Introduction

• Images use lots of memory to store and lots of bandwidth to transmit
• How can we store them compactly?
• There are 3 redundancies in images: coding, interpixel, and psychovisual
• We wish to remove the redundancies to save space.
Introduction (2)

- Image compression is not "built-in" in MATLAB.
- We will be looking at code to compress images, both MATLAB code and C code called from MATLAB.
- Bring your book!

Image Compression Systems

- Two blocks: encoder and decoder.
- \( f(x,y) \rightarrow \text{encoder} \rightarrow c(x,y) \)
- \( c(x,y) \rightarrow \text{decoder} \rightarrow F(x,y) \)
- \( F(x,y) = f(x,y) \): lossless compression
- \( F(x,y) \neq f(x,y) \): lossy compression
Compression Systems (2)

- Scientific or medical images: preserving original data important.
- Pictures: original data not important as long as visual quality is ok.
- For image compression, the compression ratio $c_R$ of the image is important: $c_R = \# \text{ bits original}/\# \text{ bits compressed}$. We want $c_R > 1.0$.

MATLAB Code pp. 283 - 284

- The code to calculate the compression ratio uses function `bytes(arg)` to calculate the number of bytes in the 2 images. Function `imratio` divides bytes in plain image by bytes in encoded image to get the ratio.
- If `arg` is a string (image filename), the `dir` function gets the number of bytes.
MATLAB Code pp. 283 - 284

• If `arg` is a structure variable, the code adds up the lengths of the fields. The function `fieldnames` returns the names of the fields.
• If `arg` is something else, the `whos` function calculates the bytes.

MATLAB Code pp. 283 - 284

• Usage (p. 284):
  • `r = imratio (imread('imagename'), 'imagename')` prints `r = 35.1612` for image `bubbles25.jpg`
Error in Lossy Compression

• In lossy compression, the decoded image $F$ is different from the original image $f$. The error at pixel $(x,y)$ is $e(x,y) = F(x,y) - f(x,y)$.

• The total error is found by summing the pixel errors from $y = 0$ to $N-1$ and from $x = 0$ to $M - 1$ (p. 285).

Error in Lossy Compression (2)

• The root-mean-square error is found as follows:
  • calculate the square of the error for each pixel and add them all up
  • divide the result by the image size $MN$
  • take the square root of the result
Error in Lossy Compression (3)

- Code for function *compare* is on pp. 285 – 286. It computes the root-mean-square error and, if it is not 0, outputs an error image scaled from $-\text{emax}$ to $\text{emax}$ (the max error in any pixel) scaled by any desired value (default 1), as well as an error image histogram.

Image Encoder

- The encoder has 3 parts:
  - a mapper, which reduces interpixel redundancy
  - a quantizer, which eliminates psychovisual redundancy
  - a symbol coder, which eliminates coding redundancy
Image Decoder

• The image decoder has 2 parts:
  – a symbol decoder, which reverses the actions of the encoder
  – an inverse mapper, which reverses the actions of the mapper.
• Note that quantizing cannot be undone.

What Next?

• We will look first at symbol encoding and decoding (section 8.2).
• We will look next at interpixel redundancy (section 8.3).
• After that, we will look at image quantization (section 8.4).
• Finally we will look briefly at JPEG examples (sections 8.5 and 8.6).
8.2 Coding Redundancy

- A grey-level image with levels 0 to 255 can be represented by an array of 8-bit bytes.
- An $M \times N$ image can thus be represented using $8MN$ bits.
- Using Huffman coding, we can reduce the number of bits required to store the image.

Huffman Coding

- The idea is to assign frequently-occurring numbers a small number of bits, and less frequently occurring numbers a larger number of bits.
- In the usual case, this will reduce the number of bits required to store the $M \times N$ image.
Huffman Coding (2)

- Let $r_k$ be a discrete random variable for $k = 1, 2, \ldots, L$. $r_k$ represents grey level $k$.
- We count how often $r_k$ occurs in the image ($n_k$ times) and let its probability $p(r_k) = n_k/n$, where $n = M \times N$.

Huffman Coding (3)

- $N(r_k)$ = number of bits used to encode $r_k$. Then the average number of bits required to encode $L$ bits is $L_{\text{avg}} = \text{sum from 1 to } L(N(r_k)p(r_k))$, and the number of bits to encode the $M \times N$ image is $B = MNL_{\text{avg}}$. 
Example (p. 287, Table 8.1)

- In this example, there are 4 grey levels.
- Code 1 uses 2 bits for each pixel, so $L_{avg} = 2$.
- Code 2 uses different number of bits for different levels: 3, 1, 3, and 2. Its $L_{avg} = 3 \times 0.1875 + 1 \times 0.5 + 3 \times 0.125 + 2 \times 0.1875 = 1.8125$ bits.

Example (continued)

- The compression ratio is $2/1.8125 = 1.103$.
- A 4 x 4 image using the 4 grey levels with the 2-bit code would take 32 bits (4 x 4 x 2).
- A 4 x 4 image using code 2 would take 29 bits.
What is the Smallest Number of Bits Required?

• Information theory tells us that a random event $E$ (grey level) with an associated probability $P(E)$ contains $-\log P(E)$ units of information.

• Given a set of random events $a_i$, $i = 1, 2, \ldots, j$, with associated probabilities $P(a_k)$, the average information per source output $=$ the entropy $H$.

Smallest Number of Bits (2)

• Entropy $H = - \sum P(a_j) \log(P(a_j))$ (summed from $j = 1$ to $J$).

• We can interpret a grey-level image histogram as giving a first-order estimate of the entropy $H$.

• The code on p. 288 computes an entropy estimate of image $x$, with $n$ grey levels (default 256).
Smallest number of Bits (3)

• The code computes an n-bin histogram of \( x \), divides by the total number of pixels (\( \text{sum}(xh(:)) \)), eliminates 0 values (whose log is \(-\infty\)), and computes the entropy.
• MATLAB function \( \text{find}(xh) \) finds only non-zero elements

Example, pp. 288 - 289

• The image has 4 grey levels, 107, 119, 123, and 168, with probabilities 0.1875, 0.5, 0.125, and 0.1875.
• The bin size is 8, and 8 values must be used to include the values 107 through 168 in evenly sized bins.
• Entropy is 1.7806, so the Huffman code of 1.8125 is not far off the best.
Properties of Huffman Codes

- Huffman code contains the smallest possible number of bits to represent each value, assuming values are encoded one at a time.
- In English, "qu" always appear together; this is a case where you might want to combine 2 letters into 1 code.

Constructing Huffman Codes

- 1. Compute probabilities for each value.
- 2. Arrange the probabilities in descending order.
- 3. Reduce: combine two least probabilities into a single symbol (i.e., construct a binary tree).
Constructing Huffman Codes (2)

• 4. Use a left branch to denote a 1, and a right branch for 0 (or vice versa).
• There are multiple Huffman trees which can be constructed this way. Each tree, however, will have the same total code length, even if values are encoded differently.

Example (pp. 288 – 290)

• Image is 4 x 4, has 16 values
• # 107 = 3, probability \( p_1 = \frac{3}{16} = 0.1875 \)
• # 119 = 8, \( p_2 = \frac{8}{16} = 0.5 \)
• # 123 = 2, \( p_3 = \frac{2}{16} = 0.125 \)
• # 168 = 3, \( p_4 = \frac{3}{16} = 0.1875 \)
Example (cont.)

• In descending order, the probabilities are 0.5 (119), 0.1875 (107 and 168), and 0.125 (123).
• Two trees can be made from these values, depending on whether the larger value is the left leaf (book's tree) or the right leaf (my tree).
Which Tree To Use?

• It doesn't matter, as long as you decode the same tree you encoded.
• The code lengths will be the same in the 2 trees. The code values will be different.
• Book: 119 - 1, 107 - 00, 123 - 010, 168 - 011; Mine: 119 - 1, 107 - 01, 123 - 001, 168 - 000

Compression Ratio

• Original image requires $4 \times 4 \times 8 = 128$ bits, or $4 \times 4 \times 2 = 32$ bits if we code only 4 values.
• Compressed image requires $4 \times 4 \times 1.8125 = 29$ bits
Program to Find the Code

- The MATLAB program to create the Huffman code is on pp. 290 - 292 of your text. It returns the code as a binary string in CODE, where CODE[i] is the code for the value whose probability is at P[i], where P[i] is an input argument, as is the original image f.

Program Details

- CODE is a global variable, which must be declared in any function which uses it, but for which only 1 copy exists.
- The program operates by finding the probabilities, reducing (making the tree), and then making the code, all as we described before.
Program Details (2)

- The cell (m,n) function is used to create an M x N array of empty matrices which can be referred to by cell number or by content. \( X = \text{cell}(M,N) \) declares the array. \( X(1) = [\ ] \) indexes and removes element 1 from the array. \( X\{1\} = [\ ] \) assigns [ ] to element 1.

Program Details (3)

- The while loop constructs the tree. Reduce( ) sorts the probabilities, merges the 2 lowest, and prunes the values from the list.
Program Details (4)

• Makecode( ) traverses the tree to make the code. It takes as arguments the codeword (an array of 0s and 1s) and a cell array element sc.
  –If sc is a cell array, it contains the 2 joined symbols, and so 2 recursive calls are done.

Program Details (5)

–On the right side, a call adds 0 to the codeword; on the left, it adds a 1.
• Example: makecode initializes sc to [], and then puts the values in the cell array: sc = {119, {107, {123, 168}}}
Example (2)

- call left assigns 1 to 119, and call right assigns 0 to each of the rest
- 119 is removed, and call left adds 1 to 107 (making it 01); call right adds 0 to the rest, making them 00.
- 107 is removed, and call left adds 1 to 123 (making it 001); call right adds 0 to 168, making it 000.

Example (3)

- All elements are now gone, and the code has been created.
  - The encoded image is 10010001 0011000000 11011 010111, 29 bits.
  - `Dec2bin(double h3f2.code)` prints the binary code, filling in with 3 0s at the end.
Example (5)

- Note that no code is a prefix of any other code.

Using the Huffman Code

- Finding the Huffman code does not compress the image.
- We must pack the (fewer) bits into bytes, filling in with 0s at the end if necessary.
- We must also save the coding so we can decode the image later.
Using the Huffman Code (2)

• Original image f2 is 4 x 4 x 8 = 128 bits = 16 bytes.
• Packing the bits into bytes is done by the program mat2huff, on pages 298 - 299, which reduces the data storage to 4 bytes (2 16-bit integers)
• Note: the comments for this program describe restrictions on its use.

Using the Huffman Code (3)

• Information necessary for decoding is in y.min, y.hist, and y.size, which are respectively the minimum value of x, the histogram of x, and the dimensions of matrix x.
• mat2huff also requires 514 bytes (about 2^9 bytes) for overhead.
Using the Huffman Code (4)

• A compression ratio of 4 for normal size images makes this amount of storage insignificant.
• $1024 \times 1024 = 2^{20}/4 = 2^{18}$ bytes, and the overhead is only $2^9$ bytes, which is $1/512$ of the image data storage.

Using mat2huff

• $f = \text{imread ("Tracy.tif");}$
• $c = \text{mat2huff (f);}$
• $\text{cr = imratio (f, c) gives cr = 1.2191}$
• $\text{save SqueezeTracy c; \% saves a binary compressed file}$
Huffman Decoding

- Huffman decoding is done by \( x = \text{huff2mat}(y) \); \( y \) was saved as the overhead bytes by mat2huff.
- \text{huff2mat} extracts the dimensions of the original image, its histogram, and its minimum value, and recreates the Huffman code.

Huffman Decoding (2)

- C function \text{unravel( )} decodes the data and then adds the minimum value.
- It uses a binary search table; since codes are unique, one simply extracts characters until a valid code is found.
Huffman Decoding (3)

• extraction is done in the while loop: if a character is found (look == 1), set link to –look, delete current node, and increment number of codes found;
• otherwise, add left and right pointers to the node and add unprocessed nodes. Then concatenate 0 or 1, remove processed nodes, and add 2 unprocessed nodes, and continue.

Huffman Decoding (4)

• Why C? more efficient than MATLAB functions.
• How do we attach C functions to MATLAB?
  –mex filename.c /* produces unravel.dll, a dynamic link library */
Huffman Decoding (5)

– Comments on usage and function of unravel are in unravel.m
– Note that the C function must include "mex.h" to make this work.
– See the summary figure 8.6 on p. 308 of the text.

Usage of huff2mat

• load SqueezeTracy;
• // c is the Huffman compressed image
• g = huff2mat c);
• f = imread ("Tracy.tif");
• rmse = compare (f, g); reports rmse = 0; i.e., images are identical.
8.3 Interpixel Redundancy

• In an image, generally a pixel’s value will be close to the values of pixels near it. That is, the value of a pixel can be reliably guessed from the values of its neighbors.
• We can use this property to compress an image.

Interpixel Redundancy (2)

• For example, on p. 208 of your text there are 2 figures and their histograms. The entropies and histograms are similar, but the Huffman coding compression ratio is only 1.07 to 1.08.
• General idea: to remove interpixel redundancy, we create a transformed (mapped) image.
Interpixel Redundancy (3)

• The mapping may be reversible: original $\leftrightarrow$ map; or the mapping may be irreversible: original $\rightarrow$ map.

• For example, we may store differences between adjacent pixels rather than the pixel values themselves.

Lossless Predictive Encoding and Decoding

• We input image fn. We also have a predictor function which predicts the next pixel value and rounds the prediction to the nearest int, fn*. Then the error fn - fn* is stored as the pixel value. The result is now Huffman coded.
Lossless Predictive Encoding and Decoding (2)

• After the image is Huffman decoded, an identical predictor function predicts pixel values \( f_{n*} \), and the error value is added to the predicted value.
  – A similar process is used when streaming video is sent over the Web: only differences between previous images are sent.

Lossless Predictive Encoding and Decoding (3)

• A linear predictor is generally used: 
  \[ f_{n*} = \text{round} \left[ \sum (\alpha_i f_{n-i}) \right] \] 
  \( M \) is called the order of the predictor, and \( \alpha_i \) are the prediction coefficients.

• A one-dimensional image (a signal) is predicted as 
  \[ f_{n*}(x, y) = \text{round} \left[ \sum (\alpha_i f(x, y-i)) \right] \]
Programs for Predictive Encoding and Decoding

• mat2lpc and lpc2mat (those are letter Ls, not number 1s) on pp. 312 - 313 of your text do the predictive encoding and decoding (not the Huffman part).

Programs (2)

• Example: image 8.7.c is encoded using a size 1 linear predictor in 1 dimension (i.e., coding is done for each scan line of the image, not in 2-d). The error image and the histogram image are shown in Figure 8.9, page 313 of your text.
8.4 Psychovisual Redundancy

- Eliminating psychovisual redundancy always causes loss of original data! Do not do this for scientific or medical images.
- Compression using psychovisual redundancy removes information not essential for normal visual processing - *i.e.*, for viewing.

Programs (3)

- The entropy of the error image has decreased from 7.3505 to 5.9727
- The Huffman compression ratio is now $cr = 1.33$, instead of 1.08.
- To demonstrate lossless coding:
  - $g = \text{lpc2mat (huff2mat (c))};$
  - compare ($f$, $g$) yields 0 difference
Psychovisual Redundancy (2)

- The compression method is referred to as quantization (combining grey levels to reduce their number).
- Example: Figure 8.10, p. 315 of your text, shows an original image (256 grey levels), a compressed image (16 grey levels), and an image IGS compressed to 16 grey levels.

Psychovisual Redundancy (3)

- Note that spurious edges occur in the second image, in the vase, where the grey level suddenly changes.
- Note that in the 3rd image (IGS, Improved Grey Scale), there is some fuzziness in the edge at the top of the vase.
IGS Quantization

- Uses the high sensitivity of the human eye to edges: edges are broken up by the addition of a pseudo-random number generated from the low-order bits of adjacent pixels.
- It smooths edges (compare 8.10.b and 8.10.c on the surface of the vase), but it also fuzzes the sharp edges at the top of the vase. (p. 315)

Program to Quantize

- A function quantize( ) to quantize an image is on p. 316 of your text. Its input is the image, the number of bits to quantize to, and the type (either none or 'igs').
- The program creates 2 bit masks, \( lo = 28 - b - 1 \) and \( hi = 28 - b - 1 \)
Program (2)

• Either use masks to suppress the low-order b bits, or work down columns in the image:
  – if MSBs of pixels are all 1, set sum to pixel values;
  – else add pixel values to LSBs of previous sum, and take MSBs of sum as quantized values.

Program (3)

– code: s = x(:, j) and y(:, j) = ...

• This procedure would normally be followed by removal of interpixel redundancy and Huffman coding.
• Code is on the next slide.
Usage of Programs

- \( f = \text{imread} \("..."\); \)
- \( q = \text{quantize} \,(f, \, 4, \,'igs'); \)
- \( qs = \text{double} \,(q)/16; \)
- \( e = \text{mat2lpc} \,(qs); \)
- \( c = \text{mat2huff} \,(e); \)
- \( cr = \text{imratio} \,(f, \, c) \text{ gives } 4.1420 \)

"Recovering" the Image

- \( e = \text{huff2mat} \,(c); \)
- \( qs = \text{lpc2mat} \,(e); \)
- \( nq = 16 \,* \,qs; \)
- \( \text{compare} \,(q, \, nq) \text{ gives } 0. \)
- \( \text{rmse} = \text{compare} \,(f, \, nq) \text{ gives } 6.84, \text{ all from quantization.} \)
8.5 JPEG Compression

• JPEG stands for Joint Photographic Experts Group, who invented this compression method. **It is lossy!**
  – It works only for still images. If you want to compress movies or video, use MPEG compression.

JPEG (2)

• JPEG compression operates on a discrete image transform, for example, the Discrete Fourier Transform or the Discrete Cosine Transform (page 318).
• Image values are required to be 8 bits, and transform coefficients are 11 bits.
JPEG (3)

- Process: encoding
  - The image is divided into 8 x 8 blocks by an extractor. Then the DCT is used on the image. Then the image is normalized and quantized, and finally the symbols are Huffman coded.

JPEG (4)

- Process: decoding
  - The symbols are decoded, the image is denormalized, the inverse DCT is applied, and the 8 x 8 blocks are merged.
Details of JPEG

- Split image into non-overlapped 8 x 8 blocks, which are processed left to right and top to bottom. Pad if necessary to fill out the last blocks.
- Shift pixel levels by subtracting $2^{m-1}$, where $m$ is the number of grey levels in the image. Then apply DCT to each block.

Details of JPEG (2)

- Normalize and quantize DCT coefficients using $T^*(u, v) = \text{round} \left[ T(u, v)/Z(u, v) \right]$, where $Z$ is given in Fig. 8.12 a, p. 318.
- Re-order elements of $T$ using the zig-zag pattern of Fig. 8.12 b, p. 318, which produces a 1-D array, generally with long runs of 0s in it.
Details of JPEG (3)

• symbol coding: The $T^* (u, v)$ are encoded for $u, v > 0$, as a difference between this $T^*$ and the previous one, and a run of 0s is encoded by its length.
• This is called an arithmetic encoding.

Program Approximating JPEG

• A program approximating this operation is given on pp. 319 - 321 of your book. `Im2jpeg()` takes an input an image and an estimate of the quality desired. The quality controls the amount of information lost, and the compression achieved.
Program Operation

- The normalizing and zig-zag arrays are set up.
- The number of input lines, level shift, and 8 x 8 DCT coefficient matrix are computed.

Program Operation (2)

- The call \( B = \text{blkproc}(A, [M, N], \text{function}, \text{fn args...}) \) computes DCT and quantizing coefficients for a block of the image.
- The call \( B = \text{im2col}(A, [M, N], 'distinct') \) outputs a matrix: each column contains elements of 1 distinct block. It allows vectorized operations, reorders the blocks and detects runs of 0s.
Difference from "Real" JPEG

• Real JPEG detects all runs; this only detects runs at ends of blocks.
• Real JPEG uses arithmetic encoding. The book's approximation uses Huffman coding.

Program Approximating JPEG Decoding

• A program for decoding is given on pp. 322 - 323 of your text. Jpeg2im() does the operations in reverse order, except that it cannot undo the quantization. It calls col2im() (the inverse of im2col()) to deal with image blocks.
Example

• Fig. 8.4, p 300, shows the original image. Fig. 8.13, p. 324, shows the JPEG compression of Fig. 8.4, using the tables of Fig. 8.12 a, p. 319. On the left, the normalization is scaled by 1; on the right, by 4.
• The code used to do this is shown on the next slide.

Usage of JPEG Approximations

• f = imread("Tracy.tif");
• c1 = im2jpeg(f);
• f1 = jpeg2im(c1);
• imratio(f, c1): 18.2  
• compare(f, f1): 2.47  
• Standard JPEG would about double cr.
Usage (2)

• `f = imread("Tracy.tif");`
• `c4 = im2jpeg(f, 4);`
• `f4 = jpeg2im(c4);`
• `imratio(f, c4): 41.783`
• `compare(f4, f): 4.42`
• Standard JPEG would about double cr.

8.6 JPEG 2000

• JPEG 2000 is JPEG with the DCT replaced by a wavelet transform.
• Lossless: bi-orthogonal wavelet transform with 5 - 3 coefficient scaling, lossless wavelet vector.
• Lossy: bi-orthogonal wavelet transform with 9 - 7 coefficient scaling, lossy wavelet vector.
Steps in JPEG 2000 Coding

• level shift (subtract $2^{m-1}$)
• do wavelet transform for rows and columns
  – get 4 sub-bands: low resolution approximation and horizontal, vertical, and diagonal frequency characteristics

Steps (2)

• Repeat the decomposition $N_L$ times, restricted to previous decomposition's approximation coefficients.
• We now have the $N_L$-scale wavelet transform, which contains coefficients for LL, HL, LH, and HH sub-bands.
Steps (3)

• The important visual information will be concentrated in a few coefficients.
  –In lossy compression, we quantize these; otherwise, we leave them alone.

Steps (4)

• Finally, the coefficients are encoded arithmetically on a bit-plane basis, in a variable length encoding much like Huffman coding.
  –The book's approximation, im2jpeg2k, uses Huffman coding instead of arithmetic coding.
Programs

- A program for im2jpeg2k is on pp. 327 - 329.
- A program for jpeg2k2im is on pp. 330 - 331.
- We will not discuss this code in detail.

Example

- Compare the left side of Fig. 8.16, p. 332, to the right side of Fig. 8.13, p. 324.
Usage of Book's Programs

• \( f = \text{imread("Tracy.tif")}; \)
• \( c1 = \text{im2jpeg2k}(f, 5, [8 \ 8.5]); \)  
  (The call implicitly asks for quantization.)
• \( f1 = \text{jpeg2k2im}(c1); \)
• \( \text{rms1} = \text{compare}(f, f1): 3.69 \)
• \( \text{cr1} = \text{imratio}(f, c1): 42.16 \)

Usage (2)

• \( f = \text{imread("Tracy.tif")}; \)
• \( c2 = \text{im2jpeg2k}(f, 5, [8 \ 7]); \)
• \( f2 = \text{jpeg2k2im}(c2); \)
• \( \text{rms2} = \text{compare}(f, f2): 5.92 \)
• \( \text{cr2} = \text{imratio}(f, c2): 87.73 \)