Compression: Images (JPEG)

What is JPEG?

- Works with colour and greyscale images
- Up to 24 bit colour images (Unlike GIF)
- Target **photographic** quality images (Unlike GIF)
- Suitable for many applications e.g., satellite, medical, general photography...
Basic JPEG Compression Pipeline

JPEG compression involves the following:

- Encoding

- Decoding – Reverse the order for encoding
Major Coding Algorithms in JPEG

The Major Steps in JPEG Coding involve:

- Colour Space Transform and subsampling (YIQ)
- DCT (Discrete Cosine Transformation)
- Quantisation
- Zigzag Scan
- DPCM on DC component
- RLE on AC Components
- Entropy Coding — Huffman or Arithmetic

We have met most of the algorithms already:

- JPEG exploits them in the compression pipeline to achieve maximal overall compression.
Quantisation

Why do we need to quantise:

- To throw out bits from DCT.
- *Example*: \((101101)_2 = 45\) (6 bits).

  Truncate to 4 bits: \((1011)_2 = 11\).

  Truncate to 3 bits: \((101)_2 = 5\).

- Quantisation error is the main source of **Lossy Compression**.
- **DCT itself is not Lossy**
- How we **throw away bits** in **Quantisation Step** is **Lossy**
Quantisation Methods

Uniform quantisation

- Divide by constant $N$ and round result ($N = 4$ or $8$ in examples on previous page).
- Non powers-of-two gives fine control (e.g., $N = 6$ loses 2.5 bits)
Quantisation Tables

- In JPEG, each $F[u,v]$ is divided by a constant $q(u,v)$.
- Table of $q(u,v)$ is called quantisation table.
- Eye is most sensitive to low frequencies (upper left corner), less sensitive to high frequencies (lower right corner).
- JPEG Standard defines 2 default quantisation tables, one for luminance (below), one for chrominance. \textit{E.g Table below}

\begin{tabular}{cccccccccccc}
16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\
12 & 12 & 14 & 19 & 26 & 58 & 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 \\
\end{tabular}
Quantization Tables (Cont)

• Q: How would changing the numbers affect the picture

E.g., if we doubled them all?

**Quality factor** in most implementations is the **scaling factor** for default quantization tables.

• **Custom quantization tables** can be put in image/scan header.

JPEG Quantisation Example

• **JPEG Quantisation Example (Java Applet)**
Zig-zag Scan

What is the purpose of the Zig-zag Scan:

• To group low frequency coefficients in top of vector.
• Maps 8 x 8 to a 1 x 64 vector
Differential Pulse Code Modulation (DPCM) on DC Component

- Another encoding method is employed
- DPCM on the DC component.
- Why is this strategy adopted:
  - DC component is large and varies, but often close to previous value (like lossless JPEG).
  - Encode the difference from previous 8x8 blocks – DPCM
Run Length Encode (RLE) on AC Components

Yet another simple compression technique is applied to the AC component:

• 1x63 vector (AC) has lots of zeros in it

• Encode as \((\text{skip}, \text{value})\) pairs, where \(\text{skip}\) is the number of zeros and \(\text{value}\) is the next non-zero component.

• Send \((0,0)\) as end-of-block sentinel value.
Huffman (Entropy) Coding

DC and AC components finally need to be represented by a smaller number of bits (Arithmetic coding also supported in place of Huffman coding):

• (Variant of) Huffman coding: Each DPCM-coded DC coefficient is represented by a pair of symbols: 
  \((\text{Size}, \text{Amplitude})\)
  where \text{Size} indicates number of bits needed to represent coefficient and \text{Amplitude} contains actual bits.

• \textbf{Size only} Huffman coded in JPEG:
  
  – \text{Size} does not change too much, generally smaller \text{Sizes} occur frequently (= \textit{low entropy} so is suitable for coding,
  
  – \text{Amplitude} can change widely so coding \textbf{no real benefit}
Huffman (Entropy) Coding (Cont)

- **Example Size category for possible Amplitudes:**

<table>
<thead>
<tr>
<th>Size</th>
<th>Typical Huffman Code for Size</th>
<th>Amplitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>-1, 1</td>
</tr>
<tr>
<td>2</td>
<td>011</td>
<td>-3, -2, 2, 3</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>-7.. -4, 4..7</td>
</tr>
<tr>
<td>4</td>
<td>101</td>
<td>-15.. -8, 8..15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Use *ones complement* scheme for negative values: *i.e* 10 is binary for 2 and 01 for -2 (bitwise inverse). Similarly, 00 for -3 and 11 for 3.
Huffman Coding DC Example

- **Example**: if DC values are 150, -6, 5, 3, -8
- Then 8, 3, 3, 2 and 4 bits are needed respectively. Send off Sizes as Huffman symbol, followed by actual values in bits.

\[(8_{huff}, 10010110), (3_{huff}, 001), (3_{huff}, 101), (2_{huff}, 11), (4_{huff}, 0111)\]

where \(8_{huff} \ldots\) are the Huffman codes for respective numbers.
- Huffman Tables can be custom (sent in header) or default.
Huffman Coding on AC Component

AC coefficient are run-length encoded (RLE)

- RLE pairs (Runlength, Value) are Huffman coded as with DC only on Value.
- So we get a triple: (Runlength, Size, Amplitude)
- However, Runlength, Size allocated 4-bits each and put into a single byte with is then Huffman coded. Again, Amplitude is not coded.
- So only two symbols transmitted per RLE coefficient:

\[(\text{RLESIZEbyte}_{huff}, \text{Amplitude})\]
Example JPEG Compression
Another Enumerated Example

<table>
<thead>
<tr>
<th></th>
<th>130</th>
<th>144</th>
<th>149</th>
<th>153</th>
<th>155</th>
<th>155</th>
<th>155</th>
<th>155</th>
<th>155</th>
<th>155</th>
</tr>
</thead>
<tbody>
<tr>
<td>144</td>
<td>151</td>
<td>153</td>
<td>156</td>
<td>159</td>
<td>156</td>
<td>156</td>
<td>156</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>155</td>
<td>160</td>
<td>163</td>
<td>158</td>
<td>156</td>
<td>156</td>
<td>156</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>159</td>
<td>161</td>
<td>162</td>
<td>160</td>
<td>160</td>
<td>159</td>
<td>159</td>
<td>159</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>159</td>
<td>160</td>
<td>161</td>
<td>162</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>157</td>
<td>157</td>
<td>157</td>
<td>157</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>162</td>
<td>162</td>
<td>161</td>
<td>163</td>
<td>157</td>
<td>157</td>
<td>157</td>
<td>157</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>162</td>
<td>162</td>
<td>161</td>
<td>161</td>
<td>158</td>
<td>158</td>
<td>158</td>
<td>158</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) source image samples

<table>
<thead>
<tr>
<th></th>
<th>-2</th>
<th>-1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>-10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>-12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>-13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) forward DCT coefficients

(c) quantization table

<table>
<thead>
<tr>
<th>162</th>
<th>162</th>
<th>162</th>
<th>161</th>
<th>161</th>
<th>161</th>
<th>161</th>
<th>161</th>
<th>161</th>
<th>161</th>
</tr>
</thead>
<tbody>
<tr>
<td>144</td>
<td>146</td>
<td>149</td>
<td>152</td>
<td>154</td>
<td>156</td>
<td>156</td>
<td>156</td>
<td>156</td>
<td>156</td>
</tr>
<tr>
<td>148</td>
<td>150</td>
<td>152</td>
<td>154</td>
<td>156</td>
<td>156</td>
<td>156</td>
<td>156</td>
<td>156</td>
<td>156</td>
</tr>
<tr>
<td>155</td>
<td>156</td>
<td>157</td>
<td>158</td>
<td>158</td>
<td>157</td>
<td>156</td>
<td>156</td>
<td>156</td>
<td>156</td>
</tr>
<tr>
<td>160</td>
<td>161</td>
<td>161</td>
<td>162</td>
<td>161</td>
<td>159</td>
<td>157</td>
<td>155</td>
<td>155</td>
<td>155</td>
</tr>
<tr>
<td>163</td>
<td>163</td>
<td>164</td>
<td>163</td>
<td>162</td>
<td>160</td>
<td>158</td>
<td>156</td>
<td>156</td>
<td>156</td>
</tr>
<tr>
<td>163</td>
<td>164</td>
<td>164</td>
<td>162</td>
<td>162</td>
<td>158</td>
<td>157</td>
<td>157</td>
<td>157</td>
<td>157</td>
</tr>
<tr>
<td>160</td>
<td>161</td>
<td>162</td>
<td>162</td>
<td>161</td>
<td>159</td>
<td>158</td>
<td>158</td>
<td>158</td>
<td>158</td>
</tr>
<tr>
<td>158</td>
<td>159</td>
<td>161</td>
<td>161</td>
<td>162</td>
<td>161</td>
<td>159</td>
<td>158</td>
<td>158</td>
<td>158</td>
</tr>
</tbody>
</table>

(d) normalized quantized coefficients

(e) denormalized quantized coefficients

(f) reconstructed image samples
The JPEG algorithm may be summarised as follows:

**im2jpeg.m** (Encoder) **jpeg2im.m** (Decoder)

**mat2huff.m** (Huffman coder)

```matlab
m = [16 11 10 16 24 40 51 61] * quality; % JPEG normalizing array
    [12 12 14 19 26 58 60 55] % and zig-zag reordering
    [14 13 16 24 40 57 69 56] % pattern.
    [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];

order = [1 9 2 3 10 17 25 18 11 4 5 12 19 26 33 ... 41 34 27 20 13 6 7 14 21 28 35 42 49 57 50 ... 43 36 29 22 15 8 16 23 30 37 44 51 58 59 52 ... 45 38 31 24 32 39 46 53 60 61 54 47 40 48 55 ... 62 63 56 64];

[xm, xn] = size(x); % Get input size.
x = double(x) - 128; % Level shift input
t = dctmtx(8); % Compute 8 x 8 DCT matrix

% Compute DCTs of 8x8 blocks and quantize the coefficients.
y = blkproc(x, [8 8], 'P1 * x * P2', t, t');
y = blkproc(y, [8 8], 'round(x ./ P1)', m);
```
y = im2col(y, [8 8], 'distinct'); % Break 8x8 blocks into columns
xb = size(y, 2); % Get number of blocks
y = y(order, :); % Reorder column elements

eob = max(y(:)) + 1; % Create end-of-block symbol
r = zeros(numel(y) + size(y, 2), 1);
count = 0;
for j = 1:xb % Process 1 block (col) at a time
    i = max(find(y(:, j))); % Find last non-zero element
    if isempty(i) % No nonzero block values
        i = 0;
    end
    p = count + 1;
    q = p + i;
    r(p:q) = [y(1:i, j); eob]; % Truncate trailing 0’s, add EOB,
    count = count + i + 1; % and add to output vector
end
r((count + 1):end) = []; % Delete unused portion of r

y = struct;
y.size = uint16([xm xn]);
y.numblocks = uint16(xb);
y.quality = uint16(quality * 100);
y.huffman = mat2huff(r);
Further Information

Further standards:

- **Lossless JPEG**: Predictive approach for lossless compression (why?), not widely used

<table>
<thead>
<tr>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
</tr>
</tbody>
</table>

- **JPEG 2000**: ISO/IEC 15444
  - Based on wavelet transform, instead of DCT, no $8 \times 8$ blocks, less artefacts
  - Often better compression ratio, compared with JPEG

![Diagram of Lossless Encoder Process]
Further Information

References:

• [http://www.jpeg.org](http://www.jpeg.org)
• [Online JPEG Tutorial](http://www.jpeg.org)
• [The JPEG Still Picture Compression Standard](http://www.jpeg.org)
• [The JPEG 2000 Still Image Compression Standard](http://www.jpeg.org)