Documentation of the JPEG algorithm implementation:

Modules and functions description

Contents:

- A user manual.
- Function documentations.

Section I: A user Manual

This is a Matlab implementation of the DCT-based JPEG image compression algorithm.

The Project files are :

- **13** .m helper function files: to perform the sub-operations within the JPEG pipeline; example: dct, inverse dct, Huffman encoding and decoding, etc..
- **2** .m function files 'compressJPEG' and 'decompressJPEG' that perform the whole JPEG compression and decompression pipelines, respectively.
- A directly runnable script 'Experiment_runs.m ' that runs the compression and decompression pipelines 8 different times with different configurations.
- An input image directory containing two sample images for test; one grayscale test image and another RGB color image.

To use the pre-written test script:

Simply run 'Experiment_runs.m' directly to run all of the 8 test cases, or run individual sections according to your choice of test case.

To perform compression using the 'compressJPEG' function you will need to :

-Read the image into a matrix. Use the test images provided or add your own.

-Pass the image matrix to the ''compress_JPEG' function along with the desirable degree of compression (1 for high, 0 for low), and block size of either 8 or 16. The coded stream of binary strings along with the Huffman dictionary will be returned, as well as the Cb and Cr down-sampled components in case of colored images.

- to decompress, pass the outputs of compressJPEG, along with the degree of compression and block size to the function "decompressJPEG". The retrieved image matrix will be returned. You may want to use "imshow" to viewit.

Function Documentations

1-[h,d, Cb, Cr]=compressJPEG(imagematrix,dgree_of_compression,bocksize)

% this function performs JPEG compression. **Input** arguments are: image
% matrix (that you'd like to compress), degree of compression, 0 for low
% compression &1 for high compression, blocksize is either 8 or 16, it
% **outputs** : h is the final coded array & d is the huffman dictionary used
% in huffman coding. Cb and Cr, are the downsampled chrominance components in case of RGB images, or NaNs in case of gray images.

This function is the outermost layer in the compression process; it calls within it all of the other functions used in the JPEG pipeline. If the image is RGB, it is converted into YCbCr, and the Cb and Cr components are down-sampled by a factor of 2. The image is then divided into blocks (either 8 or 16), then passed to the DCT2D function block by block, each block is divided by the suitable quantization table, passed to ZigZag to make it into a 1d array, arrays are combined, run length encoded, then huffman encoded and returned along with the dictionary. The coded stream output (h), is saved into a binary file in the current directory, and the huffman dictionary is saved into a textfile.

2-decomp=decompressJPEG(coded,dict,Cb,Cr, deg_of_comp,blcksize)

%this function takes as **input** a JPEG coded stream of bit strings, the Huffman %dictionary used to encode it, the degree of compression (0 for low %compression & 1 for high), a block size of either 8 or 16, as well as Cb and Cr which are the down-sampled chrominance components in case of RGB images, or NaNs in case of gray images. The **output** is the reconstructed image matrix

This function is the outermost layer in the decompression process; it calls within it all of the helper functions used in the decompression pipeline. Step 1 is calling the huffman decoder, followed by the run length decoder, then the stream is reformed into 8 by 8 matrices using inv_zigzag, the matrices are multiplied by the Quantization-table, passed to idct_2 to reverse the discrete cosine transform then recombined into the recovered image matrix. In case of colored images, the chrominance components are interpolated and converted back to RGB along with the luminance component.

3-coefficients = DCT2D(pixel_mat)

% **given** a block matrix of pixels as input, this function performs discrete cosine transform DCT, and **returns** the DCT coefficients in a matrix of the same size as the input.

DCT is the first step in the JPEG compression process. The image is divided, standardly, into 8 by 8 blocks, then DCT is performed on each of them.

4- pixelarray = Idct_2(coefficients)

%**given** a block matrix of dct coefficients , performs inverse dct and **returns** a matrix with the recalculated pixel values.

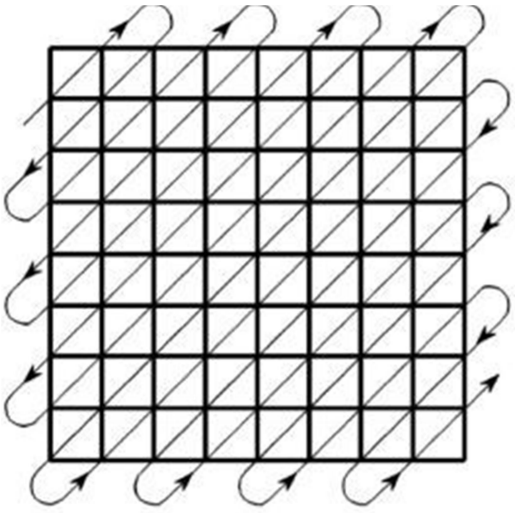
IDCT is the final step in the decompression process. It is performed block-wise, after which the blocks are recombined to the original image matrix.

5-array = ZigZag(matrix)

%**input**: square matrix

%**output**: array retrieved by serpentine pattern

Serpentine pattern is as indicated in this image:



Example: input = [1 2 3; output= [1 2 4 7 5 3 6 8 9]

4 5 6;

7 8 9]

This is step is used in the compression process to transform the quantized blocks into arrays so they could be R.L encoded efficiently.

6- function matrix = inv_zigzag(array)

%**output**: square matrix

%**input**: array retrieved by serpentine pattern

This function is used to reform the decoded arrays back into blocks before dequantization and idct. Undoes the ZigZag function.

7- function [symbols,counts, probabilities] = symbol_freq(vector)

% this function takes as **input** a stream of data representing some combination of % symbols, extracts the unique symbols, its count in the stream (# of times it is repeated) & the probability of each symbol. **Outputs** are a sorted array of symbols, an array of corresponding counts, and another with corresponding probabilities. Indices of a given symbol, its count, and its probability are identical.

Example: input vector = [0,0,1,5,5,7]

return will be symbols =[0,1,5,7], counts=[2,1,2,1] prob=[0.33,0.167,0.33,0.167]

Used in the Huff_dict function to extract the symbols from a long stream along with their probabilities.

8-[coded] = run_length(original)

this functions performs R.L encoding. **Input** is an array of symbols (integers). **Output** is a run length coded array of integer symbols and float value counts of the form count+0.5; the coded array follows the format [sym1, #of repetitions+0.5, sym2,.. so on] when the value is repeated only once, we don't write 1 after it, we only write the number of repetitions for values repeating more than once. the purpose of adding 0.5 to the # of repetition is to distinguish it from the original array symbol values. Since we know that all of the values in the array are going to

be integers for our particular case (because they result from a round operation in the quantization step), if we just write the count after each symbol directly, we will have no way of knowing whether a particular array element is a symbol or thecount of the previous symbol. We use the counts+0.5 convention to distinguish counts since real symbols will always be integers.

Example: input=[7 7 7 2 8 9 9 9 9 7] output= [7, 3.5 , 2 , 8 , 9 , 4.5 , 7]

R.L encoding is the step that follows ZigZag-ing the quantized matrix in the compressor. Its purpose is to compress the long tail of zeros resulting from quantization.

9-decoded = inv_run_length(coded)

%**input**: a run length encoded array of symbols where elements that represent Follow the convention: count+0.5

output is the original, R.L decoded array

Second step in the decoder that follows huffman decoding.

10-dict = huff_dict(s,p)

%**input**: s: array of unique symbols, p: their corresponding probabilities

%with matching indices,

%**output**: a huffman dictionary: an array of structs, each containing 2

%datafields: the symbol, and the codeword

This function generates the huffman codewords for each symbol, given an arrayof unique symbols and their probabilities. It is called within the Huff_encode function.

11-[coded_arr ,dict]= Huff_encode(in_arr)

given an **input** stream of symbols,and a huffman dictionary
the function huffman-encodes the array and returns an array of strings each element is the binary huffman encoded version (represented as a string) of its corresponding symbol . **Returns** the coded array and the huffman dictionary.

Huffman encoding is the final step in compression. It is given as input the run length encode array of quantized DCT coefficients. Its output is passed as the output of the compressJPEG function. This function calls within the symbol_freq function, passes the output symbol and probabilities arrays to the huff_dict function, then uses the resulting huffman dictionary to encode the input array.

12-decoded = Huff_decode(coded_arr,dict)

%**input**: array of bit strings, representing the huffman coded stream of symbols
% and the huff dictionary.
%**output** decode stream of symbols (int array)It is the first step in the decoding process.

13-[y, Cb_downsampled, Cr_downsampled] = RGB_to_YCbCr(rgb_img, blocksize)

% a function to convert the input RGB image to YCbCr, then downsample the
% chrominance components by a factor of 2
% INPUT: rgb_im: the 3 dimensional RGB image
% OUTPUTS: y: the luminance component,
% Cb_resampled: the downsampled blue chrominance component
% Cr_resampled: the downsampled red chrominance component

14-rgb = YCbCr_to_RGB(y,cb,cr)

%this function converts a ycbcr image to rgb given the y (luminance), and
%the downsampled chrominance (cb,cr) components.
%the downsampling factor is 2
% INPUTS: y, cb, cr: the luminance, downsampled red-chrominance,
% and downsampled blue chrominance components respectively.
% OUTPUTS: rgb: the RGB image array