The Z-Coder Adaptive Binary Coder

(extended abstract)

Léon Bottou[A], *Paul G. Howard*[A], *and Yoshua Bengio*[A,M]
[A]AT&T Labs – Research, Red Bank NJ 07701 – 7033
[M]Université de Montréal, Montréal PQ Canada H3C 3J7

Abstract

We present the *Z-Coder*, a new adaptive data compression coder for coding binary data. The Z-Coder is derived from the Golomb run-length coder, and retains most of the speed and simplicity of the earlier coder. The Z-Coder can also be thought of as a multiplication-free approximate arithmetic coder, showing the close relationship between run-length coding and arithmetic coding. The Z-Coder improves upon existing arithmetic coders by its speed and its principled design. In this paper we present a derivation of the Z-Coder as well as details of the construction of its adaptive probability estimation table.

1 Introduction

Statistical compressors operate by reducing data to a sequence of discrete "events." Each event that must be coded is drawn from a set of possible events, each having a probability associated with it. The major issues involved in designing a statistical compressor are **modeling**: mapping the data to a probabilistic model by choosing a set of possible contexts; **probability estimation**: estimating the probabilities associated with each possible event in each context; and **coding**: converting the actual events to bits in the codestream in a way that allows a decoder to recover the original sequence of events. Probability estimation is properly part of modeling, but in data compression work it is usually kept separate, or even coupled to coding. The goal of any data compression system is to minimize the number of bits required to represent the data while at the same time attempting to satisfy a number of practical constraints. Some of the more important considerations besides compression efficiency are encoding and decoding speed and latency, ease of implementation in software, hardware, or both, efficient use of memory, and freedom from intellectual property restrictions. In this paper we present a new adaptive binary coder named the Z-Coder. Besides providing close to theoretically optimal compression efficiency, it scores well on the secondary issues.

In Section 2 we develop the Z-Coder as a generalization of the Golomb coder [1], a simple, effective, and popular run-length coder. To do so we first provide a careful and detailed explanation of the Golomb coder, casting it in the terms used in our description of the Z-Coder. We show why the Golomb coder is so fast and easy to implement, but we also show how its compression efficiency might be improved. Then we make relatively minor adjustments to the Golomb coder to arrive at the Z-Coder; the Z-Coder retains the most useful properties of the Golomb coder, speed and simplicity.

The Z-Coder can also be seen to be similar to the Q-Coder [5] and QM-Coder [3] approximate arithmetic coders, in that it is a multiplication-free binary coder with incremental output and adaptive probability estimation coupled to the coding process; like the other coders in this class, it does not directly address modeling. Like them, it uses tables for its adaptive probability estimation, as we describe in Section 3. However, the Z-Coder improves on the Q-Coder and its descendents in a number of ways: 1) a better approximation for proportional interval division; 2) a very fast "fast path"; 3) a principled approach to increment computation in the probability estimation tables; and 4) a principled approach to fast early adaptation in the probability estimation tables.

2 Development of the Z-Coder

Golomb coding. The Z-Coder is a generalization of Golomb coding, a method of coding runs of binary symbols. We consider the Golomb code with parameter m; in this discussion the parameter is taken to be a power of 2, although with minor adjustments it may take on any positive integer value. We use a Golomb code to code a run of r consecutive occurrences of a *more probable symbol* (MPS) followed by a single occurrence of a *less probable symbol* (LPS). The codeword for such a run has two parts. The first part is the value of $\lfloor r/m \rfloor$ coded as a unary number, that is, $\lfloor r/m \rfloor$ **1**s followed by a **0**. The second part is the value of $r \pmod{m}$, coded as an ordinary $(\log_2 m)$-bit binary number. Successive runs are coded exactly the same way.

Bin filling. We can think of Golomb encoding as an exercise in filling bins. Imagine that we have a conveyor belt supplying small objects, called MPS-objects, each of approximately the same size. Runs of MPS-objects are occasionally interrupted by a large object, called an LPS-object. At a given point in time, the conveyor belt will supply either an MPS-object of size $-\log_2 p_{\mathsf{MPS}}$, which is smaller than the size of a bin, or an LPS-object of size $-\log_2(1 - p_{\mathsf{MPS}})$, which is larger than the size of a bin. We may interpret p_{MPS} as the probability that the object is an MPS-object, although the probability is no longer relevant after we have fixed the Golomb parameter m.

We are trying to code the number of MPS-objects in each run. We have bins that can hold m MPS-objects each; each bin has m slots, each of size $\Delta = 1/m$, and the MPS-objects can be forced to expand or contract to exactly fill the slots.

We start filling bins with MPS-objects. When a bin is full, we output **1**; it is no longer necessary to consider that bin. When an LPS-object appears, we output **0**, and then output a $(\log_2 m)$-bit binary number to indicate how many MPS-objects were in the last partially filled bin. The LPS-object fills the remainder of that bin as well as $\log_2 m$ other entire bins. The initial choice of the value of m determines both the number of MPS-objects that can fit into one bin and the number of bins required for an LPS-object.

In this interpretation, the bins that are filled with MPS-objects correspond exactly to the **1** bits in the unary part of the Golomb code. The partially filled bin corresponds to the **0** bit at the end of the unary part, and the other bins occupied by the LPS-

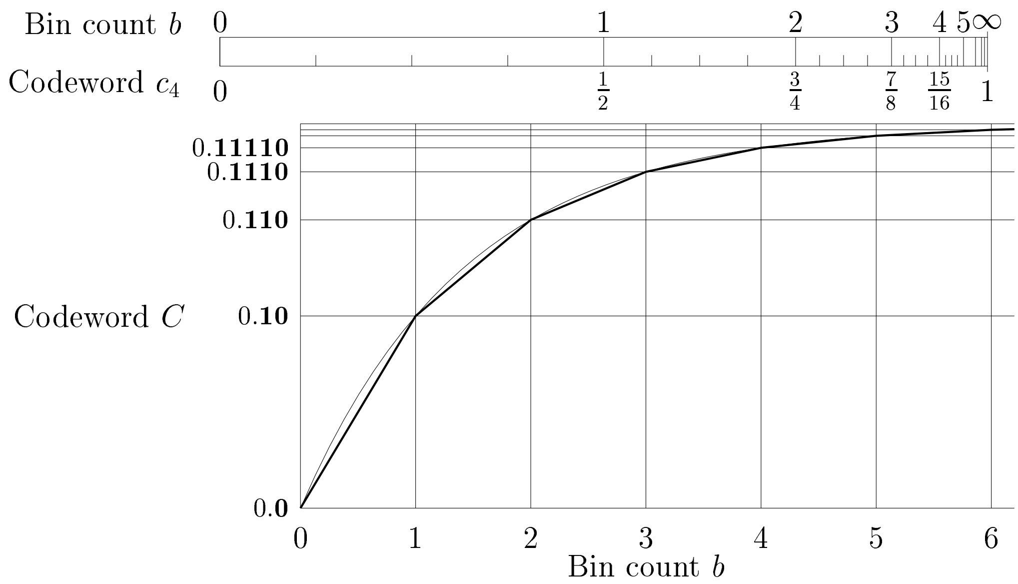

Figure 1: Two views of the mapping between bin counts and codewords. Top: the codeword space chopped up and assigned to bins and slots, assuming $m = 4$. Bottom: codeword as a function of bin count. The light curve is the function using exact arithmetic coding, $C = 1 - 2^{-b}$.

object correspond to the bits in the binary part of the Golomb code.

Interpretation of code stream. Now we consider the coded bitstream as a function of the number of bins filled with MPS-objects. We denote the bin count, a real number in the general case, by b. At the end of a run of r MPS-objects, $b = r/m$. If we put a binary point in front of the stream of output bits, the result is a number in $[0, 1)$. The beginning of this number is a codeword for the number of bins occupied by the first run of MPS-objects (or equivalently, for the length of that run). It turns out that the codeword is defined by the following piecewise linear function, in which $\{b\}$ denotes the fractional part of the bin count b:

$$C(b) = \left(1 - \frac{1}{2^{\lfloor b \rfloor}}\right) + \frac{\{b\}}{2^{1+\lfloor b \rfloor}} = 1 - \frac{2 - \{b\}}{2^{1+\lfloor b \rfloor}}. \tag{1}$$

This mapping is shown two different ways in Figure 1. The first term in the middle expression is the code for the number of full bins; the other term is the code for the fraction of the partially filled bin that is filled. If we define the codeword function in terms of the run length r to be $c_m(r) = C(r/m)$, then we have a formula for the Golomb code with parameter m.

The decoder will interpret any code stream in $[c_m(r), c_m(r+1))$ as beginning with a run of length exactly r. The less significant trailing bits of this range are used to decode the remaining runs. To allow this, we must transform the range and the code point so that the range represents a collection of possible bin counts, that is, a range of the form $[C(b), C(\infty))$. We do this by translating the range and code point upward to the high end of $[0, 1)$. The translated code point, when interpreted as a bin count, correctly includes the bins occupied by the LPS-object.

Coding algorithm. To be more precise, we now recast the coding process in terms of coding variables. These variables are typically stored in registers, but for now we treat them as real numbers. As we begin the coding, we do not know the length of the first run; it could be any integer in the range $[0, \infty)$, so the codeword could be any value in the range $[c_m(0), c_m(\infty))$. We give the name C to the *code point*, that is, the value of the code stream interpreted as a number in $[0, 1)$. We give the name A to the current *low point* of the range of possible codewords; initially $A = c_m(0) = 0$. The *high point* of the range is $c_m(\infty) = 1$.

After we have put k MPS-objects into bins, the range of possible codewords is $[c_m(k), c_m(\infty))$, so $A = c_m(k)$. If the next object were an LPS-object that terminates the run of MPS-objects, the codeword would be known to be in the range $[c_m(k), c_m(k+1))$. If the next object were an MPS-object that continues the run of MPS-objects, the new range would be $[c_m(k+1), c_m(\infty))$. We give the name Z to the *split point* $c_m(k+1)$. The decoder will interpret a value of $C \in [A, Z)$ as indicating that length of the first run was exactly k, while a value of $C \in [Z, 1)$ means that the length of the first run was more than k.

After coding the first run, we would like to put the coder back into the state in which the range of possible code words is of the form $[A', 1)$. To do so we simply shift the range and the code point upward by $1 - Z$, so the code point is $C + 1 - Z$ and the range is $[A + 1 - Z, 1)$, which is of the proper form. In the Golomb code this is easy, involving a simple fixed precision addition.

Finally we *renormalize* the range by repeatedly moving A and C to the left, doubling the size of the range at each step. In the Golomb code this is implemented by a sequence of bit shifting operations. Eventually this process makes the range $[0, 1)$, and the code point C is adjusted properly. Renormalization is not necessary for the theory, but when we recognize that we have extracted all the information from the leading bits, we can discard them and use the space they occupied in the machine registers. Hence runs of arbitrary length can be coded using $(\log_2 m)$-bit registers.

Here is pseudocode for decoding one bit, assuming that $\Delta = 1/m$, A is initially 0, and C is set to the full coded bitstream:

Golomb decoder

$Z := A + \Delta$;	increment based on slot size
if $(C > Z)$ { bit := MPS; $A := Z$; }	deal with MPS
else { bit := LPS; $A := A + 1 - Z$; $C := C + 1 - Z$; }	deal with LPS
while $(A \geq 1/2)$ { $A := 2(A - 1/2)$; $C := 2(C - 1/2)$; }	renormalize

Why Golomb coding is easy. When considered in these terms, Golomb coding is easy for three reasons. First, all the MPS-objects are forced to be the same size, the slot size, and that size is a divisor of the bin size, so no MPS-object ever straddles a bin boundary. Hence $c_m(k)$ and $c_m(k+1)$ are always in the same segment of the piecewise linear function defined by Equation (1). During the coding of a single symbol, the low point A never moves along two different slopes of the bin-count-to-codeword curve of Figure 1, so no special non-linear processing is needed to handle this case. Second, because $1 - Z$ is always a multiple of a power of 2, we do not have to deal with arithmetic carries when doing the upward shift of A and C by $1 - Z$; fixed-point addition suffices. Third, because $Z - A$ is always a power of 2, the upward shift leaves A on a bin boundary. Each renormalization step discards one bit from the code stream, and reduces the full bin count by 1. Eventually the range becomes $[0, 1)$ (and the corresponding bin count becomes 0) as we are about to start coding a new run. These conditions can be made to hold even for Golomb codes in which m is not a power of 2, but that is outside the scope of this paper.

Limitations of Golomb codes. Golomb codes, while easy to understand and implement and fast in operation, have limitations. A general data compression system has to be able to deal with arbitrary sequences of events with different probabilities[1]. But in a Golomb code the parameter m is fixed, and the slot size $\Delta = 1/m$ is appropriate only for a single event probability.

In fact we can change the value of the slot size Δ for each event to be encoded, but in doing so we may violate some of the assumptions that make Golomb coding so attractive. First, there may not be enough room in the current partially filled bin to accommodate a slot of size Δ, since Δ can take any value up to 1. This is not a problem when counting bins, but because of the non-linearity in the mapping between bin counts and codewords given by Equation (1), line G−1 of the pseudocode above is no longer sufficient to compute the value of the split point Z.

Second, when an LPS-object is encountered, $1 - Z$ need not be a multiple of a power of 2. Hence we can no longer say that fixed precision addition is sufficient to implement the upward shift of A and C by $1 - Z$. It is also true that after the upward shift, A may not be on a bin boundary, but this is not a serious problem.

The Z-Coder as enhanced Golomb coding. The Z-Coder is the same as Golomb coding with an enhancement to permit use of any arbitrary slot size Δ not exceeding one bin. The idea is that if the desired next slot would span a bin boundary, we use Equation (1) to compute the appropriate value of Z. Within the current bin, the slot size and increment correspond exactly, but in the overflow bin, since the codeword slope is only half as much, the remainder of the slot adds only half as much to the increment. In symbols, $Z - A = (1/2 - A) + (\Delta - (1/2 - A))/2$, or $Z = (A + \Delta)/2 + 1/4$. Hence we add line 1a to the pseudocode for the Golomb decoder:

Decoder for Z-Coder

Z−1	$Z := A + \Delta$;	increment based on MPS size
Z−1a	**if** $(Z > 1/2)$ $\{\ Z := Z/2 + 1/4;\}$	adjust for bin overlap
Z−2	**if** $(C > Z)$ $\{$ bit := MPS; $A := Z$; $\}$	deal with MPS
Z−3	**else** $\{$ bit := LPS; $A := A + 1 - Z$; $C := C + 1 - Z$; $\}$	deal with LPS
Z−4	**while** $(A \geq 1/2)$ $\{\ A := 2(A - 1/2);\ C := 2(C - 1/2);\ \}$	renormalize

This method of dealing with bin overlap is essentially the same as the "over-half processing" presented by Ono *et al.* [4] in the Arithmetic Melcode. It is a much more accurate and more principled approximation of proportional interval division than

[1] It is possible to use a different Golomb parameter after each renormalization, that is, after each code bit, but this is not frequent enough for maximum compression efficiency. It is also possible to interleave Golomb codes with different parameters in the same code stream, as in Block Melcode [4], but this increases encoding latency and coder complexity.

the "conditional exchange" used in the QM-Coder.

Engineering the Z-Coder for decoding speed. In the Z-Coder one case, that of an MPS with no need for renormalization or bin overlap adjustment, is typically the most frequent; it is also fast, involving just one addition and one assignment. By introducing a new variable F (the *fence*) and rearranging the computation, we can ensure that only one comparison is needed in this path, making it truly a "fast path", faster than that of the Q-Coder or QM-Coder. This optimization is possible in the Z-Coder because all the conditions that cause a departure from the "fast path" are in the same comparison direction.

Fast decoder for Z-Coder

$Z := A + \Delta;$	increment based on MPS size
if $(Z < F)$ $\{ A := Z;$ bit := MPS; $\}$	fast MPS path
else $\{$	
if $(Z > 1/2)$ $\{ Z := Z/2 + 1/4; \}$	adjust for bin overlap
if $(C > Z)$ $\{$ bit := MPS; $A := Z; \}$	deal with MPS
else $\{$ bit := LPS; $A := A + 1 - Z;$ $C := C + 1 - Z; \}$	deal with LPS
while $(A \geq 1/2)$ $\{ A := 2(A - 1/2);$ $C := 2(C - 1/2); \}$	renormalize
$F := \min(C, 1/2); \}$	new fence

The variables will all be treated as fixed-point numbers stored in fixed-length registers. Further optimizations are possible, such as unrolling code to reveal deterministic comparisons, replacing multiplications and divisions by shifts, and never storing the value of Z into main memory.

Encoding. The code for the encoder is similar to that of the decoder; it includes carry control by counting as in [7]:

Encoder for Z-Coder

$Z := A + \Delta;$	increment based on slot size
if (bit = MPS) $\{$	deal with MPS
if $(Z < 1/2)$ $\{ A := Z;$ **return**; $\}$	fast path MPS
else $\{ A := 1/4 + Z/2\}\};$	adjust for bin overlap
else $\{$	deal with LPS
if $(Z \geq 1/2)$ $\{ Z := Z/2 + 1/4; \}$	adjust for bin overlap
$C := C + 1 - Z;$ $A := C + 1 - Z; \}$	shift to top of unit interval
while $(A \geq 1/2)$ $\{$	
emit bit;	output bit (includes carry control by counting)
$A := 2(A - 1/2);$ $C := 2(C - 1/2); \}$	renormalize

3 Probability estimation

Coding a binary event is always conditioned on a state in the model of the data, called the *context* of the event. For each context we maintain three pieces of information about the probabilities of the two possible symbols: the value of the LPS (**0** or **1**), the probability p_{LPS} of the LPS, and the confidence that we have in our estimate. (We obviously could use MPS values instead of LPS values if it were convenient to do so.)

For storage efficiency in hardware implementations, we store all of this information in an 8-bit integer. Since it is used as an index into the probability estimation table, we call it the *index* of the context.

When an event is encoded or decoded, we already know its context. We use the index k of the context to retrieve the value of the increment Δ_k from the probability estimation table. This value is used as Δ in the coding process.

After coding an event, we may update the value of the context's index, adjusting our probability estimate based on the event just coded. In the Z-Coder, when an LPS occurs we always update the index, but when an MPS occurs we update the index with a probability that depends on the index. Thus, whether we adapt after an MPS depends on the value of a random quantity. Rather than using computational resources to obtain a random number using a regular pseudo-random-number generator, we use a more-or-less random quantity already present in the coder. In the Z-Coder we use the value of Z, and we do an MPS adaptation when $Z \geq \theta_k$. The probability of adaptation for a given index is adjusted through the threshold θ_k. Note that $\theta_k \geq 1/2$, so MPS adaptation occurs behind the fence and does not interfere with the speed of the "fast path."

Design of the probability estimation table involves several steps: determining a quantized set of probabilities and confidence levels, computing the increment Δ_k for each probability, determining the next index after an LPS adaptation, and simultaneously computing both the threshold for MPS adaptation and the next index after an MPS adaptation. We would like to change the estimated probability for a context rapidly at first, then less rapidly once the probability has become established. Table design is usually done empirically, but here we provide a more principled approach to the problem.

Choosing table entries. In the Z-Coder the probability estimation table is divided into two parts. The early adaptation part is used when we have seen few events and hence have little confidence in our probability estimate. The steady state part reflects more confidence in our estimate. The probability estimate for a context will pass through the early adaptation part as successive events are coded; eventually it will reach an index in the steady state part. From there we may adjust the probability estimate, but it will remain in the steady state part of the table. (It may be useful to allow a return to the early adaptation part, but we do not yet do so in the Z-Coder.)

Steady state entries. About one third of the table is reserved for steady state entries. The probabilities are selected starting at 0.5. Each successive probability is chosen as far away from the previous one as possible, but constrained so that either the absolute or the relative compression inefficiency is less than a threshold. The relative inefficiency criterion affects probabilities near $1/2$; the absolute criterion affects skewed probabilities. The thresholds control the number of steady state entries in the table. In the Z-Coder we use 78 steady state probabilities, resulting in a practically insignificant maximum coding inefficiency of 0.0003 code bit per message bit.

Early adaptation entries. The early adaptation part of the table is constructed by growing a tree. Each tree node corresponds to an $(n_\mathbf{0}, n_\mathbf{1})$ pair, $n_\mathbf{0}$ and $n_\mathbf{1}$ being the number of **0**s and **1**s seen so far in the context. For a given node the probability of seeing a **1** is $(n_\mathbf{1} + \epsilon)/(n_\mathbf{0} + n_\mathbf{1} + 2\epsilon)$. The value of ϵ can be adjusted to reflect an a priori estimate of the distribution of context probabilities and to influence the speed of adaptation: $\epsilon = 1$ corresponds to Laplace's prior, $\epsilon = {}^1\!/_2$ to Jeffreys' prior, and $\epsilon = 0$ to Haldane's prior. In the Z-Coder we find that $\epsilon = {}^1\!/_3$ works well.

We start with the $(0, 0)$ node, a leaf. Thereafter, some nodes are leaves, and others have two children. An LPS child corresponds to the node resulting from the adaptation that follows an LPS event; an MPS child corresponds to the node resulting from an MPS adaptation, which on average follows more than one MPS event.

At each step we construct both children of one node and add them to the tree. To select the pair to be added, we compute the potential child pair of each leaf. From among the pairs, we choose the one whose probabilities are farthest apart, in the sense of having the most *steady state* probability quantization levels between them. We continue to grow the tree in this greedy manner until the MPS-child-to-LPS-child probability-quantization-level difference is 1.

Optimal increment. We derive an expression for the optimal value of the increment Δ as a function of the symbol probability distribution p_{LPS}. Consider a theoretical experiment consisting of decoding a random string of independent random bits with $p_0 = p_1 = {}^1\!/_2$ using a given value Δ for the increment, then computing p_{LPS} in the decoded symbol string. We assume that random variable A is uniformly distributed in $[0, {}^1\!/_2)$, regardless of previous decoding actions. This assumption is supported by empirical evidence when the greatest common divisor of the increment Δ and the interval size (the representation of ${}^1\!/_2$ in a register) is small. By the nature of the code stream in the experiment, random variable C is uniformly distributed in $[A, 1)$. The assumption about the distribution of A eliminates the dependencies between consecutive decoded symbols. This is realistic: practical applications tend to mix many streams of symbols with different probabilities into the same arithmetic coder, efficiently randomizing A and C.

Under this assumption, the decoded symbols are independent identically distributed random variables. The probability of decoding an LPS can be derived using the decomposition $p_{\mathsf{LPS}} = \mathrm{Prob}(A + \Delta < {}^1\!/_2) \cdot \mathrm{Prob}(Z > C \mid A + \Delta < {}^1\!/_2) + \mathrm{Prob}(A + \Delta \geq {}^1\!/_2) \cdot \mathrm{Prob}(Z > C \mid A + \Delta \geq {}^1\!/_2)$. We find that

$$p_{\mathsf{LPS}} = \Delta - (\Delta + {}^1\!/_2) \log_e(\Delta + {}^1\!/_2) - (\Delta - {}^1\!/_2) \log_e {}^1\!/_2,$$

implicitly defining Δ as a function of p_{LPS}.

Decoding a random sequence of independent equiprobable bits produces a random sequence of independent symbols distributed as derived above. Conversely, encoding such a random sequence of symbols under the same assumptions will produce a random sequence of equiprobable bits. Thus Δ is the optimal increment when the probability of the LPS is p_{LPS}.

We have confirmed this formula by experiments seeking the optimum increment for chosen symbol probabilities. Encoding a random symbol string with this optimal increment produces about 0.5% more code bits than predicted by the entropy. This appears to be the cost of the additive approximation to the computation of Z.

	Z-Coder	QM-Coder		ELS Coder		Q15 Coder	
File size	3,262,334	3,265,082	+0.1%	3,247,700	−0.5%	3,278,599	+0.5%
Encode time (UltraSparc)	411.20	438.31	+6.6%	446.90	+8.7%	430.59	+4.7%
Decode time (UltraSparc)	334.73	342.98	+2.5%	385.21	+15.1%	432.71	+29.3%
Encode time (Sparc10, inlined)	1693.77	1810.32	+6.9%				
Decode time (Sparc10, inlined)	1451.38	1530.34	+5.4%				

Table 1: Comparative results. The figures are combined results for 96 binary image files, compressed with AT&T's SPM algorithm. The signed numbers are the percentage improvement obtained by using the Z-Coder. + means the Z-Coder is better, − means the Z-Coder is worse. The total size of the original files is 189,268,488 bytes.

Probability adaptation. In the early adaptation part of the table, the transitions to new entries after LPS and MPS adaptations are determined by the construction of the tree. The MPS adaptation threshold θ is always set to $1/2$ to allow fast adaptation.

In the steady state part of the table, transitions are always between adjacent table entries. To maintain a state without extra unnecessary transitions, LPS transitions and MPS transitions should be equally likely in a context with a given symbol probability. MPS adaptations are controlled by the threshold θ_k. The probability of an MPS adaptation, assuming again that the low point A is a uniform random number in $[0, 1/2)$, is $\mathrm{Prob}(Z > M | Z > 1/2) \cdot \mathrm{Prob}(Z > 1/2) \cdot p_{\mathsf{MPS}} = 2(1 + \Delta - 2\theta)(1 - p_{\mathsf{LPS}})$; this should equal the probability of an LPS adaptation, namely p_{LPS}. By equating the two adaptation probabilities, we derive an expression for the threshold:

$$\theta = \frac{1+\Delta}{2} - \frac{1}{4} - \frac{p_{\mathsf{LPS}}}{1 - p_{\mathsf{LPS}}}.$$

The Z-Coder adaptation algorithm differs significantly from the adaptation scheme introduced by the Q-Coder and used by the QM-Coder. These coders perform an MPS adaptation whenever encoding or decoding a MPS produces or consumes a code bit. This is similar to using a constant threshold $\theta = 1/2$. An optimally tuned Q-Coder or QM-Coder therefore produces more MPS adaptation events than LPS adaptation events. This is compensated for by careful design of asymmetrical state transition tables. The Z-Coder state tables are free of these constraints. This can be a significant advantage for creating efficient state transition tables in an analytically principled way.

4 Experimental results

We have compared the Z-Coder with three other adaptive binary coders, the QM-Coder, the Q15-Coder (a variant of the Q-Coder that uses 15 bit registers instead of 12-bit), and the Augmented ELS-Coder, based on the ELS-Coder described in [6].

In the main test, various coders including the Z-Coder have been incorporated into the SPM compression system for bilevel images described by Howard [2]. Everything in the compressed file is coded using a binary adaptive coder. The binary contexts included a mixture of skewed and non-skewed probabilities. In tests of compression ratio against the other coders, the Z-Coder did worse than the ELS-Coder, about the same as the QM-Coder, and better than the Q15-Coder. To be fair about it, the differences are all small, in the 1 percent range. The ELS-Coder seems to track non-stationary probabilities better than the Z-Coder, not unexpected since in our current implementation the Z-Coder never leaves the steady state adaptation region once it gets there. A summary of the results of this experiment, including some timing figures, appears in Table 1. The Z-Coder is consistently the fastest of the four coders tested. Detailed results will appear in the full paper.

We also performed two artificial tests. In a test of steady state behavior, coding a long sequence of random bits with fixed probabilities, the Z-Coder performed about as well as the QM-Coder, better than the Q15-Coder, and much better than the ELS-Coder. In a test of early adaptation, coding a long sequence of random bits with fixed probabilities but reinitializing the encoder index every 50 output bits, the Z-Coder did better than the QM-Coder, which was better than the Q15-Coder, which in turn was better than the ELS-Coder.

5 Conclusion

The Z-Coder is a binary statistical coder that can be used wherever adaptive (or non-adaptive) entropy coding of binary symbols is required. Its compression performance is better than that of existing arithmetic coders. Because of its derivation as a generalization of the Golomb coder, it is also extremely fast, and by our derivation we have given a unified treatment of run-length coding and approximate arithmetic coding. In addition, the Z-Coder's probability estimation is more flexible and faster than that of other arithmetic coders.

References

[1] S. W. Golomb, "Run-Length Encodings", *IEEE Trans. Inform. Theory* IT–12 (July 1966), 399–401.

[2] P. G. Howard, "Text Image Compression Using Soft Pattern Matching", *Computer Journal* 40 (1997).

[3] JBIG, "Progressive Bi-level Image Compression", International Standard ISO/IEC 11544, ITU-T Recommendation T.82, 1993.

[4] F. Ono, S. Kino, M. Yoshida & T. Kimura, "Bi-level image coding with Melcode – Comparison of block type code and arithmetic type code", Proc. IEEE Global Telecommunications Conference, Nov. 1989.

[5] W. B. Pennebaker, J. L. Mitchell, G. G. Langdon & R. B. Arps, "An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder", *IBM J. Res. Develop.* 32 (Nov. 1988), 717–726.

[6] W. D. Withers, "The ELS-coder: A Rapid Entropy Coder. ", in *Proc. Data Compression Conference*, J. A. Storer & M. Cohn, eds., Snowbird, Utah, Mar. 25-27, 1997, 475.

[7] I. H. Witten, R. M. Neal & J. G. Cleary, "Arithmetic Coding for Data Compression", *Comm. ACM* 30 (June 1987), 520–540.