Transform Coefficient Coding

<table>
<thead>
<tr>
<th>7</th>
<th>3</th>
<th>1</th>
<th>0</th>
<th>-3</th>
<th>-1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</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>
</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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

\[(0, 7)\]  \[(0, 3)\]  \[(0, 1)\]  \[(2, 1)\]  \[(1, -1)\]  \[(6, -3)\]  \[(0, -1)\]  \[(0, -2)\]  \[(0, -1)\]  \[(EOB)\]

100111  0111  001  111001  11000  11111111011000  000  0101  000  1010
Entropy Coding

Lossless Coding / Entropy Coding
- Maps sequence of symbols into sequence of bits
- Reversible Mapping: Original sequence of symbols can be reconstructed

Unique Decodability
- Each sequence of bits can only be generated by one sequence of symbols
- Most important class: Prefix codes

Simplest Variant: Scalar Lossless Codes
- Table that assigns a codeword to each symbol

Examples

<table>
<thead>
<tr>
<th>symbol</th>
<th>codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>01</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
</tr>
<tr>
<td>D</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>symbol</th>
<th>codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>01</td>
</tr>
<tr>
<td>C</td>
<td>001</td>
</tr>
<tr>
<td>D</td>
<td>000</td>
</tr>
</tbody>
</table>
Prefix Codes

- No codeword represents a prefix of any other codeword
- Prefix codes can be represented as binary trees
- There are no better uniquely decodable codes than best prefix code

⇒ **Uniquely** and **instantaneously** decodable codes

<table>
<thead>
<tr>
<th>letter</th>
<th>codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>00</td>
</tr>
<tr>
<td>b</td>
<td>010</td>
</tr>
<tr>
<td>c</td>
<td>011</td>
</tr>
<tr>
<td>d</td>
<td>10</td>
</tr>
<tr>
<td>e</td>
<td>1100</td>
</tr>
<tr>
<td>f</td>
<td>1101</td>
</tr>
<tr>
<td>g</td>
<td>111</td>
</tr>
</tbody>
</table>

Prefix codes can be represented as binary trees.
Coding Efficiency of Scalar Codes

**Design Goal:** Minimize number of required bits

- Minimize average codeword length $\bar{\ell}$ while retaining unique decodability

$$\bar{\ell} = \sum_k p_k \ell_k$$

- Assign shorter codewords to more probable symbols

**Entropy & Redundancy**

- **Entropy $H$:** Lower bound on average codeword length

$$\bar{\ell} \geq H = H(p) = - \sum_k p_k \log_2 p_k \quad \text{(equality iff $\ell_k = - \log_2 p_k$)}$$

- **Redundancy**

$$\varrho = \bar{\ell} - H = \sum_k p_k (\ell_k + \log_2 p_k) \geq 0$$

- **Relative redundancy**

$$\varrho' = \frac{\varrho}{H} = \frac{\bar{\ell} - H}{H} = \frac{\bar{\ell}}{H} - 1$$
Optimal Prefix Codes

Prefix Codes with Zero Redundancy

- Necessary condition

\[ \ell_k = - \log_2 p_k \quad \iff \quad p_k = 2^{-\ell_k} \]

⇒ Only possible if all probability masses represent negative integer powers of 2

Huffman Algorithm

- Algorithm for constructing optimal prefix codes (with minimum redundancy)

1. Construct binary code tree (for given pmf)
   - Select the two least likely symbols and create a parent node
   - Consider the combination of the two symbols as new symbol
   - Continue the algorithm until all symbols are merged into the root node

2. Construct code by labeling the branches with “0” and “1”
Example: Construction of a Huffman Code

Given:
alphabet $A$ with pmf $\{p_k\}_k$

$$
a_k \quad p_k \quad b_k
\begin{array}{ccc}
a & 0.16 & 111 \\
b & 0.04 & 0001 \\
c & 0.04 & 0000 \\
d & 0.16 & 110 \\
e & 0.23 & 01 \\
f & 0.07 & 1001 \\
g & 0.06 & 1000 \\
h & 0.09 & 001 \\
i & 0.15 & 101
\end{array}$$

First step:
assign symbols and probabilities to terminal nodes

$$
\begin{array}{ccc}
a & 16 \\
b & 4 \\
c & 16 \\
d & 23 \\
e & 7 \\
f & 6 \\
g & 9 \\
h & 8 \\
i & 13
\end{array}
$$

Next step:
re-order for better readability

$$
\begin{array}{ccc}
a & 16 \\
b & 4 \\
c & 16 \\
d & 23 \\
e & 13 \\
f & 6 \\
g & 9 \\
h & 8 \\
i & 13
\end{array}
$$

Next step:
re-order for better readability

$$
\begin{array}{ccc}
a & 16 \\
b & 4 \\
c & 16 \\
d & 23 \\
e & 13 \\
f & 6 \\
g & 17 \\
h & 8 \\
i & 13
\end{array}
$$

Next step:
re-order for better readability

$$
\begin{array}{ccc}
a & 16 \\
b & 4 \\
c & 16 \\
d & 23 \\
e & 32 \\
f & 6 \\
g & 17 \\
h & 8 \\
i & 13
\end{array}
$$

Next step:
re-order for better readability

$$
\begin{array}{ccc}
a & 16 \\
b & 4 \\
c & 16 \\
d & 28 \\
e & 32 \\
f & 6 \\
g & 17 \\
h & 8 \\
i & 13
\end{array}
$$

Next step:
label branches with 0 and 1

\[ \bar{\ell} = 2.98 \]
\[ H(p) \approx 2.9405 \]
\[ \varrho \approx 0.0395 \text{ (1.34\%)} \]

\[ \sum \ell_{\text{node}} = 2 \]

\[ H(p) \approx 2.9405 \]

\[ \varrho \approx 0.0395 \text{ (1.34\%)} \]

\[ \ell_{\text{root}} = 2 \]

\[ \sum p_k \ell_k = 2 \]

Note: nodes are labeled with $100 \ p_k$
Inefficiency of Scalar Huffman Codes

Pmfs with a probability mass $p_k$ significantly greater than 0.5

- Example: Binary pmf (0.9, 0.1)
  - Huffman code: Codewords “1” and “0”
  - Average codeword length of Huffman code: $\bar{\ell} = 1$
  - Entropy: $H = -0.9 \log_2 0.9 - 0.1 \log_2 0.1 \approx 0.469$
  - Relative redundancy: $\rho' \approx 1/0.469 - 1 \approx 113.2\%$

- Problem: Cannot use codeword lengths with less than one bit

Random Sources with Memory (dependencies between symbols)

- Symbol probabilities depend on previous symbol / previous symbols
- Cannot exploit dependencies with scalar code

Instationary Sources (statistical properties change over time)

- Optimal code changes over time
- Rather complicated to construct new code at regular intervals (need to be done simultaneously at encoder and decoder)
Conditional Huffman Codes

- Switch codeword table depending on previous symbol(s)
- Example: Stationary Markov process

<table>
<thead>
<tr>
<th>$S_{n-1} = a_0$</th>
<th>$S_{n-1} = a_1$</th>
<th>$S_{n-1} = a_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_k$</td>
<td>$p_k$ code</td>
<td>$p_k$ code</td>
</tr>
<tr>
<td>$a_0$</td>
<td>0.90 1</td>
<td>0.15 01</td>
</tr>
<tr>
<td>$a_1$</td>
<td>0.05 01</td>
<td>0.80 1</td>
</tr>
<tr>
<td>$a_2$</td>
<td>0.05 00</td>
<td>0.05 00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$H_0 = 0.5690$</th>
<th>$H_1 = 0.8842$</th>
<th>$H_2 = 1.3527$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bar{\ell}_0 = 1.1$</td>
<td>$\bar{\ell}_1 = 1.2$</td>
<td>$\bar{\ell}_2 = 1.4$</td>
</tr>
</tbody>
</table>

$H(S_n \mid S_{n-1}) = 0.7331$
$\bar{\ell}_\text{cond} = 1.1578$

- Bound on average codeword length: Conditional Entropy

$$H(S_n \mid S_{n-1}) = \sum_i p(a_i) H(S_n \mid a_i) = -\sum_{i,k} p(a_k, a_i) \log_2 p(a_k \mid a_i)$$
Block Huffman Codes

- Assign codewords to a fixed number $N$ of consecutive symbols
- Example: Binary IID process with $p_0 = 0.9$ and $p_1 = 0.1$
  - Scalar code: $\bar{\ell} = 1$, redundancy $\varrho' = 113.2\%$ (entropy $H \approx 0.469$)

<table>
<thead>
<tr>
<th>$s_0s_1$</th>
<th>$p(s_0, s_1)$</th>
<th>codewords</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0.81</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0.09</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>0.09</td>
<td>000</td>
</tr>
<tr>
<td>11</td>
<td>0.01</td>
<td>001</td>
</tr>
</tbody>
</table>

$N = 2$

$\bar{\ell}_2 = 1.29$

$\bar{\ell} = \bar{\ell}_2/2 = 0.645$

$\varrho' = 37.5\%$

$N = 3 \implies \bar{\ell} = 0.65$ ($\varrho' = 13.6\%$)

<table>
<thead>
<tr>
<th>$s_0s_1s_2$</th>
<th>$p(s_0, s_1, s_2)$</th>
<th>codewords</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0.729</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>0.081</td>
<td>100</td>
</tr>
<tr>
<td>010</td>
<td>0.081</td>
<td>101</td>
</tr>
<tr>
<td>011</td>
<td>0.009</td>
<td>11100</td>
</tr>
<tr>
<td>100</td>
<td>0.081</td>
<td>110</td>
</tr>
<tr>
<td>101</td>
<td>0.009</td>
<td>11101</td>
</tr>
<tr>
<td>110</td>
<td>0.009</td>
<td>11110</td>
</tr>
<tr>
<td>111</td>
<td>0.001</td>
<td>11111</td>
</tr>
</tbody>
</table>

- Also suitable for taking into account conditional probabilities (i.e., dependencies between symbols)
- Codeword tables exponentially increase with $N$
Block Entropy & Entropy Rate

- **Block Entropy**

\[ H_N = - \sum_{a_0, a_1, \cdots, a_{N-1}} p(a_0, a_1, \cdots, a_{N-1}) \log_2 p(a_0, a_1, \cdots, a_{N-1}) \]

- Lower bound for average codeword length (for block codes of \( N \) symbols)

\[ \bar{\ell} \geq \frac{H_N}{N} \]

- **Entropy Rate**

\[ \bar{H} = \lim_{N \to \infty} \frac{H_N}{N} \]

- **Fundamental Lossless Coding Theorem**

\[ \bar{\ell} \geq \bar{H} = \lim_{N \to \infty} \frac{H_N}{N} \]
V2V Codes

Generalization of Block Codes

- Assign codewords to symbol sequences of variable length
- All messages must be representable by selected symbol sequences
- Desirable: Redundancy-free set of symbol sequences

⇒ Select symbol sequences that represent full M-ary tree

Examples

<table>
<thead>
<tr>
<th>$\mathcal{A} = {a, b}$</th>
<th>$\mathcal{A} = {a, b}$</th>
<th>$\mathcal{A} = {x, y, z}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaaa</td>
<td>aaaaab</td>
<td>xxx</td>
</tr>
<tr>
<td>aaa 0</td>
<td>aaaab 10</td>
<td>xxy 100</td>
</tr>
<tr>
<td>aab 100</td>
<td>aab 110</td>
<td>xxz 101</td>
</tr>
<tr>
<td>ab 101</td>
<td>ab 1110</td>
<td>xy 1100</td>
</tr>
<tr>
<td>ba 110</td>
<td>ab 11110</td>
<td>xz 1101</td>
</tr>
<tr>
<td>bb 111</td>
<td>b 1111</td>
<td>y 1110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>z 1111</td>
</tr>
</tbody>
</table>
### V2V Codes as Double Tree

#### IID Source

<table>
<thead>
<tr>
<th>symbol</th>
<th>probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.80</td>
</tr>
<tr>
<td>b</td>
<td>0.15</td>
</tr>
<tr>
<td>c</td>
<td>0.05</td>
</tr>
</tbody>
</table>

**Entropy Rate**

- **Entropy Rate**: \( \bar{H}(S) = 0.88418 \) (marginal entropy)
- **Scalar Huffman**: \( \bar{\ell} = 1.2 \) (3 codewords)
- **2-Symbol Blocks**: \( \bar{\ell} = 0.93375 \) (9 codewords)
- **V2V Code**: \( \bar{\ell} = 0.88934 \) (7 codewords)

**Redundancy**

- **Redundancy**: \( \varrho = 0.00516 \) (0.58%)
## Typical Properties of Transform Coefficient Levels

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>3</th>
<th>1</th>
<th>0</th>
<th>-3</th>
<th>-1</th>
<th>0</th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></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></td>
<td></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></td>
<td></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></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>-4</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>0</th>
<th>-1</th>
<th>0</th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></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></td>
<td></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></td>
<td></td>
</tr>
</tbody>
</table>

### Typical Properties at moderate/reasonable Quantization

- Most quantization indexes are equal to zero ($p_0 \gg 0.5$)
- Probability of zero is higher for high-frequency components
- Non-zero levels are concentrated at top-left corner

➤ These properties should be exploited in entropy coding
Scalar Variable Length Codes?

Design Huffman Code
- Measure statistics for large number of pictures
- Develop Huffman code for estimated pmf

Alternative: Universal Code
- Use universal code that works well for estimated pmf
  - Unary code (for absolute values)
  - Exponential Golomb code
  - Golomb-Rice codes

Coding Efficiency?
- Require at least one bit per sample
- Average codeword length: $\bar{\ell} \geq 1 \gg \bar{H}$
- Maximum compression ratio 8:1 (for 8 bit)
- Unsuitable for transform coefficient levels in image and video coding

<table>
<thead>
<tr>
<th>$q$</th>
<th>codewords</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10 0</td>
</tr>
<tr>
<td>-1</td>
<td>10 1</td>
</tr>
<tr>
<td>2</td>
<td>110 00</td>
</tr>
<tr>
<td>-2</td>
<td>110 01</td>
</tr>
<tr>
<td>3</td>
<td>110 10</td>
</tr>
<tr>
<td>-3</td>
<td>110 11</td>
</tr>
<tr>
<td>4</td>
<td>1110 000</td>
</tr>
<tr>
<td>-4</td>
<td>1110 001</td>
</tr>
<tr>
<td>5</td>
<td>1110 010</td>
</tr>
<tr>
<td>-5</td>
<td>1110 011</td>
</tr>
<tr>
<td>6</td>
<td>1110 100</td>
</tr>
<tr>
<td>-6</td>
<td>1110 101</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Conditional or Block Codes?

Conditional Scalar Variable Length Codes
- Same problem as for simple scalar codes
- Need at least one bit per transform coefficient

Block Codes
- Block codes for blocks of transform coefficients seems reasonable
- Should provide very good coding efficiency

But:
- $N \times M$ blocks with $K$ quantization values yields codeword tables of size $K^{N \cdot M}$
- $8 \times 8$ blocks with 255 values: More than $10^{154}$ codewords
- $8 \times 8$ blocks with 2 values: $2^{64}$ codewords (address space of 64-bit arch.)

Possible usage:
- Very small block sizes (2 or 3 coefficients)
- Signal position of non-zero coefficients for small block sizes (e.g., 8) and additional transmit non-zero levels
V2V Codes?

How to select symbol sequences?

- Our problem are the zeros (large probability!)
- Could use following set of symbol sequences:

  \[ \begin{align*}
  &X \\
  &0X \\
  &00X \\
  &000X \\
  &0000X \\
  &\vdots \\
  &00000 \cdots 0
  \end{align*} \]

  where X is a place holder for quantization indexes unequal to 0

- \( N \times M \) block and \( K \) potential values: \( N \cdot M \cdot K + 1 \) codewords
- Example: \( 8 \times 8 \) block and 255 values: 16321 codewords
- Large, but feasible codeword tables (typical: offline design)
Run-Level Coding

Run-Level Coding = V2V Code with Structural Constraint

- Represent symbol sequences as \((\text{run}, \text{level})\) pairs

\[
\begin{align*}
X & \quad \rightarrow \quad \text{run}=0, \quad \text{level}=X \\
0X & \quad \rightarrow \quad \text{run}=1, \quad \text{level}=X \\
00X & \quad \rightarrow \quad \text{run}=2, \quad \text{level}=X \\
000X & \quad \rightarrow \quad \text{run}=3, \quad \text{level}=X \\
0000X & \quad \rightarrow \quad \text{run}=4, \quad \text{level}=X \\
& \quad \vdots \\
00000\cdots 0 & \quad \rightarrow \quad \text{end of block (eob)} \quad [\text{run = max. possible value}]
\end{align*}
\]

- Codewords are assigned to \((\text{run}, \text{level})\) pairs
  - \text{run}: Number of levels equal to zero preceding next non-zero level
  - \text{level}: Value of next non-zero level
  - \text{eob}: Special symbol: All remaining levels are equal to zero
Scanning of Transform Coefficient Levels

Coding Order for Run-Level Coding

- Most effective (run,level) pair is the \textit{end-of-block symbol} (eob)
- Should maximize number of zeros that are represented by eob symbol
- Define suitable coding order from low- to high-frequency components

Examples used in Practice

<table>
<thead>
<tr>
<th>zig-zag scan (JPEG)</th>
<th>diagonal scan (HEVC)</th>
</tr>
</thead>
</table>

T. Wiegand (TU Berlin) — Image and Video Coding: Transform Coefficient Coding
**Example: Run-Level Coding with Zig-Zag Scan**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-3</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>2</td>
<td>-1</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>
</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>
</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>
</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>
</tr>
<tr>
<td></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>

**excerpt of codeword table**

<table>
<thead>
<tr>
<th>(run,level)</th>
<th>codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, ±1)</td>
<td>11s</td>
</tr>
<tr>
<td>(0, ±2)</td>
<td>0100 s</td>
</tr>
<tr>
<td>(0, ±3)</td>
<td>0010 1s</td>
</tr>
<tr>
<td>(0, ±4)</td>
<td>0000 110s</td>
</tr>
<tr>
<td>(1, ±1)</td>
<td>011s</td>
</tr>
<tr>
<td>(1, ±2)</td>
<td>0001 10s</td>
</tr>
<tr>
<td>(1, ±3)</td>
<td>0010 0101 s</td>
</tr>
<tr>
<td>(2, ±1)</td>
<td>0101 s</td>
</tr>
<tr>
<td>(eob)</td>
<td>10</td>
</tr>
</tbody>
</table>

1. Scanning and conversion into (run,level) pairs
   - (0, -4)  (1, -3)  (0, 2)  (1, 2)  (2, -1)  (eob)

2. Conversion into bitstream (via table look-up)
   - 00001101  00100101  10100000  01100010  1110  (36 bits)
Run-Level Coding in JPEG

**Split level value into ...**
- Category \( C \)
- Value inside category \( C \)

**DC Coefficient**
- Separate coding of DC coefficient
- Variable-length code for category
- Fixed-length code for actual value
  (length depends on category)

<table>
<thead>
<tr>
<th>C</th>
<th>absolute levels</th>
<th>FLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1 bit</td>
</tr>
<tr>
<td>2</td>
<td>2 \ldots 3</td>
<td>2 bits</td>
</tr>
<tr>
<td>3</td>
<td>4 \ldots 7</td>
<td>3 bits</td>
</tr>
<tr>
<td>4</td>
<td>8 \ldots 15</td>
<td>4 bits</td>
</tr>
<tr>
<td>5</td>
<td>16 \ldots 31</td>
<td>5 bits</td>
</tr>
<tr>
<td>6</td>
<td>32 \ldots 63</td>
<td>6 bits</td>
</tr>
<tr>
<td>7</td>
<td>64 \ldots 127</td>
<td>7 bits</td>
</tr>
<tr>
<td>8</td>
<td>128 \ldots 255</td>
<td>8 bits</td>
</tr>
<tr>
<td>9</td>
<td>256 \ldots 511</td>
<td>9 bits</td>
</tr>
<tr>
<td>10</td>
<td>512 \ldots 1023</td>
<td>10 bits</td>
</tr>
</tbody>
</table>

**AC Coefficient**
- Zig-zag scan of AC coefficients (8×8 blocks)
- Variable-length code for (run,category) pairs
  - Maximum run is 16: There is a (16,0) pair
  - \( 16 \cdot 10 \pm 2 = 162 \) codewords (table transmitted in bitstream)
- Fixed-length code for actual value (length depends on category)
### JPEG: Example Run-Category Table (Annex K)

<table>
<thead>
<tr>
<th>run/category</th>
<th>codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOB</td>
<td>1010</td>
</tr>
<tr>
<td>0/1</td>
<td>00</td>
</tr>
<tr>
<td>0/2</td>
<td>01</td>
</tr>
<tr>
<td>0/3</td>
<td>100</td>
</tr>
<tr>
<td>0/4</td>
<td>1011</td>
</tr>
<tr>
<td>0/5</td>
<td>11010</td>
</tr>
<tr>
<td>0/6</td>
<td>111000</td>
</tr>
<tr>
<td>0/7</td>
<td>1111000</td>
</tr>
<tr>
<td>0/8</td>
<td>111110110</td>
</tr>
<tr>
<td>0/9</td>
<td>111111110000010</td>
</tr>
<tr>
<td>0/10</td>
<td>1111111110000011</td>
</tr>
<tr>
<td>1/1</td>
<td>1100</td>
</tr>
<tr>
<td>1/2</td>
<td>11011</td>
</tr>
<tr>
<td>1/3</td>
<td>1111001</td>
</tr>
<tr>
<td>1/4</td>
<td>111110110</td>
</tr>
<tr>
<td>1/5</td>
<td>11111110110</td>
</tr>
<tr>
<td>1/6</td>
<td>111111111000010</td>
</tr>
<tr>
<td>1/7</td>
<td>1111111110000101</td>
</tr>
<tr>
<td>1/8</td>
<td>1111111110000110</td>
</tr>
<tr>
<td>1/9</td>
<td>1111111110000111</td>
</tr>
<tr>
<td>1/10</td>
<td>1111111111000100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>run/category</th>
<th>codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/1</td>
<td>11100</td>
</tr>
<tr>
<td>2/2</td>
<td>1111001</td>
</tr>
<tr>
<td>2/3</td>
<td>111110111</td>
</tr>
<tr>
<td>2/4</td>
<td>111111110100</td>
</tr>
<tr>
<td>2/5</td>
<td>1111111110001001</td>
</tr>
<tr>
<td>2/6</td>
<td>1111111110001010</td>
</tr>
<tr>
<td>2/7</td>
<td>1111111110001011</td>
</tr>
<tr>
<td>2/8</td>
<td>1111111110001100</td>
</tr>
<tr>
<td>2/9</td>
<td>1111111110001101</td>
</tr>
<tr>
<td>2/10</td>
<td>1111111110001110</td>
</tr>
<tr>
<td>3/1</td>
<td>111010</td>
</tr>
<tr>
<td>3/2</td>
<td>11110111</td>
</tr>
<tr>
<td>3/3</td>
<td>111111110101</td>
</tr>
<tr>
<td>3/4</td>
<td>1111111110001111</td>
</tr>
<tr>
<td>3/5</td>
<td>1111111110010000</td>
</tr>
<tr>
<td>3/6</td>
<td>1111111110010001</td>
</tr>
<tr>
<td>3/7</td>
<td>1111111110010010</td>
</tr>
<tr>
<td>3/8</td>
<td>1111111110010011</td>
</tr>
<tr>
<td>3/9</td>
<td>1111111110010100</td>
</tr>
<tr>
<td>3/10</td>
<td>1111111110010101</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Run-Level Coding in H.262 | MPEG-2 Video

Scanning of Transform Coefficient Levels
- Zig-zag scanning of $8 \times 8$ transform blocks

Run-Level code for likely (run,level) pairs
- Codeword table with 114 codewords
- Includes end-of-block-symbol (EOB)
- Includes escape symbol for less likely (run,level) pairs
- If escape symbol is coded, then
  - actual run is coded using 6 bits
  - actual level is coded using 12 bits

Can we further improve the entropy coding of transform coefficients?
- Observation: Last non-zero level has typically small value
- Can we exploit that knowledge?
Run-Level-Last Coding (H.263 & MPEG-4 Visual)

Use 3D events \((run, level, last)\)
- **run**: Number of levels equal to zero preceding next non-zero level
- **level**: Value of next non-zero level
- **last**: Whether next non-zero level is last non-zero level in block

Entropy Coding Table
- Variable length code for most likely events \((run, level, last)\)
- 103 codewords (when not counting sign bits)
- Includes escape code, followed by 6 bits \((run)\), 8 bits \((level)\), 1 bit \((last)\)

Coded Block Pattern
- **Note**: Code cannot represent a block with all levels equal to zero
- **Need**: To signal a flag whether a block has any non-zero coefficients

**Coded block pattern**:
- Block code for four \(8 \times 8\) luma blocks of a \(16 \times 16\) macroblock
- For chroma blocks: Combined with other syntax elements
**Example: Run-Level-Last Coding**

1. Scanning and conversion into (run,level,last) events
   
   \[(0,-4,0) \quad (1,-3,0) \quad (0,2,0) \quad (1,2,0) \quad (2,-1,1)\]

2. Conversion into bitstream (without coded block pattern)
   
   \[00101111 \quad 00011110 \quad 11110010 \quad 01000001 \quad 101\] (35 bits)
Context-Adaptive Variable Length Coding (CAVLC)

Variable-Length Coding in H.264 | AVC (design for $4 \times 4$ blocks)
- Separate coding of non-zero levels and runs (i.e., location of non-zero levels)
- Reverse scan (more stable statistics)
- Conditional tables (exploitation of conditional probabilities)

Coding of non-zero levels

1. Syntax element `coeff_token` specifying
   - Number of non-zero levels in transform block
   - Number of trailing levels with absolute value equal to 1 (at most 3)
   - Multiple tables (selected table depends on number of non-zero levels in neighboring blocks)

2. Signs of trailing ones (1 bit per sign, reverse scanning order)

3. Remaining non-zero levels (i.e., except trailing ones)
   - Reverse scanning order
   - Use codeword table depends on values of already transmitted levels
Context-Adaptive Variable Length Coding (CAVLC)

**Coding of runs: Location of non-zero levels**

1. Total number of zeros before last non-zero level in coding order
   - Use codeword table depends on number of non-zero coefficients

2. Runs (in reverse scanning order)
   - Maximum possible value is given by total number of zeros and already transmitted runs
   - Codeword table is selected accordingly
   - Nothing is transmitted if run can be inferred to be equal to zero
   - Last run (number of zeros before first coefficient) is not transmitted (can be inferred)

**Advantages of CAVLC**

- Exploitation of conditional probabilities of designing codeword tables
- Exploitation of more dependencies between coefficients
- Exploitation of dependencies between neighboring blocks
CAVLC Example

scanned levels:  5  -3  0  0  0  1  0  -1  0  0  -1  0  0  0  0  0

coeff_token:    (5,3)  [ 5 non-zero levels, 3 trailing ones ]
signs of trailing ones:  1  1  0  [ in reverse scanning order ]
remaining non-zero levels:  -3  5  [ in reverse scanning order ]
total number of zeros:  6
runs:  2  1  3  [ in reverse scanning order ]
Context-based Adaptive Binary Arithmetic Coding

Arithmetic Coding

- Block coding without storage of codeword table
- Suboptimal (in comparison to Huffman code for same block size)
- Many advantages
  - Iterative codeword construction (e.g., over entire picture)
  - Easy incorporation of conditional probabilities
  - Easy incorporation of adaptive probability models

Arithmetic Coding in Image and Video Coding Standards

- Already included in JPEG, H.263, ...
  - Rarely used in practice (suboptimal design)
- **Context-based Adaptive Binary Arithmetic Coding (CABAC)**
  - Alternative entropy coding method in H.264 | AVC
  - Only entropy coding method in H.265 | HEVC
Shannon-Fano-Elias Coding

Basic Idea

- Order all symbol sequences with $N$ symbols: $s_0, s_1, s_2, \cdots$
- Each symbol sequence is associated with an interval of the cdf $F(s)$
- Transmit any number (as binary fraction) inside the corresponding interval
  - Required number of bits depends on probability of symbol sequence

$F(s)$ transmitted binary fraction: $0.011_b$ (codeword “011”)

$p(s_k) = F(s_k) - F(s_{k-1})$
Iterative Determination of Interval Boundaries

Initialization of probability interval
- Interval width: $W_0 = 1$
- Lower interval boundary: $L_0 = 0$

Iterative refinement of probability interval
- Interval width: $W_{n+1} = W_n \cdot P(s_n)$
- Lower interval boundary: $L_{n+1} = L_n + W_n \cdot c(s_n)$ with $c(s) = \sum_{a < s} P(a)$
Codeword Selection

\[ L_0 + W_0 = 1 \]

\[ L_0 = 0 \]

\[ L_N + W_N \]

How many bits for interval representative \( v \)?

- On the safe side if we use \( K = \lceil -\log_2 W \rceil \) bits
- Neighboring values with \( K \) bits (after point) are \( 2^{-K} \geq W \) apart
- One of the values \( v = z \cdot 2^{-K} \) is inside interval

Which bit sequence?

- Value guaranteed to be inside interval \( v = \lceil L \cdot 2^K \rceil \cdot 2^{-K} \)
- Bit sequence: Binary representation of \( z = \lceil L \cdot 2^K \rceil \) (with \( K \) bits)
Encoding and Decoding

\[ L_0 + W_0 = 1 \]

\[ L_0 = 0 \]

Encoding

- Iterative interval refinement for symbol sequence
- Determine number of bits \( K = \lceil -\log_2 W \rceil \) and bit sequence \( z = \lceil L \cdot 2^K \rceil \)

Decoding

- For each symbol to be decoded:
  - Determine sub-intervals for possible letters
  - Choose interval that contains value \( v = z \cdot 2^{-K} \) (given by bit sequence)
  - Refine interval for decoded symbol
Arithmetic Coding

Finite-Precision Variant of Shannon-Fano-Elias Coding

- Represent $p(a)$ and $W$ with finite number of bits (integers of $V$ and $U$ bits)
- Ensure that intervals do not overlap!
- Loss in coding efficiency due to rounding is typically negligible

$\Rightarrow$ Representation of lower interval boundary $L$ has the structure

$$L_n = 0.z_{n-U} \begin{array}{llll} \text{settled bits} & \text{active bits} & \text{outstanding bits} & \text{trailing bits} \\ \overline{aaaaa \cdots a} & \overline{011111 \cdots 1} & \overline{xxxxx \cdots x} & \overline{00000 \cdots} \end{array}$$

$z_n - c_n - U$

$\Rightarrow$ **Trailing bits:** Equal to 0, but maybe changed later

$\Rightarrow$ **Active bits:** Directly modified by the update $L_{n+1} = L_n + W_n c(s_n)$

$\Rightarrow$ **Outstanding bits:** May be modified by a carry from the active bits

$\Rightarrow$ **Settled bits:** Not modified in any following interval update
Context-based Adaptive Binary Arithmetic Coding (CABAC)

- **Binary arithmetic coding**: Simplest form, requires binarization
- **Adaptive coding**: Adaptive probability models
- **Context-based coding**: Switchable / conditional probability models
- Includes fast non-adaptive bypass mode with $P(0) = P(1) = 0.5$
**CABAC: Binarization**

- Map syntax elements to binary representation (sequences of bins)
- Can use any prefix code (Note: Entropy does not change)
- Typical simple codes such as (truncated) unary or exp. Golomb codes

<table>
<thead>
<tr>
<th>$s$</th>
<th>fixed-length</th>
<th>unary</th>
<th>truncated unary</th>
<th>exp. Golomb</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>01</td>
<td>01</td>
<td>010</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>001</td>
<td>001</td>
<td>011</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>0001</td>
<td>0001</td>
<td>00100</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>00001</td>
<td>00001</td>
<td>00101</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>000001</td>
<td>000001</td>
<td>00110</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>0000001</td>
<td>0000001</td>
<td>00111</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>00000001</td>
<td>0000000</td>
<td>0001000</td>
</tr>
</tbody>
</table>

- Probability models for individual bins and sets of bins
CABAC: Adaptive Probability Models

Concept of Adaptive Probability Models

- Estimate binary pmf \( \{p_0, 1 - p_0\} \) based on actually coded bins \( \{b\} \)
- Same estimation procedure at encoder and decoder
- Represent pmf by
  - Probability of least probable symbol (LPS): \( p_{\text{LPS}} \)
  - Value of most probable symbol (MPS): \( v_{\text{MPS}} \)
- Probability estimator based on a model of “exponential aging”

\[
p_{\text{LPS}}^{n+1} = \begin{cases} 
\alpha \cdot p_{\text{LPS}}^n & : b = v_{\text{MPS}} \\
1 - \alpha \cdot (1 - p_{\text{LPS}}^n) & : \text{otherwise}
\end{cases}
\]

with \( \alpha = \left( \frac{0.01875}{0.5} \right)^{\frac{1}{63}} \)

Implementation Adaptive Probability Models

- 126 probability states with \( p \in [0.01875; 0.98125] \)
- Represented by 7 bits (6 bits for \( p_{\text{LPS}} \) + 1 bit for \( v_{\text{MPS}} \))
- Update via table look-up

\[
p_{\text{state}} = \text{stateTable}[p_{\text{state}}][b]
\]
CABAC: Arithmetic Coding Engine

Binary Arithmetic Coding Engine
- Most simple variant of arithmetic coding
- Decoder search is reduced to one comparison (value above threshold)
- CABAC: Multiplication-free algorithm

CABAC: Two coding mode
- Regular coding mode (with update of probability model)
- Bypass mode \( (p_0 = p_1 = 0.5, \text{ faster}) \)

Design of arithmetic coding for syntax elements
- What binarization?
- How many probability models (for which bins)?
- How to select probability models (usage of conditional models)?
Transform Coefficient Coding with CABAC in H.264 | AVC

CABAC / CABAC Coefficient Coding in H.264 | AVC

Coding of Significance Map (locations of non-zero coefficients)

- Flag \texttt{cbf} indicating whether block includes any non-zero levels
- For each coefficient in scanning order
  - Flag \texttt{sig} indicating whether level is equal to 0 or not
  - If sig=1: Flag \texttt{last} indicting whether last non-zero level
  - If last=1: Stop
- Used probability models for sig and last depend on position inside block

Values of non-Zero Transform Coefficient Levels

- For all coefficients with sig=1 in reverse scanning order:
  - Absolute level minus 1: Unary prefix (adaptive) + exp. Golomb (bypass)
  - Sign (bypass mode)
- Probability models (for unary prefix part) depend on
  - Number of absolute levels equal to 1 (in coding order)
  - Number of absolute levels greater than 1 (in coding order)
### Example: CABAC Coefficient Coding in H.264 | AVC

#### Zig-zag Scan
(forward for significance map, reverse for coefficient level & sign)

<table>
<thead>
<tr>
<th>significant_coeff_flag</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>last_significant_coeff_flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>coeff_abs_level_minus1</td>
<td>8</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>coeff_sign_flag</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Coefficients are coded based on $4 \times 4$ subblocks

- Reverse scan (of subblocks and inside subblocks)
- Coded block flag indicating whether block contains non-zero levels
- $x$ and $y$ coordinate of first non-zero level in coding order
- Starting with $4 \times 4$ subblock that contains first non-zero level:
  - Subblocks as illustrated on next slide
Coded subblock flag: Whether subblock contains non-zero levels

For all levels: sig flag
  - Indicates whether level is non-zero (not transmitted if it can be inferred)

For first 8 levels with sig=1: greater1 flag
  - Indicates whether absolute value is greater than 1

For first level with greater1=1: greater2 flag
  - Indicates whether absolute value is greater than 2

For all levels with sig=1: sign flag
  - Indicates sign of transform coefficient level (bypass mode)

For all levels for which absolute level is not already specified
  - Remainder of absolute value (bypass mode, adaptive binarization)

Selected probability models for sig, greater1, greater2 depend on
  - Transform block size and location in block
  - Values of coded subblock flags in neighborhood
Example: CABAC Coefficient Coding in H.265 | HEVC

```
<table>
<thead>
<tr>
<th>x</th>
<th>9</th>
<th>0</th>
<th>0</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>-6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Diagonal Scan (same direction for all)

<table>
<thead>
<tr>
<th>last_sig_coeff_x</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>last_sig_coeff_y</td>
<td>0</td>
</tr>
<tr>
<td>sig_coeff_flag</td>
<td>1* 0 1 0 0 0 0 1 0 1 1</td>
</tr>
<tr>
<td>coeff_abs_level_greater1_flag</td>
<td>0 0 1 1 1</td>
</tr>
<tr>
<td>coeff_abs_level_greater2_flag</td>
<td>1</td>
</tr>
<tr>
<td>coeff_abs_level_remaining</td>
<td>0 4 7</td>
</tr>
<tr>
<td>coeff_sign_flag</td>
<td>0 1 0</td>
</tr>
</tbody>
</table>
```
Comparison of Discussed Transform Coefficient Coding

Experimental Comparison

- Compare actual coding with sum of first-order entropies for 8×8 blocks
- Same transform coefficient levels for all approaches
  - Run-level coding of H.262 | MPEG-2 Video
  - Run-level-last coding of MPEG-4 Visual
  - CABAC of H.265 | MPEG-H HEVC
**Summary**

**Part Summary**

**Run-Level Coding**
- Structural constrained V2V codes
- Combination with suitable scanning (e.g., zig-zag scan)
- Used in JPEG, MPEG-2 Video

**Improved Run-Level Coding Approaches**
- Run-Level-Last Coding (H.263, MPEG-4 Visual)
- Context-Adaptive Variable-Length Coding (CAVLC in H.264 | MPEG-4 AVC)

**Context-based Adaptive Binary Arithmetic Coding (CABAC)**
- Binary arithmetic coding
- Adaptive probability models (also switched models)
- CABAC in H.264 | MPEG-4 AVC
- CABAC in H.265 | MPEG-H HEVC (for multiple block sizes)