Shannon-Fano-Elias coding

Suppose that we have a memoryless source X_t taking values in the alphabet $\{1, 2, \ldots, L\}$. Suppose that the probabilities for all symbols are strictly positive: $p(i) > 0, \ \forall i$.
The cumulative distribution function $F(i)$ is defined as

$$F(i) = \sum_{k \leq i} p(k)$$

$F(i)$ is a step function where the step in k has the height $p(k)$. Also consider the modified cumulative distribution function $\bar{F}(i)$

$$\bar{F}(i) = \sum_{k < i} p(k) + \frac{1}{2} p(i)$$

The value of $\bar{F}(i)$ is the midpoint of the step for i.

Shannon-Fano-Elias coding

Since all probabilities are positive, $F(i) \neq F(j)$ for $i \neq j$. Thus we can determine i if we know $\bar{F}(i)$. The value of $\bar{F}(i)$ can be used as a codeword for i.
In general $\bar{F}(i)$ is a real number with an inifinite number of bits in its binary representation, so we can not use the exact value as a codeword.
If we instead use an approximation with a finite number of bits, what precision do we need, ie how many bits do we need in the codeword?
Suppose that we truncate the binary representation of $\bar{F}(i)$ to l_i bits (denoted by $\lfloor \bar{F}(i) \rfloor_{l_i}$). Then we have that

$$\bar{F}(i) - \lfloor \bar{F}(i) \rfloor_{l_i} < \frac{1}{2^{l_i}}$$

If we let $l_i = \lceil -\log p(i) \rceil + 1$ then

$$\frac{1}{2^{l_i}} = 2^{-l_i} = 2^{-\lceil -\log p(i) \rceil - 1} \leq 2^{\log p(i) - 1} = \frac{p(i)}{2} = \bar{F}(i) - F(i-1)$$

$$\Rightarrow \lfloor \bar{F}(i) \rfloor_{l_i} > F(i-1)$$

Shannon-Fano-Elias coding

Thus the number $\lfloor \bar{F}(i) \rfloor_{l_i}$ is in the step corresponding to i and therefore $l_i = \lceil -\log p(i) \rceil + 1$ bits are enough to describe i.

Is the constructed code a prefix code?

Suppose that a codeword is $b_1 b_2 \ldots b_l$. These bits represent the interval $[0.b_1 b_2 \ldots b_l \quad 0.b_1 b_2 \ldots b_l + \frac{1}{2^l})$ In order for the code to be prefix code, all intervals must be disjoint.

The interval for the codeword of symbol i has the length 2^{-l_i} which is less than half the length of the corresponding step. The starting point of the interval is in the lower half of the step. This means that the end point of the interval is below the top of the step. This means that all intervals are disjoint and that the code is a prefix code.

Shannon-Fano-Elias coding

What is the rate of the code? The mean codeword length is

$$\begin{aligned}\bar{l} &= \sum_i p(i) \cdot l_i = \sum_i p(i) \cdot (\lceil -\log p(i) \rceil + 1) \\ &< \sum_i p(i) \cdot (-\log p(i) + 2) = -\sum_i p(i) \cdot \log p(i) + 2 \cdot \sum_i p(i) \\ &= H(X_i) + 2\end{aligned}$$

We code one symbol at a time, so the rate is $R = \bar{l}$. The code is thus not an optimal code and is a little worse than for instance a Huffman code.

The performance can be improved by coding several symbols in each codeword. This leads to arithmetic coding.

Arithmetic coding

Arithmetic coding is in principle a generalization of Shannon-Fano-Elias coding to coding symbol sequences instead of coding single symbols.

Suppose that we want to code a sequence $\mathbf{x} = x_1, x_2, \ldots, x_n$.
Start with the whole probability interval $[0, 1)$. In each step divide the interval proportional to the cumulative distribution $F(i)$ and choose the subinterval corresponding to the symbol that is to be coded.

If we have a memory source the intervals are divided according to the conditional cumulative distribution function.

Each symbol sequence of length n uniquely identifies a subinterval. The codeword for the sequence is a number in the interval. The number of bits in the codeword depends on the interval size, so that a large interval (ie a sequence with high probability) gets a short codeword, while a small interval gives a longer codeword.

Iterative algorithm

Suppose that we want to code a sequence $\mathbf{x} = x_1, x_2, \ldots, x_n$. We denote the lower limit in the corresponding interval by $l^{(n)}$ and the upper limit by $u^{(n)}$. The interval generation is then given iteratively by

$$\begin{cases} l^{(j)} = l^{(j-1)} + (u^{(j-1)} - l^{(j-1)}) \cdot F(x_j - 1) \\ u^{(j)} = l^{(j-1)} + (u^{(j-1)} - l^{(j-1)}) \cdot F(x_j) \end{cases}$$

for $j = 1, 2, \ldots, n$

Starting values are $l^{(0)} = 0$ and $u^{(0)} = 1$.

$F(0) = 0$

The interval size is of course equal to the probability of the sequence

$$u^{(n)} - l^{(n)} = p(\mathbf{x})$$

Codeword

The codeword for an interval is the shortest bit sequence $b_1 b_2 \dots b_k$ such that the binary number $0.b_1 b_2 \dots b_k$ is in the interval and that all other numbers staring with the same k bits are also in the interval.
Given a binary number a in the interval $[0, 1)$ with k bits $0.b_1 b_2 \dots b_k$.
All numbers that have the same k first bits as a are in the interval $[a, a + \frac{1}{2^k})$.
A necessary condition for all of this interval to be inside the interval belonging to the symbol sequence is that it is less than or equal in size to the symbol sequence interval, ie

$$p(\mathbf{x}) \geq \frac{1}{2^k} \quad \Rightarrow \quad k \geq \lceil -\log p(\mathbf{x}) \rceil$$

We can't be sure that it is enough with $\lceil -\log p(\mathbf{x}) \rceil$ bits, since we can't place these intervals arbitrarily. We can however be sure that we need at most one extra bit. The codeword length $l(\mathbf{x})$ for a sequence $\mathbf{x}$ is thus given by

$$l(\mathbf{x}) = \lceil -\log p(\mathbf{x}) \rceil \quad \text{or} \quad l(\mathbf{x}) = \lceil -\log p(\mathbf{x}) \rceil + 1$$

Mean codeword length

$$
\begin{aligned}
\bar{l} &= \sum_{\mathbf{x}} p(\mathbf{x}) \cdot l(\mathbf{x}) \leq \sum_{\mathbf{x}} p(\mathbf{x}) \cdot (\lceil -\log p(\mathbf{x}) \rceil + 1) \\
&< \sum_{\mathbf{x}} p(\mathbf{x}) \cdot (-\log p(\mathbf{x}) + 2) = -\sum_{\mathbf{x}} p(\mathbf{x}) \cdot \log p(\mathbf{x}) + 2 \cdot \sum_{\mathbf{x}} p(\mathbf{x}) \\
&= H(X_1 X_2 \ldots X_n) + 2
\end{aligned}
$$

The resulting data rate is thus bounded by

$$R < \frac{1}{n} H(X_1 X_2 \ldots X_n) + \frac{2}{n}$$

This is a little worse than the rate for an extended Huffman code, but extended Huffman codes are not practical for large n. The complexity of an arithmetic coder, on the other hand, is independent of how many symbols n that are coded. In arithmetic coding we only have to find the codeword for a particular sequence and not for all possible sequences.

Memory sources

When doing arithmetic coding of memory sources, we let the interval division depend on earlier symbols, ie we use different F in each step depending on the value of earlier symbols.

For example, if we have a binary Markov source X_t of order 1 with alphabet $\{1, 2\}$ and transition probabilities $p(x_t|x_{t-1})$

$$p(1|1) = 0.8, \quad p(2|1) = 0.2, \quad p(1|2) = 0.1, \quad p(2|2) = 0.9$$

we will use two different conditional cumulative distribution functions $F(x_t|x_{t-1})$

$$F(0|1) = 0, \quad F(1|1) = 0.8, \quad F(2|1) = 1$$

$$F(0|2) = 0, \quad F(1|2) = 0.1, \quad F(2|2) = 1$$

For the first symbol in the sequence we can either choose one of the two distributions or use a third cumulative distribution function based on the stationary probabilities.

Practical problems

When implementing arithmetic coding we have a limited precision and can't store interval limits and probabilities with aribtrary resolution.

We want to start sending bits without having to wait for the whole sequence with n symbols to be coded.

One solution is to send bits as soon as we are sure of them and to rescale the interval when this is done, to maximally use the available precision.

If the first bit in both the lower and the upper limits are the same then that bit in the codeword must also take this value. We can send that bit and thereafter shift the limits left one bit, ie scale up the interval size by a factor 2.

Fixed point arithmetic

Arithmetic coding is most often implemented using fixed point arithmetic.

Suppose that the interval limits $l^{(j)}$ and $u^{(j)}$ are stored as integers with m bits precision and that the cumulative distribution function $F(i)$ is stored as an integer with k bits precision. The algorithm can then be modified to

$$l^{(j)} = l^{(j-1)} + \lfloor \frac{(u^{(j-1)} - l^{(j-1)} + 1)F(x_j - 1)}{2^k} \rfloor$$

$$u^{(j)} = l^{(j-1)} + \lfloor \frac{(u^{(j-1)} - l^{(j-1)} + 1)F(x_j)}{2^k} \rfloor - 1$$

Starting values are $l^{(0)} = 0$ and $u^{(0)} = 2^m - 1$.

Note that previously when we had continuous intervals, the upper limit pointed to the first number in the next interval. Now when the intervals are discrete, we let the upper limit point to the last number in the current interval.

Interval scaling

The cases when we should perform an interval scaling are:

1. The interval is completely in $[0, 2^{m-1} - 1]$, ie the most significant bit in both $l^{(j)}$ and $u^{(j)}$ is 0. Shift out the most significant bit of $l^{(j)}$ and $u^{(j)}$ and send it. Shift a 0 into $l^{(j)}$ and a 1 into $u^{(j)}$.
2. The interval is completely in $[2^{m-1}, 2^m - 1]$, ie the most significant bit in both $l^{(j)}$ and $u^{(j)}$ is 1. Shift out the most significant bit of $l^{(j)}$ and $u^{(j)}$ and send it. Shift a 0 into $l^{(j)}$ and a 1 into $u^{(j)}$. The same operations as in case 1.

When we have coded our n symbols we finish the codeword by sending all m bits in $l^{(n)}$. The code can still be a prefix code with fewer bits, but the implementation of the decoder is much easier if all of $l^{(n)}$ is sent. For large n the extra bits are neglible. We probably need to pack the bits into bytes anyway, which might require padding.

More problems

Unfortunately we can still get into trouble in our algorithm, if the first bit of l always is 0 and the first bit of u always is 1. In the worst case scenario we might end up in the situation that $l = 011\ldots11$ and $u = 100\ldots00$. Then our algorithm will break down.

Fortunately there are ways around this. If the first two bits of l are 01 and the first two bits of u are 10, we can perform a bit shift, without sending any bits of the codeword. Whenever the first bit in both l and u then become the same we can, besides that bit, also send one extra inverted bit because we are then sure of if the codeword should have 01 or 10.

Interval scaling

We now get three cases

1. The interval is completely in $[0, 2^{m-1} - 1]$, ie the most significant bit in both $l^{(j)}$ and $u^{(j)}$ is 0. Shift out the most significant bit of $l^{(j)}$ and $u^{(j)}$ and send it. Shift a 0 into $l^{(j)}$ and a 1 into $u^{(j)}$.
2. The interval is completely in $[2^{m-1}, 2^m - 1]$, ie the most significant bit in both $l^{(j)}$ and $u^{(j)}$ is 1. The same operations as in case 1.
3. We don't have case 1 or 2, but the interval is completely in $[2^{m-2}, 2^{m-1} + 2^{m-2} - 1]$, ie the two most significant bits are 01 in $l^{(j)}$ and 10 in $u^{(j)}$. Shift out the most significant bit from $l^{(j)}$ and $u^{(j)}$. Shift a 0 into $l^{(j)}$ and a 1 into $u^{(j)}$. Invert the new most significant bit in $l^{(j)}$ and $u^{(j)}$. Don't send any bits, but keep track of how many times we do this kind of rescaling. The next time we do a rescaling of type 1, send as many extra ones as the number of type 3 rescalings. In the same way, the next time we do a rescaling of type 2 we send as many extra zeros as the number of type 3 rescalings.

Finishing the codeword

When we have coded our n symbols we finish the codeword by sending all m bits in $l^{(n)}$ (actually, any number between $l^{(n)}$ and $u^{(n)}$ will work). If we have any rescalings of type 3 that we haven't taken care of yet, we should add that many inverted bits after the first bit.

For example, if $l^{(n)} = 11010100$ and there are three pending rescalings of type 3, we finish off the codeword with the bits $1\underline{000}1010100$.

Demands on the precision

We must use a datatype with at least $m + k$ bits to be able to store partial results of the calculations.

We also see that the smallest interval we can have without performing a rescaling is of size $2^{m-2} + 2$, which for instance happens when $l^{(j)} = 2^{m-2} - 1$ and $u^{(j)} = 2^{m-1}$.
m must be large enough, so that we can always fit all the L subintervals inside this interval. A necessary (but not sufficient) condition is thus

$$L \leq 2^{m-2} + 2$$

For the algorithm to work, $u^{(j)}$ can never be smaller than $l^{(j)}$ (the same value is allowed, because when we do a rescaling we shift zeros into l and ones into u). In order for this to be true, all intervals of the fixed point version of the cumulative distribution function must fulfill (with a slight overestimation)

$$F(i) - F(i-1) \geq 2^{k-m+2} \quad ; \quad i = 1, \ldots, L$$

Decoding

Start the decoder in the same state (ie $l = 0$ and $u = 2^m - 1$) as the coder. Introduce t as the m first bits of the bit stream (the codeword). At each step we calculate the number

$$\lfloor \frac{(t - l + 1) \cdot 2^k - 1}{u - l + 1} \rfloor$$

Compare this number to F to see what probability interval this corresponds to. This gives one decoded symbol. Update l and u in the same way that the coder does. Perform any needed shifts (rescalings). Each time we rescale l and u we also update t in the same way (shift out the most significant bit, shift in a new bit from the bit stream as new least significant bit. If the rescaling is of type 3 we invert the new most significant bit.) Repeat until the whole sequence is decoded.

Note that we need to send the number of symbols coded as side information, so the decoder knows when to stop decoding. Alternatively we can introduce an extra symbol into the alphabet, with lowest possible probability, that is used to mark the end of the sequence.

Adaptive arithmetic coding

Arithmetic coding is relatively easy to make adaptive, since we only have to make an adaptive probability model, while the actual coder is fixed.

Unlike Huffman coding we don't have to keep track of a code tree. We only have to estimate the probabilities for the different symbols by counting how often they have appeared.

It is also relatively easy to code memory sources by keeping track of conditional probabilities.

Example

Memoryless model, $\mathcal{A} = \{a, b\}$. Let n_a and n_b keep track of the number of times a and b have appeared earlier. The estimated probabilities are then

$$p(a) = \frac{n_a}{n_a + n_b} \ , \quad p(b) = \frac{n_b}{n_a + n_b}$$

Suppose we are coding the the sequence *aababaaabba*...

Starting values: $n_a = n_b = 1$.

Code a, with probabilities $p(a) = p(b) = 1/2$.

Update: $n_a = 2,\ n_b = 1$.

Code a, with probabilities $p(a) = 2/3,\ p(b) = 1/3$.
Update: $n_a = 3,\ n_b = 1$.

Example, cont.

Code b, with probabilities $p(a) = 3/4,\ p(b) = 1/4$.

Update: $n_a = 3,\ n_b = 2$.

Code a, with probabilities $p(a) = 3/5,\ p(b) = 2/5$.

Update: $n_a = 4,\ n_b = 2$.

Code b, with probabilities $p(a) = 2/3,\ p(b) = 1/3$.

Update: $n_a = 4,\ n_b = 3$.

et cetera.

Example, cont.

Markov model of order 1, $\mathcal{A} = \{a, b\}$. Let $n_{a|a}, n_{b|a}, n_{a|b}$ and $n_{b|b}$ keep track of the number of times symbols have appeared, given the previous symbol. The estimated probabilities are then

$$p(a|a) = \frac{n_{a|a}}{n_{a|a} + n_{b|a}} \ , \quad p(b|a) = \frac{n_{b|a}}{n_{a|a} + n_{b|a}}$$

$$p(a|b) = \frac{n_{a|b}}{n_{a|b} + n_{b|b}} \ , \quad p(b|b) = \frac{n_{b|b}}{n_{a|b} + n_{b|b}}$$

Suppose that we are coding the sequence *aababaaabba*...

Assume that the symbol before the first symbol was an a.

Starting values: $n_{a|a} = n_{b|a} = n_{a|b} = n_{b|b} = 1$.

Code a, with probabilities $p(a|a) = p(b|a) = 1/2$.

Update: $n_{a|a} = 2,\ n_{b|a} = 1,\ n_{a|b} = 1,\ n_{b|b} = 1$.

Example, cont.

Code a, with probabilities $p(a|a) = 2/3,\ p(b|a) = 1/3$.

Update: $n_{a|a} = 3,\ n_{b|a} = 1,\ n_{a|b} = 1,\ n_{b|b} = 1$.

Code b, with probabilities $p(a|a) = 3/4,\ p(b|a) = 1/4$.

Update: $n_{a|a} = 3,\ n_{b|a} = 2,\ n_{a|b} = 1,\ n_{b|b} = 1$.

Code a, with probabilities $p(a|b) = 1/2,\ p(b|b) = 1/2$.

Update: $n_{a|a} = 3,\ n_{b|a} = 2,\ n_{a|b} = 2,\ n_{b|b} = 1$.

Code b, with probabilities $p(a|a) = 3/5,\ p(b|a) = 2/5$.

Update: $n_{a|a} = 3,\ n_{b|a} = 3,\ n_{a|b} = 2,\ n_{b|b} = 1$.

et cetera.

Updates

The counters are updated after coding the symbol. The decoder can perform exactly the same updates after decoding a symbol, so no side information about the probabilities is needed.

If we want more recent symbols to have a greater impact on the probability estimate than older symbols we can use a forgetting factor.

For example, in our first example, when $n_a + n_b > N$ we divide all counters with a factor K:

$$n_a \mapsto \lceil \frac{n_a}{K} \rceil \ , \quad n_b \mapsto \lceil \frac{n_b}{K} \rceil$$

Depending on how we choose N and K we can control how fast we forget older symbols, ie we can control how fast the coder will adapt to changes in the statistics.

Implementation

Instead of estimating and scaling the cumulative distribution function to k bits fixed point precision, you can use the counters directly in the interval calculations. Given the alphabet $\{1, 2, \ldots, L\}$ and counters $n_1, n_2, \ldots, n_L$, calculate

$$F(i) = \sum_{k=1}^{i} n_k$$

$F(L)$ will also be the total number of symbols seen so far.

The iterative interval update can then be done as

$$l^{(n)} = l^{(n-1)} + \lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1)F(x_n - 1)}{F(L)} \rfloor$$

$$u^{(n)} = l^{(n-1)} + \lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1)F(x_n)}{F(L)} \rfloor - 1$$

Instead of updating counters after coding a symbol we could of course just update F.

Demands on the counters

Redoing our calculations for the demands on the cumulative distribution function, we arrive at

$$n_i = F(i) - F(i-1) \geq F(L) \cdot 2^{2-m} \quad ; \quad i = 1, \ldots, L$$

Since we want to be able to have $n_i = 1$, this gives us

$$F(L) \leq 2^{m-2}$$

This gives us a limit on how much data we can code before we need to rescale the counters.

For example, if we have $m = 16$, we must rescale the counters at least every time the sum of the counters becomes $2^{14} = 16384$. In most practical situations it is better to rescale even more often than this.

Prediction with Partial Match (ppm)

If we want to do arithmetic coding with conditional probabilities, where we look a many previous symbols (corresponding to a Markov model of high order) we have to store many counters.

Instead of storing all possible combinations of previous symbols (usually called *contexts*), we only store the ones that have actually happened. We must then add an extra symbol (escape) to the alphabet to be able to tell when something new happens.

In ppm contexts of different lengths are used. We choose a maximum context length N (some variants allow unbounded context lengths). When we are about to code a symbol we first look at the longest context. If the symbol has appeared before in that context we code the symbol with the corresponding estimated probability, otherwise we code an escape symbol and continue with a smaller context. If the symbol has not appeared at all before in any context, we code the symbol with a uniform distribution.

After coding the symbol we update the counters of the contexts and create any possible new contexts.

ppm cont.

There are many variants of ppm. The main difference between them is how the counters (the probabilities) for the escape symbol are handled.

Assume that n_j is the number of times we have seen symbol j in the current context and n_{esc} is a counter for the escape symbol in the current context. One simple way of estimating the symbol probabilities $p(j)$ and the escape probability $p(\text{esc})$ is by

$$p(j) = \frac{n_j}{n_{\text{esc}} + \sum_{i=1}^{L} n_i} \quad , \quad p(\text{esc}) = \frac{n_{\text{esc}}}{n_{\text{esc}} + \sum_{i=1}^{L} n_i}$$

Some variants:

- Always keep $n_{\text{esc}} = 1$
- Treat the escape symbol as a regular symbol, ie n_{esc} counts how many times we have coded the escape symbol in this context.

ppm, exclusions

When we code an escape symbol we know that the next symbol can't be any of the symbols that have appeared in that context before. These symbols can therefore be excluded when coding with a shorter context, ie we temporarily set the probability of those symbols to 0.

Another thing to consider is *update exclusion*. When coding a symbol in a context, should we also update the probability for that symbol in the shorter contexts or not? Experiments have shown that if we use a maximum context length, then update exclusion gives a slightly better performance (ie lower rate). For variants of ppm where we have unbounded context lengths, update exclusion might not give a better performance.

Compression of test data

	world192.txt	alice29.txt	xargs.1
original size	2473400	152089	4227
pack	1558720	87788	2821
Adaptive arithmetic, order 0	1528235	86691	2628
Adaptive arithmetic, order 1	1126126	66160	2219
Adaptive arithmetic, order 2	882201	55135	2378
ppmd	374361	38654	1512

pack is a memoryless static Huffman coder.

Compression of test data

The same data as the previous slide, but with performance given as bits per character. A comparison is also made with some estimated entropies.

	`world192.txt`	`alice29.txt`	`xargs.1`
original size	8	8	8
`pack`	5.04	4.62	5.34
Adaptive arithmetic, order 0	4.94	4.56	4.97
Adaptive arithmetic, order 1	3.64	3.48	4.20
Adaptive arithmetic, order 2	2.85	2.90	4.50
`ppmd`	1.21	2.03	2.86
$H(X_i)$	5.00	4.57	4.90
$H(X_i \mid X_{i-1})$	3.66	3.42	3.20
$H(X_i \mid X_{i-1}, X_{i-2})$	2.77	2.49	1.55

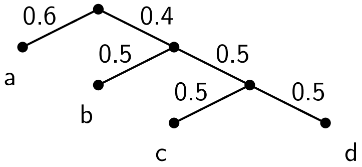
Binary arithmetic coders

Any distribution over an alphabet of arbitrary size can be described as a sequence of binary choices, so it is no big limitation to letting the arithmetic coder work with a binary alphabet.

Having a binary alphabet will simplify the coder. When coding a symbol, either the lower or the upper limit will stay the same.

Binarization, example

Given the alphabet $\mathcal{A} = \{a, b, c, d\}$ with symbol probabilities $p(a) = 0.6,\ p(b) = 0.2,\ p(c) = 0.1$ and $p(d) = 0.1$ we could for instance do the following binarization, described as a binary tree:



This means that *a* is replaced by 0, *b* by 10, *c* by 110 and *d* by 111. When coding the new binary sequence, start at the root of the tree. Code a bit according to the probabilities on the branches, then follow the corresponding branch down the tree. When a leaf is reached, start over at the root.

We could of course have done other binarizations.

The MQ coder

The MQ coder is a binary arithmetic coder used in the still image coding standards JBIG2 and JPEG2000. Its predecessor the QM coder is used in the standards JBIG and JPEG.

In the MQ coder we keep track of the lower interval limit (called C) and the interval size (called A). To maximally use the precisions in the calculations the interval is scaled with a factor 2 (corresponding to a bit shift to the left) whenever the interval size is below 0.75. The interval size will thus always be in the range $0.75 \leq A < 1.5$.

When coding, the least probable symbol (LPS) is normally at the bottom of the interval and the most probable symbol (MPS) on top of the interval.

The MQ coder, cont.

If the least probable symbol (LPS) has the probability Q_e, the updates of C and A when coding a symbol are:

When coding a MPS:

$$\begin{aligned} C^{(n)} &= C^{(n-1)} + A^{(n-1)} \cdot Q_e \\ A^{(n)} &= A^{(n-1)} \cdot (1 - Q_e) \end{aligned}$$

When coding a LPS:

$$\begin{aligned} C^{(n)} &= C^{(n-1)} \\ A^{(n)} &= A^{(n-1)} \cdot Q_e \end{aligned}$$

Since A is always close to 1, we do the approximation $A \cdot Q_e \approx Q_e$.

The MQ coder, cont.

Using this approximation, the updates of C and A become

When coding a MPS:

$$\begin{aligned} C^{(n)} &= C^{(n-1)} + Q_e \\ A^{(n)} &= A^{(n-1)} - Q_e \end{aligned}$$

When coding a LPS:

$$\begin{aligned} C^{(n)} &= C^{(n-1)} \\ A^{(n)} &= Q_e \end{aligned}$$

Thus we get an arithmetic coder that is multiplication free, which makes it easy to implement both in soft- and hardware.

Note that because of the approximation, when $A < 2Q_e$ we might actually get the situation that the LPS interval is larger than the MPS interval. The coder detects this situation and then simply switches the intervals between LPS and MPS.

The MQ coder, cont.

The MQ coder typically uses fixed point arithmetic with 16 bit precision, where C and A are stored in 32 bit registers according to

$$\begin{array}{llllllllll} C: & 0000 & cbbb & bbbb & bsss & xxxx & xxxx & xxxx & xxxx \\ A: & 0000 & 0000 & 0000 & 0000 & aaaa & aaaa & aaaa & aaaa \end{array}$$

The x bits are the 16 bits of C and the a bits are the 16 bits of A.

By convention we let $A = 0.75$ correspond to 1000 0000 0000 0000. This means that we should shift A and C whenever the 16:th bit of A becomes 0.

The MQ coder, cont.

$$\begin{array}{llllllllll} C: & 0000 & cbbb & bbbb & bsss & xxxx & xxxx & xxxx & xxxx \\ A: & 0000 & 0000 & 0000 & 0000 & aaaa & aaaa & aaaa & aaaa \end{array}$$

b are the 8 bits that are about to be sent as a byte next. Each time we shift C and A we increment a counter. When we have counted to 8 bits we have accumulated a byte. s is to ensure that we don't send the 8 most recent bits. Instead we get a short buffer in case of carry bits from the calculations. For the same reason we have c. If a carry bit propagates all the way to c we have to add one to the previous byte. In order for this bit to not give rise to carry bits to even older bytes a zero is inserted into the codeword every time a byte of all ones is found. This extra bit can be detected and removed during decoding.

Compare this to our previous solution for the 01 vs 10 situation.

Probability estimation

In JPEG, JBIG and JPEG2000 adaptive probability estimations are used. Instead of having counters for how often the symbols appear, a state machine is used, where every state has a predetermined distribution (ie the value of Q_e) and where we switch state depending on what symbol that was coded.