Mobile Systems

Lecture 4 – 17/03/15

Image coding

COMP28512

Steve Furber & Barry Cheetham

Images & video

- Static images are big.
  - Consider 10"x8" colour photo at 300 dots per inch,
  - Has 10×300×8×300 = 7.2 Mpixels
  - Each pixel is a coloured dot needing 3 bytes, for R G B.
  - Image requires: 7.2 M × 3 = 21.6 Mbytes
- Movies are enormous.
  - Consider TV quality video with 640 x 480 pixels per frame & 25 Hz frames/s
  - For colour, need 640×480 × 3 = 0.9218 Mbyte per frame = 23.04 Mbytes/second
  - That is ≈166 Gbytes for a 2 hour movie
  - 166 Gbytes ÷4.7 G ≈ 36 ∴ 36 DVD disks needed.
- Compression is needed!

Physiology

- Human visual system assumed to have 3 colour sensors: red, green & blue
- Cannot resolve more than 8 bits per colour so 3×8 = 24 bits/pixel is acceptable
- Can represent coloured image by 3 components: RGB
- Or by a luminance (monochrome) component & two chrominance (colour) components.
- Human eye less sensitive to chrominance than luminance.
- Also relatively insensitive to rapidly changing (higher frequency) fine-detailed aspects of the image.

Frequency-domain processing

- Allows mobile device to take physiological features into account when digitising images.
- Avoids encoding what the eye will not see by:
  - transforming the image into freq-domain
  - efficiently encoding only the features that will be perceived
  - reversing the transform at the receiver
- As with MP3, Discrete Cosine Transform is used
- But now it must be a 2-dimensional DCT
- In Matlab, 'dct2' is provided by 'Image Proc Toolbox'.
- Or we can use our own versions (BBdct2 & BBidct2)

Discrete Cosine Transform (DCT)

- Given $\{x[n]\}_{0,N-1}$ its '1-D' DCT is:

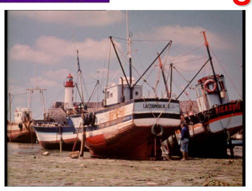
$$DCT[k] = \sum_{n=0}^{N-1} x[n]\cos\left(\left(\frac{2n+1}{2}\right)\Omega_k\right) \quad \text{where} \quad \Omega_k = \pi k / N$$

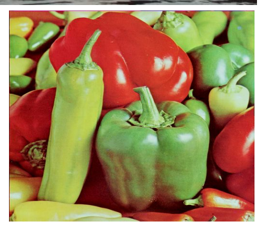
- Given $\{x[n.m]\}_{0,N-1,\ 0,M-1}$ its '2-D' DCT is:

$$DCT[k,\ell] = \sum_{k=0}^{N-1}\sum_{\ell=0}^{M-1} x[n,m]\cos\left(\left(\frac{2n+1}{2}\right)\pi k / N\right)\cos\left(\left(\frac{2m+1}{2}\right)\pi \ell / M\right)$$

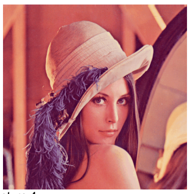
- Low values of k & ℓ correspond to slowly changing features.
- Higher values correspond to fine detail.

Some images





More images



- Find these in course web-site
- All are bit-map (.bmp) not jpg

Red, green & blue

- Any of these images may be read into MATLAB by:
- A = imread('Boats.bmp');
- Creates a 576 x 787 x 3 matrix of unsigned 8-bit integers.
- Can separate into red, green blue as follows:
- R=A(: , : , 1); G = A(: , : , 2); B = A(: , : , 3);
- R, G & B are now 576 x 787 matrices
- For processing it is best to concert these to a:
  - luminance matrix Y
  - chrominance (colour) matrix I
  - chrominance (colour) matrix Q
- See AMimages.ipynb to see how it is done in IPython

Converting from RGB to YIQ (NTSC)

- Could have $Y = 0.33 \times R + 0.33 \times G + 0.33 \times B$
  $C_R = R - Y$ ('Red difference' chrominance)
  $C_G = Y - G$ ('Green difference' chrominance)

Then: $C_R = 0.66R - 0.33G - 0.33B$
and: $C_G = 0.33R - 0.66G + 0.33B$

- Instead: $Y = 0.3\,R + 0.59\,G + 0.11\,B$
  $I = 0.6\,R - 0.28\,G - 0.32\,B$
  $Q = 0.21\,R - 0.52\,G + 0.31B$

(PAL & SECAM use different numbers but similar idea).

In matrix form

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.3 & 0.59 & 0.11 \\ 0.6 & -0.28 & -0.32 \\ 0.21 & -0.52 & 0.31 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

To convert from YIQ back to RGB form, invert the matrix:

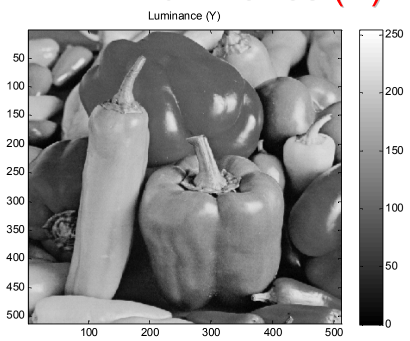
$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.95 & 0.62 \\ 1 & -0.28 & -0.64 \\ 1 & -1.1 & 1.73 \end{bmatrix} \times \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

Demo: 'bigDCT2' method

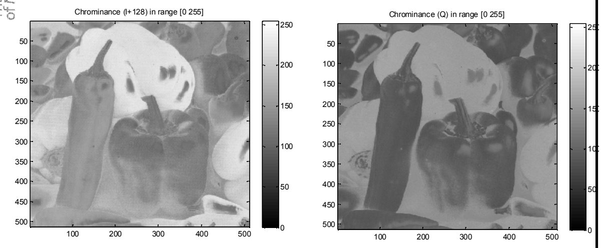
- Apply DCT2 to Y, I & Q
- Huge matrices, but DCT2 is efficient (like fft)
- Run 'Lecture4imageBigDCT2.m' & observe the graphs.
- Note: Have tried to avoid the use of functions in the 'Image Processing Toolbox'.
- Own versions of dct2 & idct2 provided.

peppers.bmp

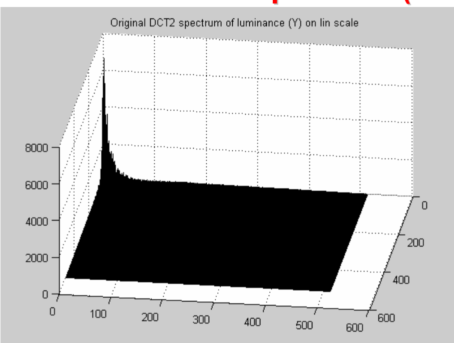
Luminance (Y)



I & Q chominance



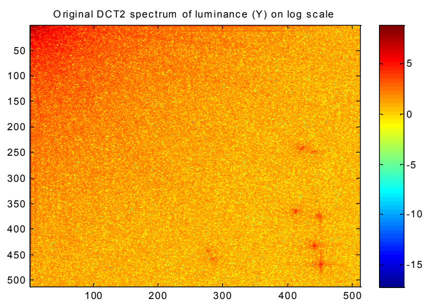
DCT2 Luminance spectrum (lin scale)



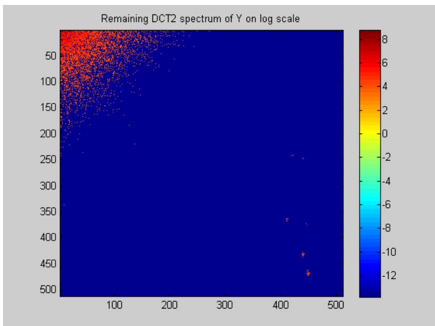
Comment

- Observe how energy is concentrated a the lower frequencies.
- In fact there seems little energy at higher frequencies in either direction.
- Look at the spectrum on a log scale with colour representing amplitude.

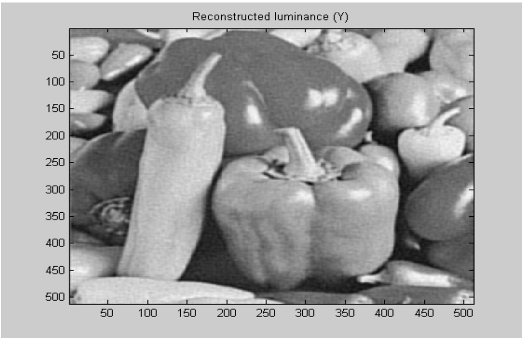
Luminance DCT2 spectrum (log-scale)



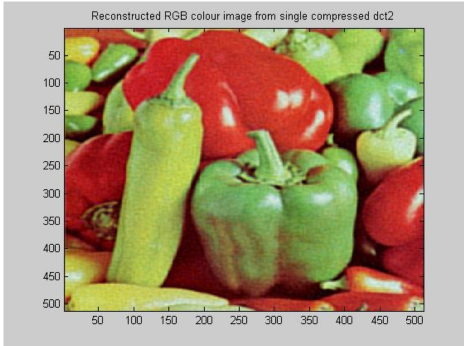
Set small amplitudes to zero



Resulting luminance



Do the same for I & Q



Comments on demo

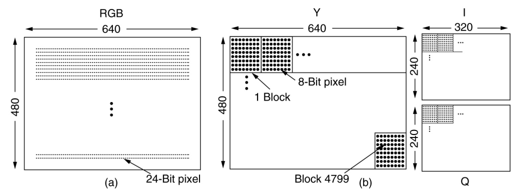
- Coloured picture converted to Y, I & Q.
- Take DCT2 of each & plot mag-spectrum.
- Notice concentration of energy in top corner.
- Set to zero any values < some threshold.
- Creates lots of zeros.
- Go back to an image via an inverse DCT2.
- Can see reconstructed image
  & its modified spectrum (with lots of blue).
- Any perceivable loss of quality?

Data reduction

- Number of non-zero coeffs for Y reduced from 262,000 to 8,692.
- About 3% are left
- All the rest are now zero.
- Similar for I & Q ??
- Send only the non-zero values - quite a reduction!
- Coding the non-zero values is hard – where do they occur?
- Image compression is not done like this.
- Let's see how JPEG does it:

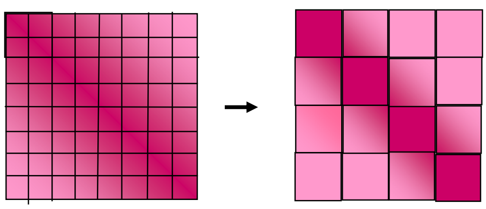
JPEG image compression: Step 1

- Divide image into 8×8 coloured pixel 'tiles'.
- For each tile, convert each RGB pixel to:
  - a measure of luminance (Y) plus
  - two chrominance measurements (I,Q or U,V).
- Reduce 8×8 chrominance to 4×4 by averaging 2x2 blocks.



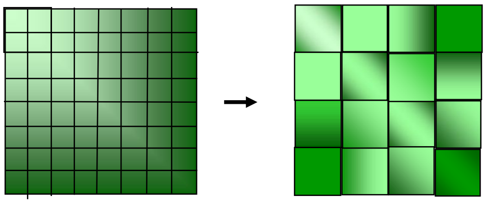
One way of reducing the bit-rate

- 8 x 8 red chrominance tiles reduced to 4 x 4



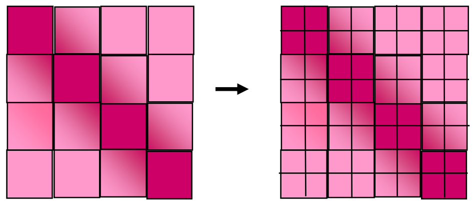
Also

- 8 x 8 green chrominance tiles reduced to 4 x 4

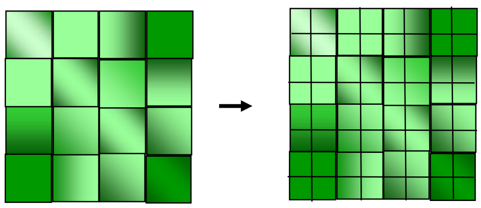


Red chrominance back to 8x8

- 8 x 8 red chrominance tiles reduced to 4 x 4

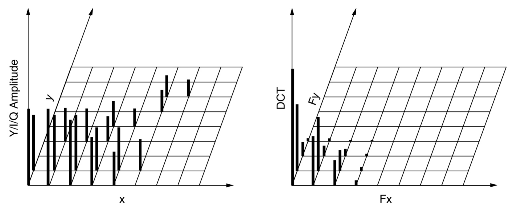


Green chrominance back to 8x8



JPEG compression: Step 2

- Use 2D Discrete Cosine Transform (DCT2)
- Apply it to 8×8 tiles of Y & tiles of I & Q
- Get 2 frequency axes: Fx & Fy
- High-frequency components usually small.



JPEG image compression: Step 3

- Quantize DCT coeffs using quantisation table below.
- Divide each coeff by table entry, then round to integer.
- Controls number of bits per coeff needed ∴ accuracy

DCT Coefficients

150	80	40	14	4	2	1	0
92	75	36	10	6	1	0	0
52	38	26	8	7	4	0	0
12	8	6	4	2	1	0	0
4	3	2	0	0	0	0	0
2	2	1	1	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Quantization table

1	1	2	4	8	16	32	64
1	1	2	4	8	16	32	64
2	2	2	4	8	16	32	64
4	4	4	4	8	16	32	64
8	8	8	8	8	16	32	64
16	16	16	16	16	16	32	64
32	32	32	32	32	32	32	64
64	64	64	64	64	64	64	64

Quantized coefficients

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Quantisation

- Encoder integer divides DCT coeffs by 2^n to 'lose' n bits.

- 21 ÷ 4 = 5 (rem 1 discarded)
- 10101 101
- At decoder:
- 5 x 4 = 20
- 10100
- 'Quantisation error' incurred

MANCHESTER 1824 — The University of Manchester

JPEG image compression: Step 4

- Bottom corner DCT component of each 8x8 tile has both frequencies zero.
- Sorry – it's the top corner in some graphs
- Represents average value of tile.
- Also known as the 'DC-DC component'
- Changes slowly from tile to tile.
- Differences often small but very noticeable
- Encode differences in DC-DC components between tiles
- Uniform luminance area then has all zero differences.

Lecture 4 COMP28512 31

MANCHESTER 1824 — The University of Manchester

JPEG image compression: Step 5

- 'zig-zag' scan to read out coeffs
- Count the number of successive zeros
- Here there are 38.
- Record '38' as 'run length code.

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Lecture 4 COMP28512 32

MANCHESTER 1824 — The University of Manchester

Step 5 (continued)

- Run length encoding
  - $(Z_0, N_0), (Z_1, N_1), (Z_2, N_2), (Z_3, N_3), \ldots$
    - Z_i: number of consecutive zeros
    - N_i: next non-zero number
- Example
  - 0 0 0 0 0 2 3 0 0 0 0 0 0 0 0 1
  - (5,2), (0,3), (8,1)

Lecture 4 COMP28512 33

MANCHESTER 1824 — The University of Manchester

JPEG image compression: Step 6

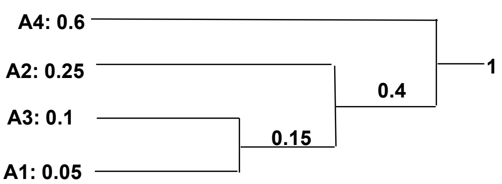
- Apply Huffman encoding to the quantised numbers.
- Assume there are just four: 95, 16, 13, 9
- We could just allocate an 8-bit integer to each.
- But we can do much better.
- Call these numbers A, B, C, D (or A1, A2, A3, A4)
- Idea is to use fewer bits for common numbers & more bits for less common numbers.
- A bit like 'Morse code', but codes are 'self terminating'.
- Always know when each Huffman code-word ends.
- Can use a default Huffman coding look-up table.
- Or we can generate our own 'image specific' table.
- Latter incurs overhead but may save bits overall.

Lecture 4 COMP28512 34

MANCHESTER 1824 — The University of Manchester

Huffman coding 1

- Variable length, self-terminating codes.
- Given 4 numbers: A1, A2, A3, A4 occurring with probabilities: 0.05, 0.25, 0.1, 0.6



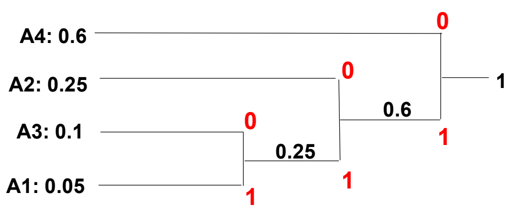
- Arrange in decreasing order of probabilities
- Then link two with lowest probability.
- Add probs & repeat. Sometimes ordering changes.

Lecture 4 COMP28512 35

MANCHESTER 1824 — The University of Manchester

Huffman coding 2

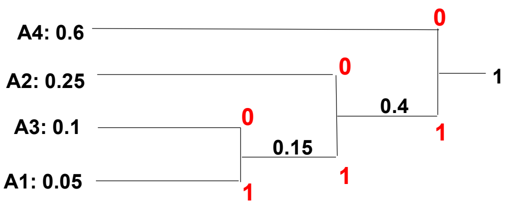
- Label corners 0 or 1 as shown below:



Lecture 4 COMP28512 36

Huffman coding 3

- Read backwards from end of tree to each of A1, A2, A3, A4



A4: 0   A2: 10   A3 110   A1: 111

Huffman coding result

A1 111
A2 10
A3 110
A4 0

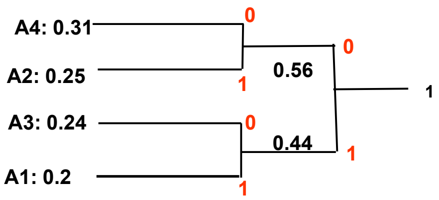
- Self terminating & more efficient than:

A1 00
A2 01
A3 10
A4: 11

for the given probabilities.

But more difficult to decode. See [wiki]

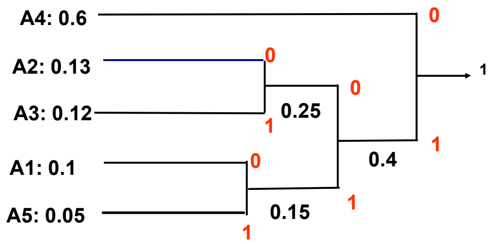
Another example



A1 = 11   A2=01   A3 = 10   A4 = 00

Ha! Ha! Not variable length for this example

Yet another example

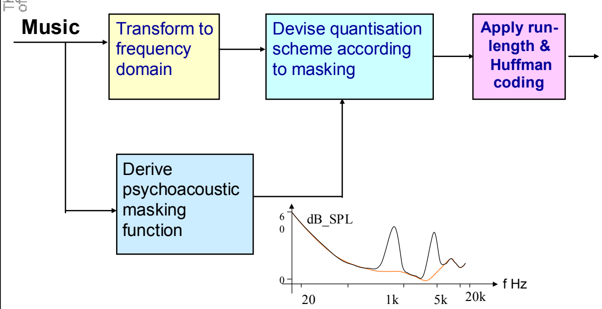


A1: 110; A2: 100; A3: 101; A4: 0; A5: 111

Disadvantage of Huffman coding

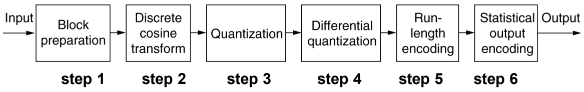
- Huffman coding is used for both image compression & mp3 music encoding.
- It is highly efficient & lossless
- Other aspects of mp3, JPEG & MPEG make them 'lossy'.
- Disadvantage of Huffman is its sensitivity to bit-errors.
- Due to its variable length & self terminating code-words.
- If one code-word has a bit-error it may be mis-interpreted as part of a longer or shorter code-word.
- On previous slide 010110100…encodes A4 A3 A3 A4,A4…
- If first 0 → 1, we get 110110100 which is A1 A1 A4 ,A2 …
- Many symbols after the bit-error may be affected.
- Whole mp3 or JPEG file may be unusable.

Reminder about mp3



MANCHESTER 1824
The University of Manchester

JPEG image compression: Summary



Lecture 4 COMP28512 43

MANCHESTER 1824
The University of Manchester

MATLAB demonstration

- Course web-site has the following progs:
  - Lecture4tiledimageEncoder.m
  - Lecture4tiledimageDecoder.m
- Also a selection of image files.
- Encoder implements steps 1-3 & stores the resulting data in a file: 'imageData.dat'.
  - '4x4' I-tiles are combined in fours to give 8x8 I-tiles.
  - Similarly for Q-tiles.
- Decoder reads the data & reconstructs the image.
- No run-length coding has been implemented yet.
- No Huffman coding yet.
- Other aspects are shown to work.

Lecture 4 COMP28512 44

MANCHESTER 1824
The University of Manchester

JPEG image compression

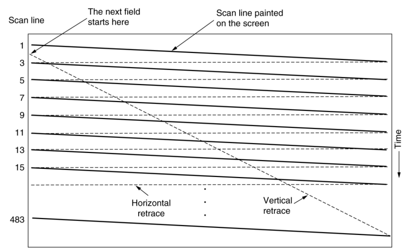
97KB, 698 x 658 | 18KB, 356 x 336 | 4KB, 160 x 151

Lecture 4 COMP28512 45

MANCHESTER 1824
The University of Manchester

Analog TVs: video scheme

- Scanning pattern used for NTSC video & TV
- (PAL is similar)



Lecture 4 COMP28512 46

MANCHESTER 1824
The University of Manchester

Frame-rate & interleaving

- Films capture 24 frames per second & display each frame for ≈ (1/24) s.
- TVs display 25 frames/s with each image scanned from top to bottom
- 25 Hz 'flicker' would be visible & annoying!
  - so 'interleave' the scan at 50 Hz
  - even & odd lines updated in alternate frames
- Computers display full image at 60+ frames/s
  - 'progressive scan'

Lecture 4 COMP28512 47

MANCHESTER 1824
The University of Manchester

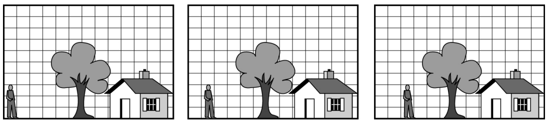
Digitally encoding moving pictures

- Encoding each frame as JPEG would be inefficient
- Would not exploit temporal redundancy due to similarity of each frame to those before & after.
- Could we send differences between complete frames?
  - OK for static scenes
  - Not so efficient where frames 'pan' from side to side or 'zoom'
- Best to use 'motion compensation'
  - Find similarity between **parts** of images in successive frames
  - send motion information where similarity is strong
  - then encode any remaining differences as JPEG
  - (when parts of images are similar, they are 'correlated')

Lecture 4 COMP28512 48

Motion compensation

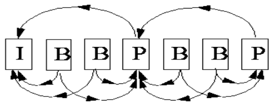
- Consider three consecutive frames
- Notice similarity of the figure walking towards the tree.
- Search for matching block in consecutive frames.
- Encode the movement & remaining differences.
- MPEG standard does not specify search algorithm



MPEG video compression

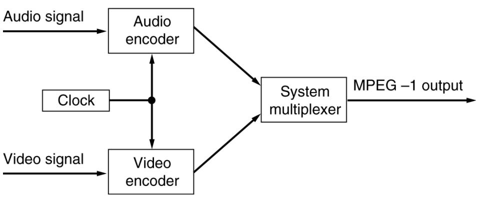
- Encode each frame as an 'I-frame', 'P-frame' or 'B-frame'.

- **I-frame** is an Image encoded as JPEG
- **P-frame** encodes positions of moving blocks predicted from Previous I & P frames, & remaining differences.
- **B-frame** encodes positions of moving blocks estimated from Both previous & next I & P frames, & remaining differences.



MPEG video with audio

Synchronization of the audio & video streams in MPEG-1.



Summary

- Uncompressed images use a lot of data
- Moving images (video) even more so
- Compression can save memory, & comms bandwidth
- Also saves download time & cost of storage media.
- JPEG compresses images by ~10x
- MPEG compresses video by ~100x
- In both cases, compression is 'lossy'. Not like 'zip'.
- Perceived loss of quality is small, but not zero.
- Built-in quality/compression trade-off
  - in choice of coefficient quantization matrix
  - other steps are largely lossless
- Sensitivity to bit-errors is greatly increased because of HC

Extra problem

- Symbols A,B,C,D E,F,G have probabilities:
  0.12, 0.13, 0.07, 0.07, 0.1, 0.36, 0.15
- Devise a Huffman code & consider how it would be decoded.