IMAGE COMPRESSION TUTORIAL

Abstract

Various encoding techniques have been implemented and understood to have a clearer understanding in working of the algorithm.

VAISHNAV Mohit and BATERIWALA Malav

Under the Supervision of
STOLZ CHRISTOPHE

UNIVERSITÉ DE BOURGOGNE

Universitat de Girona

HERIOT WATT UNIVERSITY

Table of Contents

Table of Figures:

Arithmetic Coding

Arithmetic coding is a form of entropy encoding used in lossless data compression. Normally, a string of characters such as the words "hello there" is represented using a fixed number of bits per character, as in the ASCII code. When a string is converted to arithmetic encoding, frequently used characters will be stored with fewer bits and not-so-frequently occurring characters will be stored with more bits, resulting in fewer bits used in total. Arithmetic coding differs from other forms of entropy encoding, such as Huffman coding, in that rather than separating the input into component symbols and replacing each with a code, arithmetic coding encodes the entire message into a single number, an arbitrary-precision fraction q where $0.0 \leq q < 1.0$. It represents the current information as a range, defined by two numbers. Recent Asymmetric Numeral Systems family of entropy coders allows for faster implementations thanks to directly operating on a single natural number representing the current information.

Encoding and decoding: overview

In general, each step of the encoding process, except for the very last, is the same; the encoder has basically just three pieces of data to consider:

- The next symbol that needs to be encoded
- The current interval (at the very start of the encoding process, the interval is set to [0,1], but that will change)
- The probabilities the model assigns to each of the various symbols that are possible at this stage (as mentioned earlier, higher-order or adaptive models mean that these probabilities are not necessarily the same in each step.)

The encoder divides the current interval into sub-intervals, each representing a fraction of the current interval proportional to the probability of that symbol in the current context. Whichever interval corresponds to the actual symbol that is next to be encoded becomes the interval used in the next step.

Example: for the four-symbol model above:

- the interval for NEUTRAL would be [0, 0.6)
- the interval for POSITIVE would be [0.6, 0.8)
- the interval for NEGATIVE would be [0.8, 0.9)
- the interval for END-OF-DATA would be [0.9, 1).

When all symbols have been encoded, the resulting interval unambiguously identifies the sequence of symbols that produced it. Anyone who has the same final interval and model that is being used can reconstruct the symbol sequence that must have entered the encoder to result in that final interval.

It is not necessary to transmit the final interval, however; it is only necessary to transmit *one fraction* that lies within that interval. In particular, it is only necessary to transmit enough digits (in whatever base) of the fraction so that all fractions that begin with those digits fall into the final interval; this will guarantee that the resulting code is a prefix code.

MATLAB CODE

```
%Image Compression Tutorial
```

```
%This image compression assignment is provided as an assessment tool for
%the Course on Image processing. Supervisor concerned for this is STOLZ
%Christopher
```

```
%We have to implemen three type of encoding and decoding techniques:
%Arithmatic, Huffman and JPEG. Then try this implementation on various
%input streams.
```

```
%Arithmatic Encoder:

%Input arguments for the following function are the list to different
%sumbols for

clc
clear
```

```
%Taking the input from the user: Remove the comment for taking input
%prompt='  Enter the word    ';
%s=input(prompt,'s');
```

```
s = 'BE_A_BEE';
%Finding the length of string
len=length(s);

%Finding out the unique elements from each:
s_u = unique(s);
disp(['Unique elements in the string are ',s_u]);
len_u = length(s_u);
p = zeros(len_u,1);

%Finding the probability of each element:
for i=1:len_u
       p(i)=length(strfind(s,s_u(i)));
end
p = p/(sum(p));

[b, d] = arithmetic_encoding(s_u, p, s);
disp(['Binary coded value of the string is ',b]);
disp(['d coded value of the string is ',num2str(d)]);

fprintf('Decoding the sequence:')
d_d = arithmetic_d(s_u, p, d, len);
disp(['Value of the string is ',d_d]);
```

```
Unique elements in the string are ABE_
```

```
%Explicitely passing the values, probability and the string to be encoded:

[b, d] = arithmetic_encoding(['A','C','T','G'], [0.5,0.3,0.15,0.05], ['A','C','T','A','G','C']);
disp(['Binary coded value of the string is ',b]);
disp(['d coded value of the string is ',num2str(d)]);

fprintf('Decoding the sequence:')
d_d = arithmetic_d(['A','C','T','G'], [0.5,0.3,0.15,0.05], d, 6);
disp(['Value of the string is ',d_d]);
```

```
%Defining the functions:
function [binary, deci] = arithmetic_encoding(s_u, p, s)

%After the counting of the frequencies of each symbols and sort them in the
%decreasing order, we calculate the corresponding intervals of probability
%with a High and a Low bound. The Range of the intervals corresponds to the probability of the
symbol.
%Then, the sequence is iteratively coded with the following formulas:

    HighRange(s_u(1)) = 1.;
    LowRange(s_u(1)) = 1 - p(1);
    for i = 2 : length(s_u)
        HighRange(s_u(i)) = LowRange(s_u(i-1));
        LowRange(s_u(i)) = HighRange(s_u(i)) - p(i);
    end

    %Initialize the OldLow to 0 and OldHigh to 1
    OldLow = 0.;
    OldHigh = 1.;

    %Fill the binary_stream by calculating with the given formulas
    for i = 1 : length(s)
        Range = OldHigh - OldLow;
        newhigh = OldLow + Range * HighRange(s(i));
        newlow = OldLow + Range * LowRange(s(i));
        OldHigh = newhigh;
        OldLow = newlow;
    end

    %converting the number to the binary form from fraciton
    q = quantizer('single');
       binary = num2bin(q,newlow);
    deci = newlow;
end
```

```
Binary coded value of the string is 00111111001101010110001000100000
d coded value of the string is 0.70853
Decoding the sequence:

Binary coded value of the string is 00111111000111100110110111000110
d coded value of the string is 0.61886
Decoding the sequence:
```

Arithmatic Decoding:

```
function decoded_sequence = arithmetic_d(s_u, p, d, size)

    %Create a list of all high ranges and a list of all low ranges for
    %symbols.

    HighRange = zeros(length(s_u),1);
    LowRange = zeros(length(s_u),1);
    HighRange(1) = 1.0;
    LowRange(1) = vpa(1. - p(1),4);
    for i = 2 : length(s_u)
        HighRange(i) = LowRange(i-1);
        LowRange(i) = vpa(HighRange(i) - p(i),4);
    end

    %Fill the decoded_sequence by calculating with the given formulas. A
    %test condition is made to check if the decoded_sequence is at value 0,
    %and thus, 0 should not be included in the DNA alphabet.
    decoded_sequence = strings(1,size);
    for i = 1 : size
        for j = 1 : length(s_u)
            if (d < HighRange(j)) && (d >= LowRange(j))
                decoded_sequence(i) = s_u(j)
                d = vpa((d - LowRange(j)) / (HighRange(j)-LowRange(j)),4);
                break
            end
        end
    end
end
```

```
decoded_sequence =

  1×8 string array

    "B"    ""    ""    ""    ""    ""    ""    ""


decoded_sequence =

  1×8 string array

    "B"    "E"    ""    ""    ""    ""    ""    ""


decoded_sequence =

  1×8 string array

    "B"    "E"    "_"    ""    ""    ""    ""    ""
```

```
decoded_sequence =

  1×8 string array

    "B"    "E"    "_"    "A"    ""    ""    ""    ""


decoded_sequence =

  1×8 string array

    "B"    "E"    "_"    "A"    "_"    ""    ""    ""


decoded_sequence =

  1×8 string array

    "B"    "E"    "_"    "A"    "_"    "B"    ""    ""


decoded_sequence =

  1×8 string array

    "B"    "E"    "_"    "A"    "_"    "B"    "E"    ""


decoded_sequence =

  1×8 string array

    "B"    "E"    "_"    "A"    "_"    "B"    "E"    "E"

  Columns 1 through 8

    "Value of the stri…"    "B"    "E"    "_"    "A"    "_"    "B"    "E"

  Column 9

    "E"
```

Published with MATLAB® R2017a

Introduction to Huffman Coding

In computer science and information theory, a **Huffman code** is a optimal prefix code that is commonly used for lossless data compression. The process of finding and/or using such a code proceeds by means of **Huffman coding**, an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".

The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (*weight*) for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted. However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods.

Basic Procedure

Compression

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols, N . A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the **symbol** itself, the **weight** (frequency of appearance) of the symbol and optionally, a link to a **parent** node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain a **weight**, links to **two child nodes** and an optional link to a **parent** node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and n-1 internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The process begins with the leaf nodes containing the probabilities of the symbol they represent. Then, the process takes the two nodes with smallest probability, and creates a new internal node having these two nodes as children. The weight of the new node is set to the sum of the weight of the children. We then apply the process again, on the new internal node and on the remaining nodes (i.e., we exclude the two leaf nodes), we repeat this process until only one node remains, which is the root of the Huffman tree.

The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority:

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
    1. Remove the two nodes of highest priority (lowest probability) from the queue
    2. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
    3. Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

Since efficient priority queue data structures require O(log n) time per insertion, and a tree with n leaves has $2n-1$ nodes, this algorithm operates in O(n log n) time, where n is the number of symbols.

If the symbols are sorted by probability, there is a linear-time (O(*n*)) method to create a Huffman tree using two queues, the first one containing the initial weights (along with pointers to the associated leaves), and combined weights (along with pointers to the trees) being put in the back of the second queue. This assures that the lowest weight is always kept at the front of one of the two queues:

1. Start with as many leaves as there are symbols.
2. Enqueue all leaf nodes into the first queue (by probability in increasing order so that the least likely item is in the head of the queue).
3. While there is more than one node in the queues:
    1. Dequeue the two nodes with the lowest weight by examining the fronts of both queues.
    2. Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the new weight.
    3. Enqueue the new node into the rear of the second queue.
4. The remaining node is the root node; the tree has now been generated.

In many cases, time complexity is not very important in the choice of algorithm here, since *n* here is the number of symbols in the alphabet, which is typically a very small number (compared to the length of the message to be encoded); whereas complexity analysis concerns the behavior when *n* grows to be very large.

It is generally beneficial to minimize the variance of codeword length. For example, a communication buffer receiving Huffman-encoded data may need to be larger to deal with especially long symbols if the tree is especially unbalanced. To minimize variance, simply break ties between queues by choosing the item in the first queue. This modification will retain the mathematical optimality of the Huffman coding while both minimizing variance and minimizing the length of the longest character code.

Decompression

Generally speaking, the process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that particular byte value). Before this can take place, however, the Huffman tree must be somehow reconstructed. In the simplest case, where character frequencies are fairly predictable, the tree can be preconstructed (and even statistically adjusted on each compression cycle) and thus reused every time, at the expense of at least some measure of compression efficiency. Otherwise, the information to reconstruct the tree must be sent a priori. A naive approach might be to prepend the frequency count of each character to the compression stream. Unfortunately, the overhead in such a case could amount to several kilobytes, so this method has little practical use. If the data is compressed using canonical encoding, the compression model can be precisely reconstructed with just n bits of information (where n-1 is the number of bits per symbol). Another method is to simply prepend the Huffman tree, bit by bit, to the output stream. For example, assuming that the value of 0 represents a parent node and 1 a leaf node, whenever the latter is encountered the tree building routine simply reads the next 8 bits to determine the character value of that particular leaf. The process continues recursively until the last leaf node is reached; at that point, the Huffman tree will thus be faithfully reconstructed. The overhead using such a method ranges from roughly 2 to 320 bytes (assuming an 8-bit alphabet). Many other techniques are possible as well. In any case, since the compressed data can include unused "trailing bits" the decompressor must be able to determine when to stop producing output. This can be accomplished by either transmitting the length of the

decompressed data along with the compression model or by defining a special code symbol to signify the end of input (the latter method can adversely affect code length optimality, however).

Example :

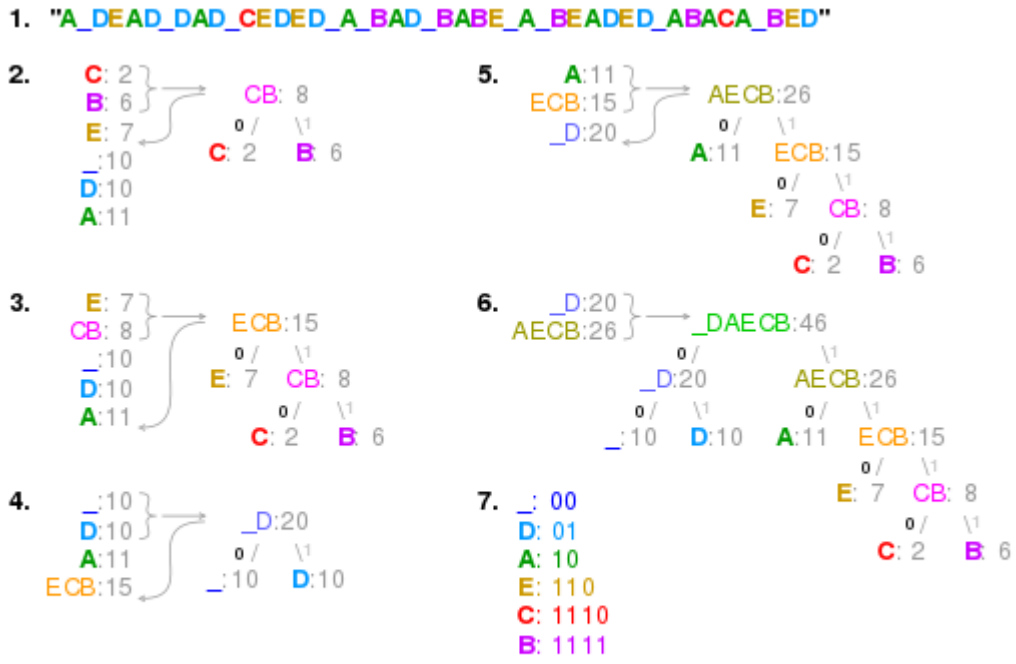

Figure 1: Example of Huffman

MATLAB CODE

```
clc
clear

alphabet = {'0' '1' '2' '3' '4' '5' '6' '7'};
p = [0.05 0.2 0 0.2 0.1 0.25 0.05 0.15];
[tree, tab] = hufftree(alphabet,p);
function [tree, table] = hufftree(alphabet,prob)

   for l=1:length(alphabet)       % create a vector of nodes (leaves), one for each letter
      leaves(l).val = alphabet{l};
      leaves(l).zero= '';
      leaves(l).one='';
      leaves(l).prob = prob(l);
   end

   % combine the two nodes with lowest probability to a new node with the summed prob.
   % repeat until only one node is left
   while length(leaves)>1
      [dummy,I]=sort(prob);
      prob = [prob(I(1))+prob(I(2)) prob(I(3:end))];
      node.zero = leaves(I(1));
      node.one  = leaves(I(2));
      node.prob = prob(1);
      node.val = '';
      leaves = [node leaves(I(3:end))];
   end


   % pass through the tree,
   % remove unnecessary information
   % and create table recursively (depth first)
   table.val={}; table.code={};
   [tree, table] = descend(leaves(1),table,'');
end

   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [tree, table] = descend(oldtree, oldtable, code)

    table = oldtable;
    if(~isempty(oldtree.val))
      tree.val = oldtree.val;
      table.val{end+1}  = oldtree.val;
      table.code{end+1} = code;
    else
      [tree0, table] = descend(oldtree.zero, table, strcat(code,'0'));
      [tree1, table] = descend(oldtree.one,  table, strcat(code,'1'));
      tree.zero=tree0;
      tree.one= tree1;
    end
```

```
end

% huffencode.m
%
% takes a cell-vector and a huffman-table
% returns a huffman encoded bit-string

function bitstring = huffencode(input, table)
    bitstring = '';
    for l=1:length(input)
      bitstring = strcat(bitstring,table.code{strcmp(table.val,input{l})}); % omits letters that
are not in alphabet
    end
end
```

Introduction to 1-D Lossless

Lossless compression is a class of data compression algorithms that allows the original data to be perfectly reconstructed from the compressed data. By contrast, lossy compression permits reconstruction only of an approximation of the original data, though this usually improves compression rates (and therefore reduces file sizes).

Lossless data compression is used in many applications. For example, it is used in the ZIP file format and in the GNU tool gzip. It is also often used as a component within lossy data compression technologies (e.g. lossless mid/side joint stereo preprocessing by MP3 encoders and other lossy audio encoders).

Lossless compression is used in cases where it is important that the original and the decompressed data be identical, or where deviations from the original data would be unfavourable. Typical examples are executable programs, text documents, and source code. Some image file formats, like PNG or GIF, use only lossless compression, while others like TIFF and MNG may use either lossless or lossy methods. Lossless audio formats are most often used for archiving or production purposes, while smaller lossy audio files are typically used on portable players and in other cases where storage space is limited or exact replication of the audio is unnecessary.

Most lossless compression programs do two things in sequence: the first step generates a *statistical model* for the input data, and the second step uses this model to map input data to bit sequences in such a way that "probable" (e.g. frequently encountered) data will produce shorter output than "improbable" data.

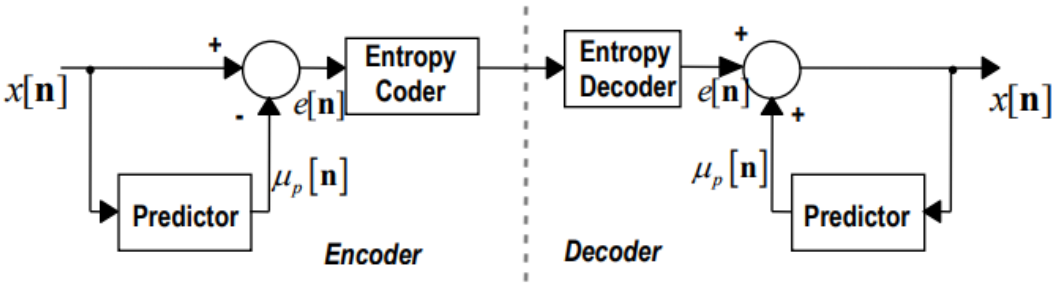

Figure 2: Huffman Method

The primary encoding algorithms used to produce bit sequences are Huffman coding (also used by DEFLATE) and arithmetic coding. Arithmetic coding achieves compression rates close to the best possible for a particular statistical model, which is given by the information entropy, whereas Huffman compression is simpler and faster but produces poor results for models that deal with symbol probabilities close to 1.

There are two primary ways of constructing statistical models: in a *static* model, the data is analyzed and a model is constructed, then this model is stored with the compressed data. This approach is simple and modular, but has the disadvantage that the model itself can be expensive to store, and also that it forces using a single model for all data being compressed, and so performs poorly on files that contain heterogeneous data. *Adaptive* models dynamically update the model as the data is compressed. Both the encoder and decoder begin with a trivial model, yielding poor compression of initial data, but as they learn more about the data, performance improves. Most popular types of compression used in practice now use adaptive coders.

Lossless compression methods may be categorized according to the type of data they are designed to compress. While, in principle, any general-purpose lossless compression algorithm (*general-purpose* meaning that they can accept any bitstring) can be used on any type of data, many are unable to achieve significant compression on data that are not of the form for which they were designed to compress. Many of the lossless compression techniques used for text also work reasonably well for indexed images.

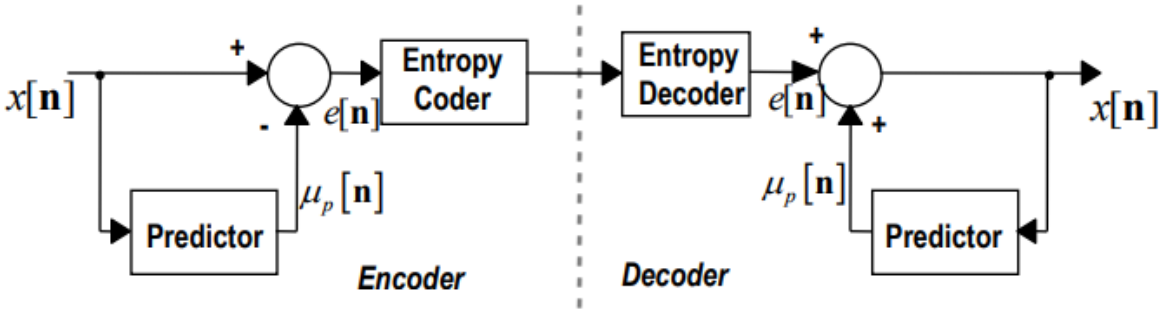

Figure 3: Huffman Decoding Method

MATLAB CODE

```
clc
clear
close all
```

```
%Implementation of the 1D lossless image compression:
```

```
%Read the image:
I = imread('lena-grey.bmp');
figure, imshow(I, [])
title('Original Image')
```

Figure 4: Lena Original Image

```
%To calculate the size
[l, b] = size(I);
```

```
%initializing the array:
e = zeros(size(I));
```

```
%Implementing 1D Predictive coding:
for i = 1:l
    for j = 2:b
        e(i,j) = I(i,j) - I(i,j-1);
    end
end
```

```
%showing encoded image:
figure, imshow(e, []);
title('Encoded Image')
```

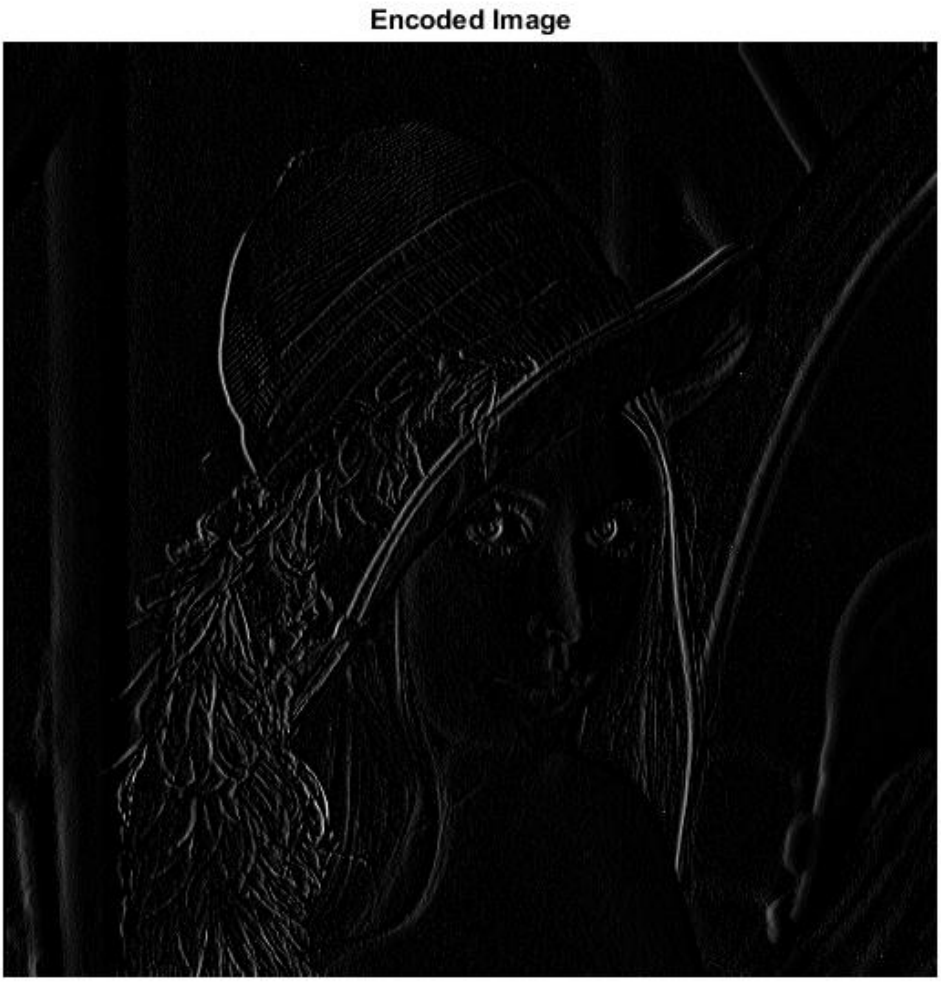

Figure 5: Lena Encoded Image

```
%entropy of original image:
e_o = entropy(I);
disp(['Entropy Value of the Original Image is ',num2str(e_o)]);
```

```
Entropy Value of the Original Image is 7.4455
```

```
%entropy of encoded image:
e_e = entropy(e);
disp(['Entropy Value of the encoded Image is ',num2str(e_e)]);
```

```
Entropy Value of the encoded Image is 0.99817
```

```
%histogram of Original image:
figure, histogram(I)
title('Histogram of Original image')

%histogram of encoded image:
figure, histogram(e)
title('Histogram of encoded image')
```

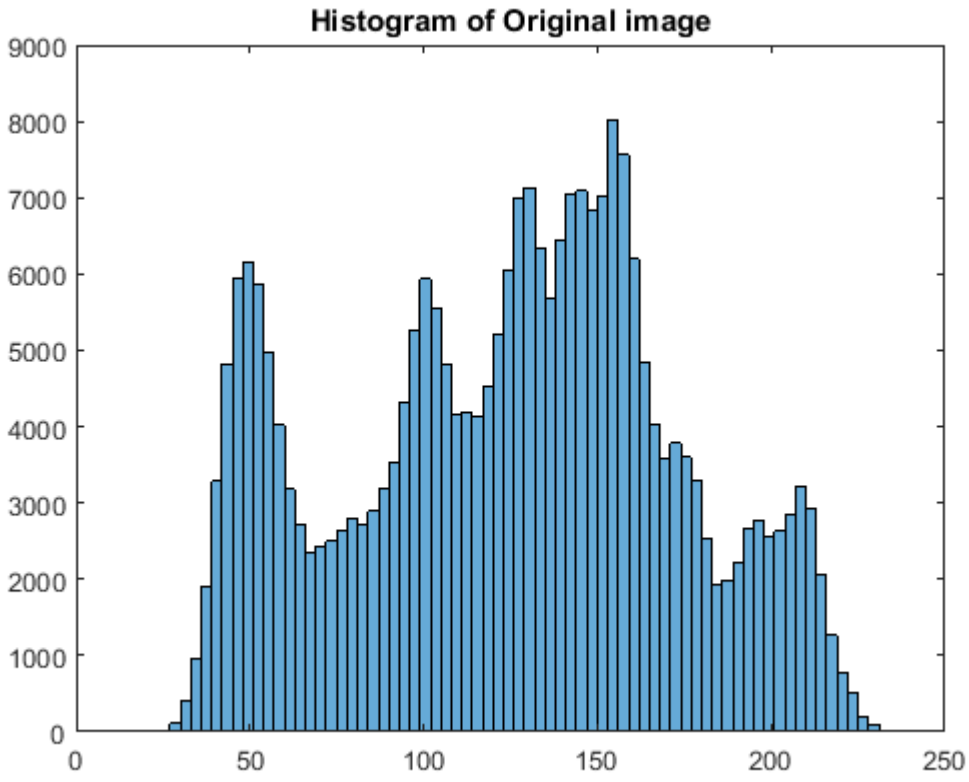

Figure 6: Histogram of Original Image

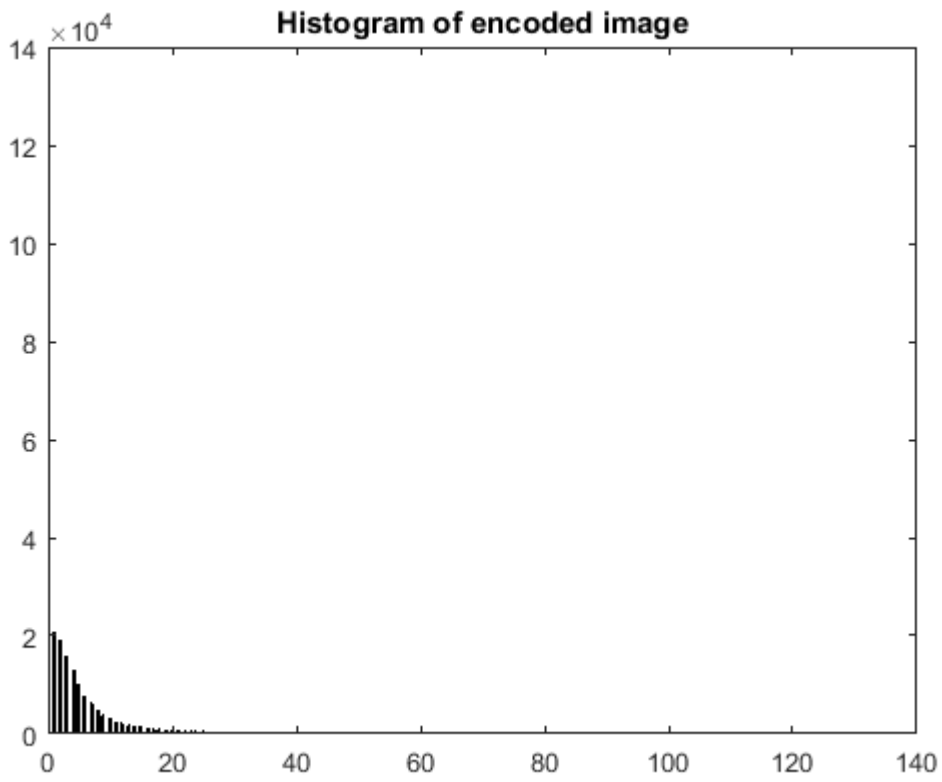

Figure 7: Histogram of encoded image

Published with MATLAB® R2017a

JPEG

What is JPEG?

JPEG stands for Joint Photographic Experts Group, which was a group of image processing experts that devised a standard for compressing images (ISO).

So, JPEG (or JPG) is not really a file format but rather an image compression standard. The JPEG standard is complicated with many different options and color space regulations. It was not widely adopted. A much simpler standard version was advocated at the same time, called JFIF. This is the image compression algorithm that most people mean when they say JPEG compression, and the one that we will be describing in this class. Note that the file extensions .jpeg and .jpg have stuck, even though the underneath algorithm is (strictly speaking) JFIF compression.

The underlying assumptions of the JPEG algorithm

The JPEG algorithm is designed specifically for the human eye. It exploits the following biological properties of human sight:

(1) We are more sensitive to the illuminocity of color, rather than the chromatric value of an image, and

(2) We are not particularly sensitive to high-frequency content in images.

The algorithm can be neatly broken up into several stages: There is an input image I, which goes through the following process:

1) A colour transform, 2) A 2D discrete cosine transform on 8x8 blocks, 3) A quantization (filtering) stage, 4) Huffman encoding.

Finally, a compressed image is returned in the .jpg file format. This format contains the compressed image as well as information that is needed to uncompressed, with other information to allow for reexpanding the image.

An uncompressed image of mixed peppers

MATLAB has various images uploaded into MATLAB. Here, is one of some peppers

```
I = imread('peppers.png');

imshow(I)
```

Figure 8: Peppers image

It currently requires about the following number of bits to store:

```
ImageSize = 8*prod(size(I))
```

```
ImageSize =

     4718592
```

Color transform: Convert RGB to YCbCr

Right now, the image is stored in RGB format. While this colorspace is convenient for projecting the image on the computer screen, it does not isolate the illuminance and color of an image. The intensity of color is intermixed in the colorspace. The YCbCr is a more convenitent colorspace for image compression because it separates the illuminance and the chromatic strength of an image.

```
Y = I;

A = [.229 .587 .114 ; -.168736 -.331264 .5 ; .5 -.418688 -.081312];

Y(:,:,1) = A(1,1)*I(:,:,1) + A(1,2)*I(:,:,2) + A(1,3)*I(:,:,3);

Y(:,:,2) = A(2,1)*I(:,:,1) + A(2,2)*I(:,:,2) + A(2,3)*I(:,:,3) + 128;

Y(:,:,3) = A(3,1)*I(:,:,1) + A(3,2)*I(:,:,2) + A(3,3)*I(:,:,3) + 128;



% Let's use MATLAB's inbuilt convert (because of the normalizations):
Y = rgb2ycbcr(I);
```

Plot Y'CbCr colorspace

Let's see what that colorspace looks like.

```
lb = {'Y', 'Cb', 'Cr'};


for channel = 1:3
    subplot(1,3,channel)
    Y_C = Y;
    Y_C(:,:,setdiff(1:3,channel)) = intmax(class(Y_C))/2;
    imshow(ycbcr2rgb(Y_C))
    title([lb{channel} ' component'],'fontsize',16)
end
```

Figure 9: Y, Cb, Cr Componenets

Our eyes are senstitive to illuminance, but not color

Since our eyes are not particularly sensitive to chrominance, we can "downsample" that stuff. Here, we remove x100 amount of "color" from the image and see that it has barely changed:

```
subplot(1,2,1)

imshow( I )

title('Original')

subplot(1,2,2)

Y_d = Y;

Y_d(:,:,2) = 10*round(Y_d(:,:,2)/10);

Y_d(:,:,3) = 10*round(Y_d(:,:,3)/10);

imshow(ycbcr2rgb(Y_d))

title('Downsampled image')
```

Figure 10: Downsampling of image for JPEG

Eyes are senstitive to intensity

However, if we downsample the illuminance by x10, then there is a noticeable difference. (You'll have to zoom in to see it.)

```
subplot(1,2,1)

imshow( I )

title('original')
```

```
subplot(1,2,2)
Y_d = Y;
Y_d(:,:,1) = 10*round(Y_d(:,:,1)/10);
imshow(ycbcr2rgb(Y_d))
title('Downsample illuminance')
```

Figure 11: Downsampling Illuminance

JPEG downsampling

Here, we will be a little conservative and downsample the chrominance by only a factor of 2.

```
Y_d = Y;
Y_d(:,:,2) = 2*round(Y_d(:,:,2)/2);
Y_d(:,:,3) = 2*round(Y_d(:,:,3)/2);
```

The 2D discrete cosine transform

Once the image is in YCrCb color space and downsampled, it is partitioned into 8x8 blocks. Each block is transformed by the two-dimensional discrete cosine transform (DCT). Let's extract one 8x8 block of pixels for demonstration, shown here in white:

We apply the DCT to that box:

```
clf
box = Y_d;
II = box(200:207,200:207,1);
```

```
Y = chebfun.dct(chebfun.dct(II).').';

surf(log10(abs(Y))), title('DCT coefficients')

set(gca,'fontsize',16)
```

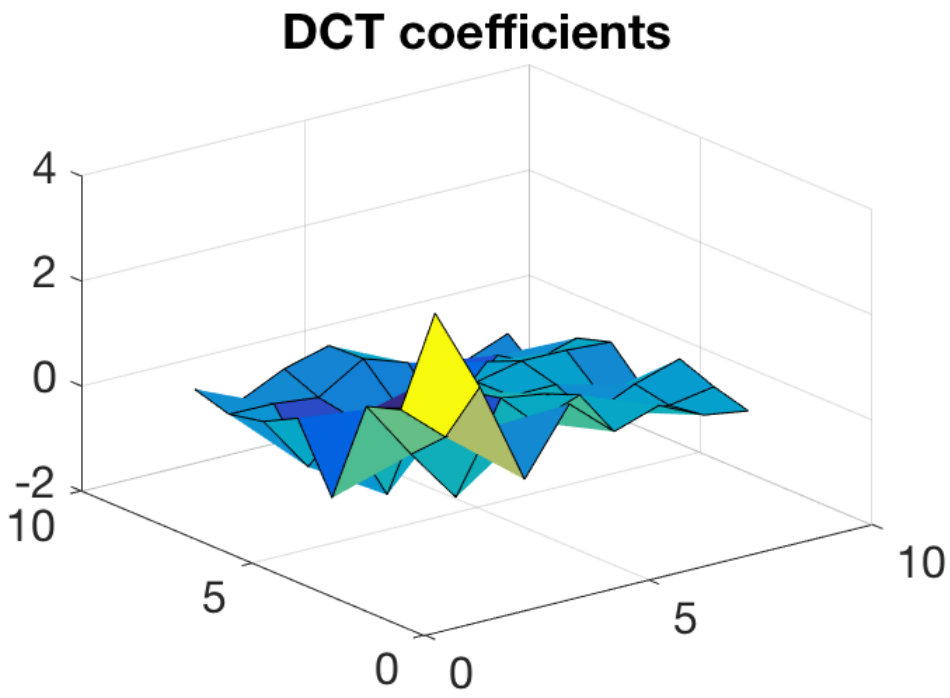

Figure 12: DCT Coefficients obtained

Quantization table

Next we apply a quantization table to Y, which filters out the high frequency DCT coefficients:

```
Q = [16 11 10 16 24 40 51 61 ;

     12 12 14 19 26 28 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];
```

```
before = nnz(Y);

Y = Q.*round(Y./Q);

after = nnz(Y);
```

The number of nonzero DCT coefficients after quantization is:

```
before, after
```

```
before =

    64

after =

     6
```

We now apply this compression to every 8x8 block. We obtain:

```
I = imread('peppers.png');
Y_d = rgb2ycbcr( I );
% Downsample:
Y_d(:,:,2) = 2*round(Y_d(:,:,2)/2);
Y_d(:,:,3) = 2*round(Y_d(:,:,3)/2);
% DCT compress:
A = zeros(size(Y_d));
B = A;
for channel = 1:3
    for j = 1:8:size(Y_d,1)-7
        for k = 1:8:size(Y_d,2)
            II = Y_d(j:j+7,k:k+7,channel);
            freq = chebfun.dct(chebfun.dct(II).').';
            freq = Q.*round(freq./Q);
            A(j:j+7,k:k+7,channel) = freq;
            B(j:j+7,k:k+7,channel) = chebfun.idct(chebfun.idct(freq).').';
        end
    end
end
```

```
subplot(1,2,1)
imshow(ycbcr2rgb(Y_d))
title('Original')
subplot(1,2,2)
imshow(ycbcr2rgb(uint8(B)));
title('Compressed')
shg
```

Figure 13: Final Compressed Image

Compression so far

The quantization step has successfully compressed the image by about a factor of 7.

```
CompressedImageSize = 8*nnz(A(:,:,1)) + 7*nnz(A(:,:,2)) + 7*nnz(A(:,:,3))

CompressedImageSize/ImageSize
```

```
CompressedImageSize =

      701189

ans =

   0.148601320054796
```

The formula above is obtained by noting that we downsampled in Cr and Cb are downsampled.

Huffman encoding

We now encode the DCT blocks using Huffman encoding:

```
b = A(:);

b = b(:);

b(b==0)=[];  %remove zeros.
b = floor(255*(b-min(b))/(max(b)-min(b)));
symbols = unique(b);
prob = histcounts(b,length(symbols))/length(b);
dict = huffmandict(symbols, prob);
enco = huffmanenco(b, dict);
FinalCompressedImage = length(enco)
```

```
FinalCompressedImage =

      695755
```

Final compression

We have successfully reduced the pepper image by about x7, while being extremely conservative:

```
length(enco)/ImageSize   % x7 compression.
```

```
ans =

   0.147449705335829
```

A different image

The images of peppers is not ideal for JPEG. Here is an image for which JPEG gets a better compression rate (of about x30).

```
I = imread('saturn.png');
ImageSize = 8*prod(size(I));
Y_d = rgb2ycbcr( I );
% Downsample:
Y_d(:,:,2) = 2*round(Y_d(:,:,2)/2);
Y_d(:,:,3) = 2*round(Y_d(:,:,3)/2);
% DCT compress:
A = zeros(size(Y_d));
```

```
B = A;
for channel = 1:3
    for j = 1:8:size(Y_d,1)-7
        for k = 1:8:size(Y_d,2)-7
            II = Y_d(j:j+7,k:k+7,channel);
            freq = chebfun.dct(chebfun.dct(II).').';
            freq = Q.*round(freq./Q);
            A(j:j+7,k:k+7,channel) = freq;
            % do the inverse at the same time:
            B(j:j+7,k:k+7,channel) = chebfun.idct(chebfun.idct(freq).').';
        end
    end
end
b = A(:);
b = b(:);
b(b==0)=[];  %remove zeros.
b = floor(255*(b-min(b))/(max(b)-min(b)));
symbols = unique(b);
prob = histcounts(b,length(symbols))/length(b);
dict = huffmandict(symbols, prob);
enco = huffmanenco(b, dict);
FinalCompressedImage = length(enco);

FinalCompressedImage/ImageSize
```

```
ans =

   0.031958287037037
```

Here's what the images look like:

```
subplot(1,2,1)

imshow(I)

title('Original')
subplot(1,2,2)
```

```
imshow(ycbcr2rgb(uint8(B)));
title('Compressed')
```

Conclusion:

We have seen a few compression algorithms and their working. Their MATLAB implementation has been presented below every code. Using these techniques gave us the idea about how the compression is useful in our daily life. Let that be an image, video sequence or data without any encoding method their size would have been huge and to save those in our todays portable computers is almost close to impossible. With the help of these coding methods we can store vast amount of data in our daily life to either smartphones or any other electronic gadgets. The compression ratio achieved at the cost of losing negligible data is humongous.