ECE 421 – Introduction to Signal Processing

Project 4: Lossy Image Compression

Hangjin Liu and Dror Baron; Spring 2021

Due: March 29, 2021

1 Administrative Instructions

1. The project should be submitted in pairs or triples.
2. For any clarification or doubts, please contact the TA (email: Hangjin Liu (email: hliu25 AT ncsu DOT edu).
3. You should submit an electronic copy via Moodle by midnight the day that the project is due.
4. Your report should describe any mathematical derivations, responses to questions, results including any plots, and your MATLAB code. Please justify your answers carefully.

2 Motivation

Images and video signals require a great deal of data to transmit, thus dominating bandwidth requirements in conventional (wired) and wireless communication systems. For example, 80% of web traffic in 2019 will be comprised of video,* and video data traffic is growing rapidly. Moreover, *wireless bandwidth* in mobile phone systems is a limited natural resource, meaning that when mobile phone users transmit, they interfere with transmissions of other users. Keeping in mind that uncompressed video data requires many giga bytes (GB) per minute, and even compressed video requires several GB per hour, *storage space* is also a major limitation. These communication bandwidth and storage space challenges motivate us to represent video signals efficiently.

This project focuses on *lossy data compression* of images. In contrast to lossless data compression, which represents data perfectly without any loss or distortion, lossy compression deals with situations where a small amount of distortion between the input and output is allowed. That is, the end user will likely be comfortable with a minor amount of *distortion* between the original image or video signal and the one being displayed. At the same time, allowing for some distortion provides an opportunity to drastically reduce the data rates required to store or communicate the data.

Even short video signals are comprised of hundreds of image frames, and so video signals require much larger amounts of data than image signals. Nonetheless, this project focuses on *image compression*, because it captures the essence of video compression while being more approachable in a course project. As part of this project, you will design and create code for a simple lossy image compression system loosely based on the *JPEG standard* for lossy image compression. While your implementation will be rudimentary, for example it will only handle gray-scale images, the insights gained from this project will illustrate key aspects of image and video compression.

The rest of the project is organized as follows. JPEG is described in Section 3. A detailed list of tasks we want you to work on appears in Section 4. For your convenience, our notation is summarized in Section 5.

*Based on Cisco's Visual Networking Index.

3 JPEG

3.1 Data compression background

The JPEG standard determines how to encode (compress) and later decode (decompress) images. Consider a digital representation of an image. The representation will be in discrete space. Our *input image* is a 2-dimensional (2D) signal, $x \in \mathbb{R}^{M \times N}$, comprised of M rows and N columns of pixels. It may be convenient to think of x as a matrix of size $M \times N$. To keep it simple, we are only considering *gray scale images*, meaning that each pixel is real valued.[†] In contrast, a typical representation for *color images* involves three color planes: red, green, and blue (RGB).

Encoder: The *encoder* converts the input x to a stream of bits, $b \in \{0,1\}^+$, where the superscript plus sign denotes sequences of finite positive length, and each element of the sequence is a bit, i.e., an element of the set $\{0,1\}$.[‡] More formally, the encoder can be interpreted as an encoding function f that maps the matrix input to a bit sequence or bit string,

$$f : \mathbb{R}^{M \times N} \to \{0,1\}^+.$$

Decoder: The *decoder* function g maps the bit sequence back to a matrix,

$$g : \{0,1\}^+ \to \mathbb{R}^{M \times N}.$$

The *output* of the decoder, $\widehat{x}$, is also a 2D image,

$$\widehat{x} = g(f(x)) \in \mathbb{R}^{M \times N},$$

and can be interpreted as an *approximation* of the input image x.

Distortion: The output $\widehat{x}$ will often differ from the input x. The reason for this difference is that the encoder's input space – all possible matrices comprised of MN real numbers – is larger than the encoder's output space of finite-length bit strings. Therefore, there could be many inputs x that map to the same bit string b, yet those inputs all map to a single output, $\widehat{x} = g(b)$.

Our goal is to compress x well, meaning that we want the bit string $b = f(x)$ to be short. At the same time, we want the output image, $\widehat{x} = g(b) = g(f(x))$, to resemble x. This brings up the concept of *distortion* (difference or error) between x and $\widehat{x}$. Ideally, the distortion is small when the output image $\widehat{x}$ looks similar to the input x, and large when they look different. A perceptual notion of distortion involves people looking at test images and scoring their perceived differences; this is a difficult and subjective process! In contrast, most research and development on image and video processing uses mathematically tractable distortion measures. One common distortion measure is the *mean squared error* (MSE),

$$D(x, \widehat{x}) = \frac{1}{MN} \sum_{m=1}^{M} \sum_{n=1}^{N} (x_{m,n} - \widehat{x}_{m,n})^2, \tag{1}$$

where $x_{m,n}$ and $\widehat{x}_{m,n}$ are pixel values of the input and output images, x and $\widehat{x}$, respectively, at row $m \in \{1, \ldots, M\}$ and column $n \in \{1, \ldots, N\}$. Calculating the distortion $D(\cdot, \cdot)$ involves squaring pixel-wise errors between x and $\widehat{x}$, summing the squared errors, and normalizing by MN.

[†] In practice, after loading an image into MATLAB, you may discover that the pixels have integer values. These formats can be converted to real values using MATLAB's double command.

[‡] Intuitively speaking, an empty sequence conveys no information, and we only consider mapping to positive-length sequences. Similarly, an infinitely long sequence is impractical.

Coding rate: We want to describe x using a modest number of bits while achieving a low distortion with respect to $\widehat{x} = g(f(x))$. That said, there is a trade-off between the number of bits used to describe x and the distortion, $D(x, \widehat{x})$. To quantify this trade-off, define the *coding rate* R,

$$R(x) = \frac{|b|}{MN} = \frac{|f(x)|}{MN}, \tag{2}$$

where $|b|$ is the length of the bit string $b = f(x)$ used to encode x. Similar to our definition of distortion (1), the rate is normalized by MN. That is, $D(x, \widehat{x})$ is the mean (average) per-pixel squared error, and $R(x)$ is the average per-pixel coding rate.

3.2 Image patches

JPEG does not process the entire image at once. Instead, it partitions the image into patches, where each block is comprised of 8×8 pixels. For example, a 128×128 image is partitioned into 16×16 patches, because $128 = 8 \times 16$. We denote the number of pixels in a patch by P; P is typically $8 \times 8 = 64$. Because x is comprised of MN pixels,[§] the number of patches is MN/P. Patch number p is denoted by x_p, where $p \in \{1, \ldots, MN/P\}$.

3.3 Discrete cosine transform

In ECE 421 we study the *discrete Fourier transform* (DFT), which computes Fourier coefficients for a finite-length block. One could first apply the DFT to each row of an image patch, and then to columns, leading to the 2D DFT. Unfortunately, the 2D DFT is impractical for image patches, because pixel values are real valued, and it is somewhat complicated to process complex valued Fourier coefficients. In contrast, the *discrete cosine transform* (DCT) is a real valued linear transform, meaning that DCT coefficients computed for real valued patches are real valued.

JPEG computes X_p, the 2D DCT of each image patch x_p. Similar to the DFT, DCT coefficients indicate how much energy the patch contains at different frequencies. For many image patches, much of the energy in the coefficients lies in coefficients that correspond to lower frequencies. As a toy example, consider the following 4×4 image patch,

$$x_p = \begin{bmatrix} 245 & 236 & 231 & 233 \\ 232 & 240 & 234 & 237 \\ 143 & 243 & 232 & 239 \\ 112 & 213 & 240 & 242 \end{bmatrix},$$

which was taken from an image we processed. The 2D DCT coefficients of this patch are also of size 4×4, and take the following values,

$$X_p = \begin{bmatrix} 888.00 & -72.21 & -46.50 & -28.00 \\ 56.71 & 81.98 & 47.84 & 18.99 \\ -12.00 & -10.83 & 2.50 & 10.44 \\ -9.42 & -10.51 & -13.86 & -13.48 \end{bmatrix},$$

where the capital notation (i.e., X_p instead of x_p) denotes DCT coefficients corresponding to x_p. The coefficient with largest magnitude is the 888 in row $m = 1$ and column $n = 1$, corresponding to the DC frequency for the patch; the DC coefficient often has the largest magnitude among the 2D DCT coefficients. Other coefficients with large magnitudes are 81.98, -72.21, 56.71, 47.84, and -46.50. You can see that these tend to have lower m and n indices; magnitudes tend to decay as

[§]To keep things simple, suppose that M and N divide 8 evenly; this will be implemented in Section 4.

m and n are increased. Therefore, we make two observations. First, the 2D DCT is a *sparsifying transform*, meaning that the coefficients will be sparse; typically few coefficients are large. Indeed, it is well known that the DCT has asymptotically optimal energy compaction for some classes of smooth signals. Second, the large coefficients tend to cluster at small m and n.

3.4 Encoding DCT coefficients

Quantization: The DCT coefficients are real valued, yet we want to encode them using a limited number of bits. Therefore, we *quantize* or discretize them to a finite number of levels. For example, we can round our 4×4 matrix DCT coefficients to the nearest multiple of 30,

$$\widehat{X}_p = \begin{bmatrix} 900 & -60 & -60 & -30 \\ 60 & 90 & 60 & 30 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \tag{3}$$

where $\widehat{X}_p$ denotes the approximate DCT coefficients for patch p.

Formally, the quantizer divides the coefficients X_p by the *quantizer step size*, $\Delta = 30$, and rounds to the nearest integer,

$$q(\text{coeff}) = \text{round}(\text{coeff}/\Delta). \tag{4}$$

Similarly, $q(X_p)$ is the quantized version of DCT coefficients of the entire patch, X_p,

$$q(X_p) = \begin{bmatrix} 30 & -2 & -2 & -1 \\ 2 & 3 & 2 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}. \tag{5}$$

In our example, (5) contains a 4×4 patch of quantized DCT coefficients. We will later see how these integers are encoded (compressed) to bits, then decoded (decompressed) back to integers, and we obtain the approximate coefficient by multiplying $q(\text{coeff})$ by Δ,

$$\widehat{\text{coeff}} = \Delta \cdot q(\text{coeff}).$$

Similarly, $\widehat{X_p}$ are approximate DCT coefficients of the entire patch, X_p. In our example, (3) contains a 4×4 patch of approximate DCT coefficients, $\widehat{X}_p$.

Zigzag scan: Because large coefficients tend to be clustered at lower frequencies with small m and n, we process the DCT coefficients of a patch in a *zigzag scan* pattern. The zigzag scan converts $q(X_p)$ (in our example of size 4×4) into a sequence of numbers (16 in our example). The order of the scan follows the numbers from 1 to 16 below,

$$\begin{bmatrix} 1 & 2 & 6 & 7 \\ 3 & 5 & 8 & 13 \\ 4 & 9 & 12 & 14 \\ 10 & 11 & 15 & 16 \end{bmatrix}.$$

The order in which we scan the diagonals swaps directions (we zig, then zag). In our example (5), these 16 numbers are

$$30, -2, 2, 0, 3, -2, -1, 2, 0, 0, 0, 0, 1, 0, 0, 0. \tag{6}$$

Run length coding: Our example (6) contains runs of repeated zeros, for example the last three numbers. The prevalence of zeros is even more pronounced when the image patch is larger

(JPEG uses 8×8 patches while our toy example is 4×4) and the quantizer step size is large (increasing Δ quantizes more coefficients to zero). Additionally, because the nonzeros tend to concentrate at lower frequencies, and thus appear mostly at the beginning of the zigzag scan, there are often long runs of zeros later. *Run length coding* encodes the lengths of runs of zeros, and often describes many zeros with just a few bits. *In this project you will not perform run length coding.*

Conversion to bits: The last step of JPEG converts a sequence of integers to bits. One way to do so uses *Huffman coding*, which converts each symbol of the input sequence into a bit string, and concatenates bit strings corresponding to all the symbols into a longer string.

In our project we instead use *arithmic codes*. An arithmetic code also processes a sequence symbol by symbol. Denote the sequence of quantized DCT coefficients by $q(X_{p,1}), q(X_{p,2}), \ldots, q(X_{p,P})$ (recall that P is the number of pixels per image patch, which equals the number of corresponding DCT coefficients). Whereas Huffman codes convert each symbol $q(X_{p,i})$ into a short bit string, we assign $q(X_{p,i})$ a probablity, $\Pr(q(X_{p,i}))$, and the overall probability corresponding to the entire sequence is the product of individual symbol probabilities,

$$\Pr(q(X_p)) = \prod_{i=1}^{P} \Pr(q(X_{p,i})).$$

One can imagine slices of a pie of probabilities, and as we process more symbols, the slice becomes thinner. Before any symbols have been processed, the slice has probability 1; after one symbol, it has size (probability) $\Pr(q(X_{p,1}))$; after two, $\Pr(q(X_{p,1}))\Pr(q(X_{p,2}))$, and so on. The arithmetic code uses bits to specify a number that fits within the slice. Because each bit halves the numerical range being specified, the number of bits required to encode the final slice is roughly

$$-\log_2(\Pr(q(X_p))) = -\log_2\left(\prod_{i=1}^{P} \Pr(q(X_{p,i}))\right) = -\sum_{i=1}^{P} \log_2(\Pr(q(X_{p,i}))),$$

where $\log_2(\cdot)$ denotes the base-2 logarithm, and we employ the additive property of logarithms, i.e., $\log_2(\alpha\beta) = \log_2(\alpha) + \log_2(\beta)$.

4 Tasks

Having surveyed the main steps involved in JPEG, you will put together a simple image compression system and test it. Please perform all tasks below with two images. The first image is a well known test image called Peppers, which you can download here:

`http://sipi.usc.edu/database/database.php?volume=misc&image=15#top.`

The second image can be any image you want to use, preferably of the person or people submitting the project. We are posting example solutions for Peppers, which should help you check that your implementation is reasonable. *Your job is to provide MATLAB code for all steps below in your report, and results for your second image.*

1. **Load image:** Your first step will be to load the images into MATLAB, convert them into gray scale images, normalize the images, and print them. Use the following command to load the images in gray scale format:

   ```
   image = rgb2gray(imread(file_name))
   ```

The color image loaded by the imread command will be converted to a grayscale image by the rgb2gray command. Next, use the mat2gray command to normalize the pixel values of the image to the range [0,1], and print the normalized image using the imshow command. Note that imshow will scale the image before displaying, and can be used to display normalized images. (Please provide a plot of your second image.)

2. **Partition to patches:** Convert your images into 8×8 patches. To do so, first make sure that each image partitions into patches without spare rows and columns. For example, if x has size 20×33, you can convert it into an image of size 16×32, which is later converted into 2×4 patches, as follows,

   ```
   x = x(1:16,1:32);
   ```

   Next, convert the properly sized images into a collection of patches. In our example, there will be $2 \times 4 = 8$ patches in total, where we note in passing that the number of patches can also be computed by dividing numbers of pixels, i.e., $MN/P = 16 \times 32/(8 \times 8) = 512/64 = 8$. (You may find MATLAB's reshape command helpful in converting an 8×8 matrix into a vector, and possibly converting vectors back to image patches in later parts of this project.) Please provide your code for creating patches, and print 2–3 patches from your second image.

3. **Discrete cosine transform:** Apply MATLAB's dct2 command to each of the patches. Note that the 2D DCT coefficients corresponding to an 8×8 image patch are also of size 8×8. We discussed earlier how DCT coefficients are often sparse with more energy at lower frequencies. To evaluate these properties, plot the *sorted* magnitudes (absolute values) of 2D DCT coefficients for the patches of Task 2. Your MATLAB code could resemble

   ```
   semilogy(sort(reshape(abs(dct_coeff), 64, 1)), '*');
   ```

   The semilogy command is useful for showing values that are orders of magnitude apart, and sorting will make the plot easier to interpret visually.

   In addition to plotting magnitudes of 2D DCT coefficients of individual patches, we will illustrate properties of DCT coefficients statistically over thousands of patches. To do so, (*i*) compute the DCT coefficients for all the patches; (*ii*) compute the average magnitude for each of the $P = 64$ coefficients;[¶] and (*iii*) for your second image, plot the average magnitude of the 64 coefficients using a command such as surf or mesh. You should see that the average energy decays as we move to higher frequencies.

4. **Quantization:** For each patch p, quantize the corresponding DCT coefficients, X_p, using (4). When Δ is larger, more coefficients are quantized to zero, and the quantization error should be larger.[‖] To see this, recall our definition of distortion (1). After quantizing the coefficients, $q(X_p)$ will be encoded to bits (Task 5 of the project), decoded to $\widehat{X}_p$, converted via the inverse DCT back to a new patch, $\widehat{x}_p$, and collecting all the patches will result in our output image, $\widehat{x}$. Because the DCT coefficients are quantized, the decoded coefficients, $\widehat{X}_p$, will differ from X_p, and when we compute $\widehat{x}$ using the inverse DCT, $\widehat{x}$ will differ from x. Below you will examine the trade-off between Δ and $D(x, \widehat{x})$.

[¶] Not |average(coeffs)|, which could be quite small due to averaging positive and negative numbers, but average(|coeffs|).

[‖] Not only will the distortion increase as more nonzero real valued coefficients are quantized to zero, but larger nonzeros will be quantized more coarsely.

Please select several (more than 2) values of Δ such that the percentages of nonzero quantized coefficients – averaged over thousands of image patches – will vary from roughly 1% (this requires a relatively large Δ) to around 50% (the smallest Δ). After the DCT coefficients of all the image patches have been quantized, apply the inverse DCT, idct2, to produce the decoded image, $\widehat{x}$. For your second image, list in a table the step sizes, corresponding percentages of nonzero coefficients, and distortions between x and $\widehat{x}$. (MATLAB's nnz command can count the number of nonzeros.) You should see that smaller Δ reduces the distortion. Additionally, please plot $\widehat{x}$ for several step sizes. You may want to provide a close-up of parts of x compared to corresponding parts of $\widehat{x}$, in order to highlight the loss of quality due to quantization.

5. **Arithmetic coding:** We will encode MN/P quantized patches, $q(X_p)$. Consider how each $q(X_p)$ is comprised of 64 DCT coefficients,

$$q(X_p) = (q(X_{p,1}), \ldots, q(X_{p,i}), \ldots, q(X_{p,P})),$$

where $q(X_{p,i})$ is DCT coefficient i within $q(X_p)$. We group the MN coefficients by i. For each $i \in \{1, \ldots, 64\}$, we call the MN/P quantized coefficients a *coefficient plane*,

$$\text{plane}_i = (q(X_{p=1,i}), \ldots, q(X_{p=MN/P,i})).$$

We will encode plane_1, then plane_2, and so on up to plane_{64}.**

Each plane, plane_i, is processed by an arithmetic encoder. MATLAB has functions that support arithmetic encoding and decoding,

`https://www.mathworks.com/help/comm/ug/arithmetic-coding-1.html`

The coefficient plane can be processed as follows,

```
code_i = arithenco(plane_i, counts_i); % encoder
decode_i = arithdeco(code_i, counts_i, length(plane_i)); % decoder
isequal(plane_i, decode_i) % outputs 0 or 1
```

where we first encode plane_i to $\text{code}_i \in \{0,1\}^+$, next we decode code_i to decode_i, and finally we check that plane_i was decoded correctly.

The counts variable appears in MATLAB's commands for both arithmetic encoding and decoding. The encoder and decoder assume that the plane contains integers whose values start at 1, which can be ensured by adding $1 - \min(\text{plane}_i)$ to the MN/P numbers in plane_i. We then count how many times each integer appears in plane_i; note that MATLAB expects all counts to be at least 1, and so we add 1 to all the counts using MATLAB's hist command,

```
plane_i_modified = plane_i + 1 - min(plane_i);
counts = hist(plane_i_modified, 1:max(plane_i_modified)) + 1;
```

For each step size Δ considered in Task 4, please compute the lengths of codes for all the planes, i.e., $\text{length}(\text{code}_i)$ for $i \in \{1, 2, \ldots, 64\}$. Summing the 64 coding lengths, you will obtain an overall length required to encode the image x using Δ. When Δ is large, the quantization will be coarse including plenty of zeros, and we expect the coding length to be short. Smaller step sizes induce fine quantization and larger coding lengths. At the same time, Step 4 showed that large Δ increases the distortion. Therefore, there is a trade-off between the coding rate $R(x)$ (2) and distortion $D(x, \widehat{x})$. For your second image, please plot a rate-distortion curve showing the trade-off between R and D.

**We encode each plane separately instead of each $q(X_p)$ separately, because coefficients within each plane presumably have a similar statistical distribution, and data compression is more efficient when processing symbols that follow the same distribution.

5 Notation

- x input image.
- M rows in x.
- N columns in x.
- b bit string output by the encoder.
- f encoding function.
- g decoding function.
- $\widehat{x}$ output image.
- $D(x, \widehat{x})$ distortion between x and $\widehat{x}$.
- $m \in \{1, \ldots, M\}$ row index.
- $n \in \{1, \ldots, N\}$ column index.
- $x_{m,n}$ pixel value in row m and column n.
- $|\cdot|$ length operator.
- $R(x)$ coding rate.
- P number of pixels or DCT coefficients in a patch.
- p patch number.
- x_p image patch.
- X_p DCT coefficients corresponding to x_p.
- $\widehat{X}_p$ approximate coefficients at the decoder.
- $q(X_p)$ quantized DCT coefficients.
- plane_i quantized DCT coefficient i among all MN/P patches.

6 Mathematical Problem

The following problem is unrelated to the project, and involves working out some math, which will allow us to provide you with feedback.

Question 1. Consider the signal $x(n) = \{2, 1, \underline{-3}, 6, 0, 4\}$ with Fourier transform $X(\omega)$. Compute the following quantities without explicitly computing $H(\omega)$.

1. $X(0)$.
2. $\int_{-\pi}^{\pi} X(\omega) d\omega$.
3. $X(\pi)$.
4. $\int_{-\pi}^{\pi} |X(\omega)|^2 d\omega$.