanning is then applied to DCT coefficients resulting a 1D sequence of coefficients from low frequencies to high frequencies, as shown in Figure 2.

Since the zigzag scanned sequence of DCT coefficients are related to DCT coefficients from low to high frequencies, these coefficients in the sequence have a general tendency of diminishing magnitude values.

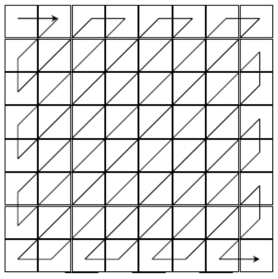

Figure 2: Zigzag scanning of DCT coefficients in 2D space to yield an 1D sequence.

A common way of compression is to apply a threshold to such a 1D sequence and to truncate the data when the first coefficient magnitude value drops below the threshold.
The above process is repeated for all blocks, resulting in a long 1D sequence containing the DCT coefficients after truncating small value coefficients. This step belongs to one part of the source coding which achieves lossy image compression.

Block-2: Scalar Quantization and Source Encoding

Once the image is compressed, the values of coefficients require to be quantized and encoded. Quantization can be performed by a scalar quantizer or a vector quantizer. A scalar quantizer converts each symbol into a nearest quantized grid, and is a rather straightforward process. In vector quantization, each vector of source symbols (a vector of DCT coefficients in this case) is converted into a source code from a codebook. The vector quantization is generally more efficient while the scalar quantization is easier to implement. For the sake of simplicity for this experiment, where the emphasis in on the study of packet loss and erasure channel behavior, we choose to use a scalar quantizer. Further, each DCT coefficient (alternatively, the difference of DCT coefficients) is quantized to $2^m = 256$ levels (or, $m = 8$ bits per symbol).
After quantization, a source encoding scheme is usually applied to further compress the data. Typically used source encoders are the Huffman encoder and the arithmetic encoder, both having variable code lengths. The average length of a Huffman code depends on the statistical frequency with which the source produces each symbol from its alphabet. A Huffman code dictionary, which associates each data symbol with a codeword, has the property that no codeword in the dictionary is a prefix of any other codeword in the dictionary. Arithmetic coding is often used for data sources having a small alphabet. The length of an arithmetic code, instead of being fixed relative to the number of symbols being encoded, depends on the statistical frequency with which the source produces each symbol from its alphabet. For long sequences from sources having skewed distributions and small alphabets, arithmetic coding compresses better than Huffman coding. To simplify this experiment, the source encoding and decoding steps are removed from our simulated system.

Block-3: Packetization

In the packet switching networks, packets are sent through the networks. Packetization is usually done in several levels, for example, source symbols are first packetized before applying channel encoding (also referred to as the 'source packetization'). In each level of the communication system, a packet is encapsulated into a larger packet where a new head is inserted (the data in a packet is usually referred to as the "payload"). Re-packetization may also be performed in several more stages, e.g., in an interleaver (for 'intermediate packetization'), and in the transport layer (referred to as the 'transport packetization'). Here we consider the source packetization process. In setting the packet size, one should consider the fragmentation limit of intermediate nodes in IP networks. In the Internet, the maximum transfer unit or the maximum packet size that can be transmitted without split or recombination is equal to 1500 bytes in the transport and network layers. For wireless networks, the maximum transfer unit is usually much smaller, and 100 bytes are commonly used by researchers. In this experiment, source packetization is applied, which partitions the source encoded data into blocks before a channel encoding scheme is applied. The strategy for such a packetization scheme is to make each packet as independent as possible. Hence, if a packet is lost during the transmission it will only affect a local image region instead of possibly introducing severe global impact. For example, one may choose a GOB (group of blocks) or an MB (macro block) as a packet. For packetization of video, an MB usually includes the differentially encoded motion vector and DCT coefficients in a block. In our experiment, the motion vector is excluded in the MBs since only 2D images are considered. For the sake of simplicity in this laboratory work, we also choose each packet such that it only contains one block of DCT coefficients (i.e., the quantized 1D zigzag scanned and truncated sequence from an individual DCT transformed block).

Block-4: Channel/network Encoding

For robustness to packet losses in the IP based networks, one essential step is to add forward error correction (FEC) in the network layer for protecting packets. From our previous knowledge, we know that channel encoding adds redundancy to the source encoded symbols. Thereby, errors can be detected and corrected afterwards. For channel encoding of source data packets, linear block coding methods rather than convolutional coding methods are more suitable because of the requirement of handling individual packets independently. Among the block coding methods, Reed-Solomon (RS) coding is one of the most frequently used methods for bursty packet loss networks.
RS codes belong to the BCH code family. A RS code is denoted by RS(n, k), where n is the codeword length, k is the number of source symbols in a codeword (each symbol is of m bits). Hence, the number of parity symbols is $(n-k)$. RS codes are based on Galois field (GF), where a codeword from GF(2^m) (for the binary case) has a length of $2^m - 1$. A RS code corresponding to that message is an n-column Galois field array in GF(2^m). The codeword length n must be between 3 and $2^m - 1$. For a k-source symbol packet and n codeword length, the RS codes can correct a maximum of $\lfloor (n-k)/2 \rfloor$ symbol errors, summarized in Table 1. For a RS(n, k) code, the channel coding rate is equal to $r_c = k/n$. Clearly, selecting coding

notation	definition
n	number of symbols per codeword
k	number of symbols per message
m	number of bits per symbol
$t = \lfloor (n-k)/2 \rfloor$	maximum error correcting capability

Table 1: Definition of notations in the RS codec

rate should be a tradeoff between bandwidth, coding efficiency, and robustness to channel errors (hence, the image quality at the receiver side). Further details and reading materials on channel coding methods and RS codes can be found in [7, 8, 9].

Block-5: Interleaving

Interleaving is a commonly used scheme in communications, especially for packet switching networks. In a typical way of interleaving for block codes (e.g. RS codes), L codewords are arranged in the rows of a matrix, then each column of the matrix is sent as a new packet to the network. In the receiver side, an inverse process called de-interleaving is added. If a packet is lost, the error will appear as an error symbol in all these L packets due to the interleaving. Without interleaving, one entire packet or codeword is corrupted. Since a RS decoder can maximally correct $\lfloor (n-k)/2 \rfloor$ symbol errors, it is much more efficient to design a RS codec if an interleaver is used. The disadvantage of using an interleaver is the extra delay and buffer required to temporally store L codewords.

IP Networks: Erasure Channel Modeling

A main problem of sending packets through IP networks for multimedia data using RTP/UDP is the packet loss. A communication system with packet loss in the network layer is usually called "erasure channels". Packet loss can be caused by, e.g., network congestion with buffer overflow, long delay in delivery packets, and many more. One of the commonly used models for packet erasures in the network layer is the two-state Markov model [3], containing a good state S_{Good} (a packet is received) and a bad state S_{Bad} (a packet is lost), as shown in Figure 3. Under the model, packet loss in an erasure channel can be described by the probability of packet losses P_B and the average bursty length L_B described as follows:

$$P_B = \frac{P_{GB}}{P_{GB} + P_{BG}}, \quad L_B = \frac{1}{P_{BG}}$$

where P_B is the probability for the bad state, P_G the probability for the good state, P_{GB} is the transition probability from the good state to the bad state, and P_{BG} the transition probability from the bad to the good state. By setting the packet loss probability and average bursty length one can simulate the bursty packet loss networks, and then observe the impact to the reconstructed image in the receiver side with/without channel coding and with/without interleaving schemes.

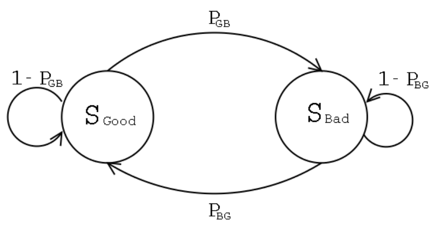

Figure 3: A two-state Markov model for modeling packet erasures in the network layer or link layer.

Block-6: Deinterleaving

This process is an inverse process of interleaving in Block-5. In the deinterleaver, each L received packets are arranged as the columns of a matrix, and then each row from this matrix is used as a received codeword.

Block-7: Channel/network decoding

This is the inverse process of channel encoding, where a channel codeword is decoded, in the block coding case, to a k source symbols with possible errors corrected (under the limit of a maximum $\lfloor (n-k)/2 \rfloor$ symbol errors for a RS code).

Block-8 and Block-9: Depacketization and Recovering DCT Coefficients by Lookup Quantization Codebook

The depacketization is a reverse process of packetization (Block-3) . Recall that channel decoding results in source symbols where each is a quantized value (a binary sequence) ranging from 0 to $2^k - 1$. The quantized value is thus required to be converted back to a quantized DCT coefficient value. This can be done by using a quantization codebook, which associates each quantized value to a DCT coefficient value. For example, for a 8 quantization level codebook, each quantized value has a code within the set $\{000, 001, ...111\}$, relating to the DCT coefficient value $c_{i,j}$ in the range of $[c_{min}, c_{man}]$. Therefore, an inverse process to the quantization (Block-2) shall be applied, where the conversion can be done by lookup codebook.

Block-10: Block-based Inverse 2D-DCTs and Image Decompression

This step is the reverse processing of Block-1. For each packet of DCT coefficients obtained from Block-10, zeros are padded to each 1D DCT sequence (recall that small DCT coefficients are truncated during the compression in Block-1) to reach the full length of $M \times M$ (where M is the DCT block size). Then the inverse zigzag process is applied to put the 1D sequence into 2D space. After that, an inverse 2D DCT is applied to the block of decoded DCT coefficients. This process is repeated for all packets (hence, all blocks), resulting in a reconstructed image in the receiver side of the communication system.

2 Implementation Issues: Matlab Functions

(a) For implementing these blocks, we need to use some functions in the Matlab Communication toolbox. For scalar quantization, the Matlab function **quantiz** can be used, for example:

$$\begin{array}{l} \text{samples} = [-2.4, -1, -.2, 0, .2, 1, 1.2, 1.9, 2, 2.9, 3, 3.5, 5]; \\ \text{codebook} = [-2.5 : 0.5 : 5.5]; \\ \text{partition} = [-2.5 : 0.5 : 5.0]; \\ [\text{index, quantized}] = \text{quantiz(samples,partition,codebook)}; \end{array}$$

(b) For RS encoding and decoding, the Matlab functions **rsenc** and **rsdec** can be used, e.g.:

$$\begin{array}{lll} n = 7;\ k = 3; & \% & \text{Codeword length and message length;} \\ m = 3; & \% & \text{Number of bits in each symbol;} \\ \text{msg} = gf([1\ 6\ 4;\ 0\ 4\ 3], m); & \% & \text{Message is a Galois array;} \\ \text{code} = rsenc(\text{msg}, n, k); & \% & \text{Codes will be a Galois field;} \\ \text{codeword} = \text{code}.x; & \% & \text{Extract codewords from a Galois array with a Matlab class } \textit{gf}. \end{array}$$

(c) For interleaving, there exists a Matlab function **matintrlv** for matrix interleaver which fills a matrix with data elements row by row and then sends the matrix contents to the output column by column. You may also make the short Matlab codes yourself.
More details can be found from using help functions in Matlab Communications toolbox and the introductory examples therein.

References

[1] Ming-Ting Sun and Amy R. Reibman, ”Compressed video over IP networks”, Marcel Dekker, Inc., 2001.

[2] Abdul H. Sadka, Compressed video communications, John Wiley & Sons, Ltd, 2002.

[3] Fai Zhai, Aggoelos Katsaggelos, Joint Source-Channel Video Transmission, synthesis lectures on image, video and multimedia processing, Morgan & Claypool Publishers, USA, 2007.

[4] Irene Gu, the handout of the Laboratory Exercise 2 ”2D Image Compression using Transforms and Subband Filters” in the Multimedia and Video Communications course (SSY150), Chalmers University of Technology, Sweden, 2008.

[5] Iain Richardson and Iain E. G. Richardson, H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia, John Wiley & Sons, Ltd, 2003.

[6] M. Ghanbari, Standard Codecs: Image Compression to Advanced Video Coding, IEE, 2003.

[7] Iriving S. Reed and Xuemin Chen, Error-Control Coding for Data Networks, Kluwer Academic Publishers, 1999.

[8] S.Lin and D.J.Costello, Jr. Error Control Coding: Fundamentals and Applications, Prentice Hall. Inc., 1983.

[9] User's Guide for Matlab Communications toolbox, The MathWorks, Inc.

1. Laboratory Exercises

This laboratory exercise is aimed at hands-on learning on multimedia communication over IP networks. One of the main aims is to study the impact of an erasure channel (in the network layer) to the reconstructed signal in the receiver side, in addition to the conventional bit errors in the channel connected to the physical layer. Another aim is to gain first hand experience through studying a sample image communication system, as shown in Figure 4. For the sake of convenience the figure is included below again. This is done by making your own Matlab programs and doing computer simulations. The tests are done by adding more more pairs of building blocks at each step (i.e. forward and inverse blocks, e.g. packetization and depacketization, encoding and decoding, etc), followed by evaluating the correctness of the functions performed by the Matlab codes.

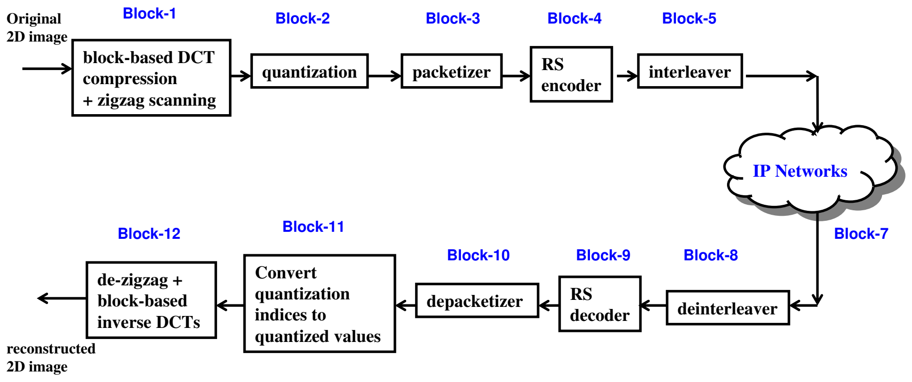

Figure 4: Block diagram of end-to-end image communications over IP networks.

Summary of the Tasks

Task 1: Building each block of the sample communication systems in Figure 4 by making your Matlab codes.

Task 2: Test the impact of packet losses to the received signal, under the erasure channel model in Figure 3.

Form of Laboratory Exercise, Report and Assessment

This laboratory exercise should be done by each group (maximum 2 students per group). After the laboratory exercise, each group will submit *one* short written report by the dead line (this can be found in the course web-page). The report shall include:

- Results obtained from this laboratory exercise (as specified below);
- the MATLAB program codes you wrote and used (each step shall be clearly marked in the codes);
- A zipped file containing the resulting images from the receiver side, related to different network settings;
- Discussions and interpretations of your results.

Further, each student will attend an oral questioning/discussion session with the course teacher for the individual assessment.

Exercises

Task-1: Scalar quantization

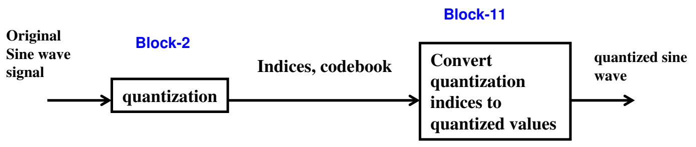
Original Sine wave signal
Block-2
quantization
Indices, codebook
Block-11
Convert quantization indices to quantized values
quantized sine wave

Figure 5: Task 1

This task builds up Matlab codes to perform the blocks shown in Figure 5, and then verifies the correctness of the codes.

Step 1.1: Generate a synthetic signal
First, we shall generate a synthetic continuous valued sine wave signal, e.g. by the following Matlab codes:

$t = [0 : .1 : 10 * pi];$ % Times at which to sample the sine function
$signal = sin(t);$ % Original signal, a sine wave

This sine wave signal will be used for the time being as the input to the quantizer. Plot the signal.

Step 1.2: Implement Block-2
Apply the scalar quantization to this signal: *each signal sample is quantized to m=8 bits (or 256 levels) with a uniform quantization step size.* Plot the original signal overlapped with the quantized signal values in one figure.

Hint: use Matlab function *quantiz*. The outputs contain the indices and quantized values of the signal. Type help quantiz for details.

Step 1.3: Implement Block-11
This step is to make the Matlab codes to implement Block-11 in the above figure. The input is the sequence of quantized indices and codebook from Step 1.2 (or, Block-2), and the output is the sequence of quantized values.

Step 1.4: Verification
Connect Block-2 and Block-11, and verify the correctness of your Matlab codes for these two blocks. Plotting the original and quantized signal from the output of Block-11 in the same figure.

Task 2: Packetization and depacketization

This task builds up Matlab code to perform the two white blocks (Block-3 and Block-10) shown in Figure 6, which are inserted between Block-2 and Block-11 generated in Task-1. Packets obtained from this step are 'source packets' which are usually different from the packets in the network layer. Further, verification of the added blocks will be performed.

Step 2.1: implement packetization in Block-3
Choose the size of source packets as $k = 127$ (i.e. 127 symbols per packet), and divide the 1D source symbol sequence into packets (*use the signal indices from the scalar quantization!*). Make a Matlab program to arrange these packets sequentially by rows in an array, where the number of rows is

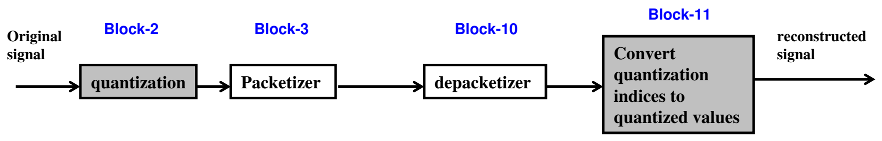

Figure 6: Task 2

equal to the total number of packets.

Step 2.2: implement depacketization in Block-10
Make a Matlab program that performs the inverse process of packetization in Step 2.1: That is, from a given 2D array where each row is a packet (containing k source symbols), convert the 2D array into a 1D sequence of source symbols.

Step 2.3: Verification
Insert packetization (Block-3) and depacketization (Block-10) into the system as shown in Figure 6. Verify the correctness of these 2 blocks by plotting the output quantized signal and the original signal in the same figure, and comparing the two curves.

Task 3: Reed-Solomon (RS) encoding and decoding

This task builds up Matlab codes to perform RS encoding in Block-4 and RS decoding in Block-9 (as shown in the two white boxes of Figure 7), and then verify the correctness of your Matlab program.

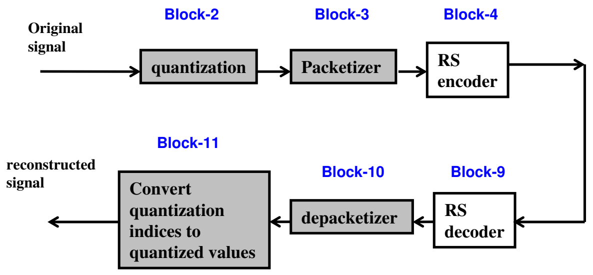

Figure 7: Task 3

Step 3.1: Implement RS encoding in Block-4
In the output from the packetizer (Block-3), which contains an 2D array, each row is a message (or packet) containing k source symbols. The task is now to apply a RS encoder to these packets. One can use the Matlab function *rsenc* to perform this. Choose the length of RS(n,k) codeword as $n = 2^m - 1$, where $m = 8$ is also equal to the bits for each source symbol. A RS code with a n codeword length, for an input source code of k symbols (also equal to the packet size in this case) can correct up to $\lfloor (n-k)/2 \rfloor$ symbol errors, and detect $(n-k)$ errors.
To extract the codewords in rows from the RS encoder performed by the Matlab function *rsenc* (which generates a Galois array of codes), use *code.x*.

Hint: use the Matlab function *rsenc*. An example of RS encoding is as follows, where we assume that the packet array is named as 'data', where each row contains a source message word of k symbols (a source packet), each symbol is represented by m bits:

$m = 8;$	%bits per symbol
$n = 2^m - 1; k = 127;$	%codeword length and message length
$msgwords = gf(data, m);$	%represent data by using a Galois array
$[codes, cnumerr] = rsenc(msgwords, n, k);$	%perform RS encoder
$codewords = codes.x$	%extract rows of codewords from the GF array

Step 3.2: Implement RS decoding in Block-9
Apply the RS decoder using the array of codewords from the RS encoder output as the input. Assuming the decoded message words as 'dec_msg', check the correctness by using the Matlab codes:
isequal(dec_msg,msgwords)
Hint: use the Matlab function *rsdec*. Type Matlab help for more details.

Step 3.3. Verification
Connect all the gray-shaded blocks shown in Figure 8, and verify the correctness of the 2 blocks (RS encoder and decoder) by plotting the output quantized signal and the original signal in the same figure, and comparing the two curves.

Task 4: Image compression by block-based 2D DCT, zigzag to 1D source symbol sequence; and its reverse process

This task builds up Matlab code to perform the two white blocks (Block-1 and Block-12) and then inserts these two blocks into the system shown in Figure 8. Verification of the Matlab codes for implementing these two blocks will then be performed.

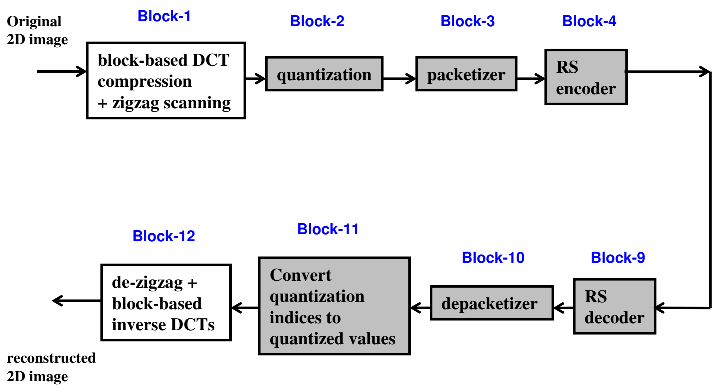

Figure 8: Task 4

Step 4.1: Load a 2D image
Load a 2D image into the Matlab (a small 2D image 'lenna' can be downloaded from the course website). Convert the image format to intensity image where the range of image values is in [0.0,1.0]. (hint: use the Matlab function 'mat2gray'), and display it.

Step 4.2: Set a compression ratio
For the sake of simplicity, compression will be performed by removing the same number of coefficients from each DCT transformed block. Therefore, for a given compression ratio $R_{\text{compression}}$, the number of coefficients removed in each N=16x16 block is approximately equal to $N_1 = round(r * 16^2)$. Set the compression ratio to $R_{\text{compression}}$=0.5.

Step 4.3: Implement Block-1

Step 4.3.1: Block-based 2D DCTs

Repeat the process in Lab.2: divide the image into blocks, each is of size 16 $\times$ 16 (pixels), apply the 2D DCT to each block and insert the DCT coefficients into the corresponding 2D positions.

Step 4.3.2: Zigzag scanning DCT coefficients

Performing zigzag scanning to DCT coefficients in each block to obtain a 1D sequence (see Figure 2 for zigzag scanning), *only the first $N_c = N - N_1$) DCT coefficients in each block are included in the scanned sequence*. Repeat this step for all blocks (DCT blocks are scanned sequentially and row by row), and put scanned data from different blocks sequentially according to the order they appear, and this results in one 1D sequence.

Step 4.4: Implement Block-12

Step 4.4.1: Inverse process of zigzag scanning DCT

This step is the exact inverse process of Step 4.3.2. Each N_c DCT coefficients in the 1D sequence are put back sequentially to a corresponding 2D block, also with N_1 zeros added for those removed DCT coefficients from the compression process in Step 4.3.2. This process is repeated for each N_c coefficients in the 1D sequence, and the corresponding blocks are in the same order as performed in Step 4.3.2. This results in decoded block-based DCT coefficients.

By temporarily setting N_1=0, and combining Step 4.2.2 (zigzag scanning) and Step 4.4.1 (the inverse of zigzag), one should obtain an image with exactly the same DCT coefficients as the input image to Step 4.3.2. Check and verify this to be sure that the Matlab codes for the zigzag scanning is bug free.

Step 4.4.2: Block-based inverse 2D DCTs

This process is the exact inverse process of Step 4.3.1. For each N=16x16 block, 2D inverse DCT is performed. The process is repeated over all blocks, results in a reconstructed image at the receiver side.

To test whether block-based DCT is bug free, perform block-based DCTs in Step 4.3.1 immediately followed by block-based inverse DCTs in Step 4.4.2. Check and verify whether the resulting image is exactly the same as the original image, which indicates that the Matlab codes in this part are bug free.

Step 4.5: Verification

Plug the Matlab codes for Block-1 (in Step 4.3) and Block-12 (in Step 4.4) into the system shown in Figure 8, and verify whether the reconstructed image is exactly the same as the original image (errors are allowed up to the scale of 1/2 quantization level). If the above is satisfied, the Matlab program for newly inserted blocks can be considered as bug free.

Task 5: noise: bit errors and packet losses

This task builds up Matlab codes to perform the white block (Block-7), and then inserts this block into the system shown in Figure 9. Then the Matlab codes for implementing the added block will be verified. Two cases are considered: In the 1st case, bit errors are added by introducing a few symbol errors in each codeword according to a given error probability. In the 2nd case, either an entire "transport packet" is lost, or no error occurs in a packet. This packet loss is done according to a give packet loss probability. For simplicity, we shall set a fixed number of bit errors in each codeword (for the first case), or set a fixed percentage of packet lost rate (for the 2nd case) in this laboratory work.

Step 5.1: (Case 1: bit errors in the channel)

Make Matlab codes that is able to add t errors in each codeword, and insert this part of the codes in between Block-4 (RS encoder) and Block-9 (RS decoder). This will be used for testing the capability

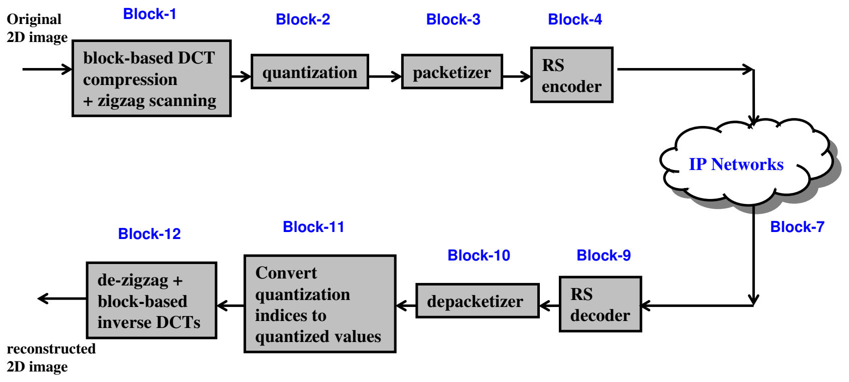

Figure 9: Task 5

of RS codec in correcting bit errors from the channel. According to the RS coding theory, a (n, k) RS code is able to correct $\lfloor (n-k)/2 \rfloor$ errors in each codeword. Set $t < \lfloor (n-k)/2 \rfloor$ in your Matlab program and check whether the codeword errors are corrected from the outputs of the RS decoder. If not, then you need to check bugs in your Matlab codes.

Hint: see Matlab communication toolbox Error-control coding example on detecting and correcting errors in a Reed-Solomon code, for example:

$codes = rsenc(msgw, n, k);$ % Encode the data
$noise = (1 + randint(nw, n, 2^m - 1)). * randerr(nw, n, t);$ % 't' errors per row, for 'nw' codewords
$cnoisy = codes + noise;$ % add noise to the code
$[dc, nerrs, corrcode] = rsdec(cnoisy, n, k);$ % decode the noisy code

Step 5.2: (Case 2: network packet loss)
Make a Matlab code that is able to erase a given number of packets entirely (in the network layer). Assuming a total number of packets is 'nw', remove 10% of packets (e.g. randomly generate the indices for 10% of packets, and then replace the codewords of these packets by zeros). Insert this part of the codes in between Block-4 (RS encoder) and Block-9 (RS decoder) - using "Switch" in Matlab to switch between Case 1 (in Step 5.1) or Case 2 (in Step 5.2) types of noise.

Hint: An example that generates one packet loss:

$e\_packet = zeros(1, n);$ % generate a codeword with zero value, n is codeword size
$errorpacket = gf(e\_packet, m);$ % generate an error packet in Matlab class *gf*
$code\_noisy(i, :) = errorpacket;$ % replace the ith codeword by a packet with zero values

Note, the indices of lost packets should be generated randomly.

Task 7: Adding matrix interleaver and deinterleaver

This task builds up Matlab codes to perform the white block (Block-5 and 8), and then inserts this block into the system shown in Figure 10, completing the building up of the entire system. The Matlab codes for implementing these two added block will then be verified.

Step 7.1: Interleaving
Given an 2D array as the input where each row is a packet (a RS codeword in this occasion), the output packets from a "vector interleaver" are then the columns of this array. To implement this,

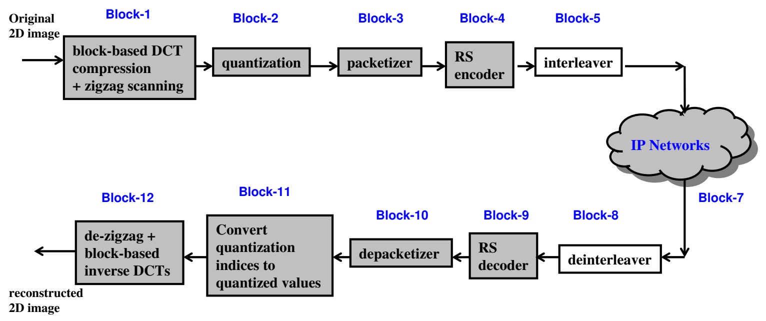

Figure 10: Task 7

you can either use the Matlab matrix interleaving function *matintrlv* (type Matlab help for more details), or make your own Matlab codes to implement this (Assuming the codeword length is n, accumulate n codewords in a matrix and then transpose this matrix. Each row is then a packet to be sent via the channel. The process repeats for every n codewords.
(Important hint: noting that the output from the RS encoder is a *gf* array. The codewords from 'code' can be extracted by using code.x.)

Step 7.2: De-interleaving
This is the exact inverse process of interleaving. Given an array of n packets in columns, deinterleaving output is the rows (received RS codewords in this occasion). Similarly, you can use Matlab function *matdeintrlv* or make your own Matlab codes to implement this. It is worth noting that the input codewords to the RS decoder should be a *gf* array.

Step 7.3: Verification
Plug in the Matlab codes from Block-5 (in Step 7.1) and Block-8 (in Step 7.2) in Figure 10. However, eliminate Block-7 (i.e. assuming no channel noise) and check whether the original input image and the reconstructed image are equal (up to a scale of quantization errors). If so, then the Matlab codes for implementing these 2 blocks are done correctly.

Task 8: Noise impact on transmitting compressed images over Internet

After built up the entire system, we are now ready for testing the system!
Step 8.1: Test the following cases:

(a) Impact of "symbol errors" to the reconstructed images at the receiver side:

- **(a.1)** Set channel noise as "symbol errors" (i.e., Case-1 in Step 5.1). Set $t = 60$ symbol errors for each codeword, remove the interleaver (Block-5) and deinterleaver (Block-8) from the system in Figure 10, test whether all errors are corrected. Plot the original image and the reconstructed image;
- **(a.2)** Repeat the test as in (a.1) however this time the interleaver and deinterleaver blocks are inserted. Compared the result obtained from (a.1) and (a.2): does interleaver help to improve the error correction? Why ?
- **(a.3)** Repeat (a.1), except changing $t = 100$, and test whether all errors are corrected. Plot the original image and the reconstructed image;

(b) Impact of "packet loss" to the reconstructed images at the receiver side:

- **(b.1)** Set noise as "packet losses" in the network (i.e., Case-2 in Step 5.2). Set the packet loss rate as 3% and test the system in Figure 10 without interleaver (Block-5) and deinterleaver (Block-8). Plot the reconstructed image, and explain the results. Does RS codec manage to correct the lost packets?

(b.2) Repeat the same process as in (b.1) with 3% packet loss, however, this time interleaving and deinterleaving blocks are inserted into the system. Plot the reconstructed image, and explain the results. Comparing with no interleaving case, does interleaving offers any improvement in performance?

(c) Compute the reconstructed image quality in the receiver side (the PSNR and the MSSIM):
Note: In all cases, use the non-compressed original images from the sender side as the the true ideal images when computing PSNR and MSSIM.

(c.1) For symbol errors:
Compute the PSNR and MSSIM values for the images obtained from the receiver side, in the cases in (a.1) and (a.3) (i.e., without interleaving) and (a.2) (i.e. with interleaving). Write down the resulting PSNR and MSSIM values for each case.

- Compare the results between (a.1) and (a.2) in terms of image quality, and comment;
- Compare the results between (a.1) and (a.3) in terms of image quality, and comment.

(c.2) For packet losses:
Compute the PSNR and MSSIM values for the images obtained from the receiver side, in the cases in (b.1) (i.e., without interleaving) and (b.2) (i.e. with interleaving). Write down the resulting PSNR and MSSIM values, compare and comment the image quality from these 2 cases.

Report Writing

Write a report including: your test results, your Matlab programs, your notes on issues requiring attention, your observations and comments from this laboratory exercise. In addition, the following discussions should also be included in your report:

1. Describe the effect of matrix interleaver to the quality of reconstructed images at the receiver side in the presence of ”packet loss” case, and in the presence of ”symbol errors” case. Explain the reasons. (brief, no more than 4 lines of text in total!)

2. Describe that in source compression using DCT, do you use a global threshold for the whole image, or a local threshold for each block? What would be the anticipated difference in these 2 settings ? What would be the complication to your algorithm if you use a global setting ?

3. To limit the lab. work, this lab. only transmit compressed 2D images over error prune Internet.

3.1. If we replace the Internet by a **wired communication network**, which corresponding changes (or, extra blocks) you would require to add into the block diagram of Fig.4?

3.2. If we wish transmit compressed **videos** instead of compressed images over error prune Internet as shown in Fig.4, can you briefly sketch the scenario of your system?

3.3. If we wish to transmit compressed videos together with (synchronized) compressed speech signals, can you briefly sketch the scenario of your new system (using a figure with draft block diagram) ?