An overview of the basic principles of the Q-Coder adaptive binary arithmetic coder

by W. B. Pennebaker
J. L. Mitchell
G. G. Langdon, Jr.
R. B. Arps

The Q-Coder is a new form of adaptive binary arithmetic coding. The binary arithmetic coding part of the technique is derived from the basic concepts introduced by Rissanen, Pasco, and Langdon, but extends the coding conventions to resolve a conflict between optimal software and hardware implementations. In addition, a robust form of probability estimation is used in which the probability estimate is derived solely from the interval renormalizations that are part of the arithmetic coding process. A brief tutorial of arithmetic coding concepts is presented, followed by a discussion of the compatible optimal hardware and software coding structures and the estimation of symbol probabilities from interval renormalization.

1. Introduction

The Q-Coder is an adaptive binary arithmetic coding system which allows different, but compatible, coding conventions to be used in optimal hardware and optimal software implementations. It also incorporates a new probability-estimation technique which provides an extremely simple yet robust mechanism for adaptive estimation of probabilities during the coding process.

This paper presents an overview of the principles of the Q-Coder. A brief discussion of the basic principles of arithmetic coding is presented in Section 2. A discussion of the coding conventions which lead to optimal, compatible hardware and software implementations of arithmetic coding follows in Section 3. In addition, Section 3 introduces some aspects of implementation using fixed-precision arithmetic. Section 4 covers the estimation of probabilities by a new technique which uses only the interval renormalization that is a necessary part of the finite-precision arithmetic coding process. Dynamic probability estimation makes the Q-Coder an adaptive binary arithmetic coder. Section 5 gives some experimental results.

2. Basic principles of binary arithmetic coding[1]

Traditionally, Huffman coding [2] is used to code a sequence of symbols which describes the information being compressed. As an example, **Figure 1** shows a possible Huffman tree for a set of four symbols—w, x, y, and z—with respective probabilities 0.125, 0.125, 0.25, and 0.5. The vertical axis of Figure 1 represents the number line from 0 to 1, which is the probability interval occupied by the four symbols. Each of the four symbols is assigned a subinterval

[1] A much more extensive tutorial on arithmetic coding is found in [1].

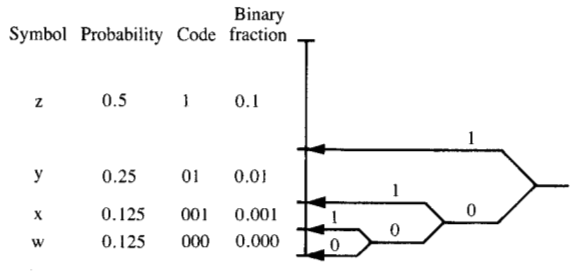


Figure 1

Example of a Huffman coding tree.

of size proportional to the probability estimate of that symbol. If each subinterval is identified by its least or base value, the four symbols are identified respectively by the binary numbers 0.000, 0.001, 0.01, and 0.1. Note that the subinterval size (or probability estimate) determines the length of the code word. *Ideally* this length for some symbol a is given by $\log_2 p_e(a)$, where $p_e(a)$ is the probability estimate for symbol a. For the example in Figure 1, the probabilities have been chosen such that the code lengths are ideal.

The tree in Figure 1 is constructed in a particular way to illustrate a concept which is fundamental to arithmetic coding: The code words, if regarded as binary fractions, are pointers to the particular interval being coded. In Figure 1 the code words point to the base of each interval. The general concept that a code string can be a binary fraction pointing to the subinterval for a particular symbol sequence is due to Shannon [3] and was applied to successive subdivision of the interval by Elias [4]. The idea of *arithmetic coding*, derived by Rissanen [5] from the theory of enumerative coding, was approached by Pasco [6] as the solution to a finite-precision constraint on the interval subdivision methods of Shannon and Elias.

Any decision selecting one symbol from a set of two or more symbols can be decomposed into a sequence of binary decisions. For example, **Figure 2** shows two possible decompositions of the four-symbol choice of Figure 1. From a coding-efficiency point of view there is no difference between the two alternatives, in that the interval size and position on the number line are the same for both. However, from the point of view of computational efficiency, decomposition (a) is better. Fewer computations are required to code the most probable symbol. Thus, although the Huffman coding tree is not required to achieve efficient compression, it remains useful as an approximate guide for minimizing the computational burden.

In general, as coding of each additional binary decision occurs, the precision of the code string must be sufficient to

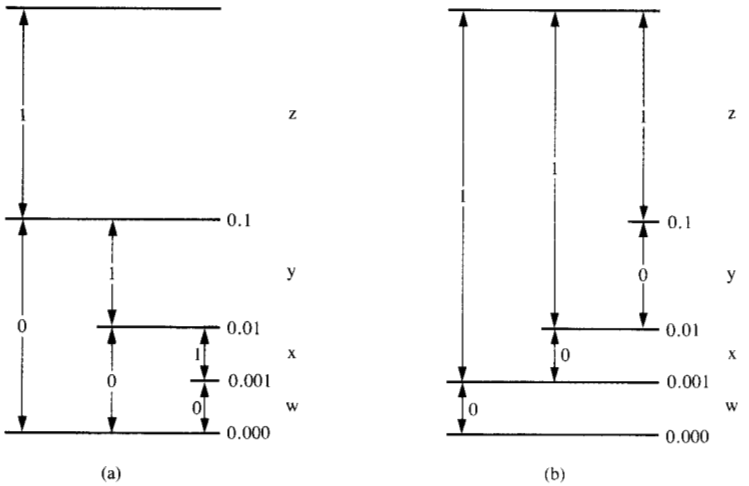


Figure 2

Two examples of decomposition of a four-symbol alphabet into a sequence of three binary decisions: (a) Example corresponding to Figure 1. (b) Example showing the most probable symbol encoded as a succession of three binary decisions.

provide two distinguishable points within the subinterval $p(s)$ allocated to the sequence of symbols, s, which actually occurred. The number of bits, b, required to express the code string is then given by [3]

$$4 > 2^b p(s) \geq 2,$$

which can be rewritten using the left inequality as

$$b < 2 - \log_2(p(s)).$$

After many symbols are coded, $p(s)$ becomes very small, and the code-string length required to express the fraction approaches the ideal value of $-\log_2 p(s)$.

An example of Elias coding is shown in **Figure 3** for the binary decision sequence, M L L M, where M is the more probable symbol (MPS) and L is the less probable symbol (LPS). The interval subdivision in Figure 3 is a generalization of that in Figures 2(a) and (b). The interval subdivision process is defined in terms of a recursion that selects one subinterval as the new current interval. The recursive splitting of the current interval continues until all decisions have been coded. By convention, as in Figure 1, the code string in Figure 3 is defined to point at the base of the current interval. The symbol-ordering convention is adopted from [7], where the MPS probability estimate, P_e, is ordered above the LPS probability estimate, Q_e, in the current interval. The translation of the 0 and 1 symbols into MPS and LPS symbols and the subsequent ordering of the MPS and LPS subintervals is important for optimal arithmetic coding implementations [8].

After each symbol is coded, the probability interval remaining for future coding operations is the subinterval of the symbol just coded. If the more probable symbol M is coded, the interval allocated to the less probable symbol L must be added to the code-string value so that it points to the base of the new interval.

Arithmetic coders such as the Q-Coder avoid the increasing-precision problem of Elias coding by using a fixed-precision arithmetic. Implementation in fixed-precision arithmetic requires that a choice be made for the fixed-precision representation of the interval. Then, a renormalization rule must be devised which maintains the interval size within the bounds allowed by the fixed-precision representation. Both the code string and the interval size must be renormalized identically, or the identification of the code string as a pointer to the current interval will be lost. Efficiency of hardware and software implementations suggests that renormalization be done using a shift-left-logical operation.

The Elias decoder maintains the same current-interval size as the encoder, and performs the same subdivision into subintervals. The decoder simply determines, for each decision, the subinterval to which the code string points. For finite-precision implementations following the coding conventions above, however, the decoder must subtract any

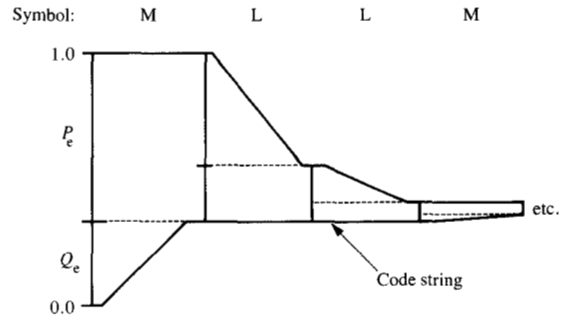


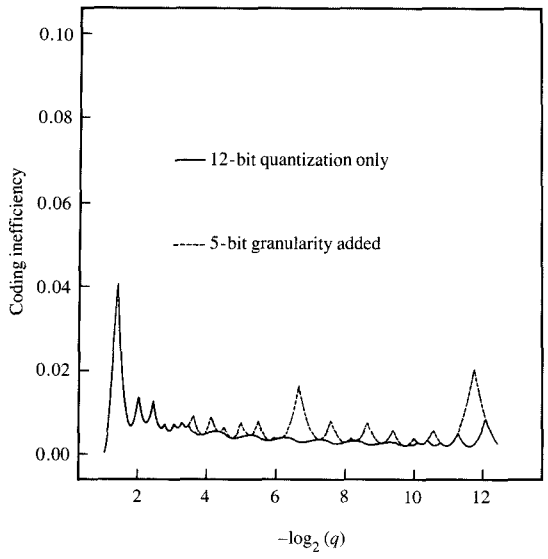
Figure 3

Example of Elias-type coding on string M L L M.

interval added by the encoder, after decoding a given symbol. The code-string remainder will be smaller than the corresponding current-interval measure, since it is a pointer to a particular subinterval within that interval. Renormalization then keeps the precision of the arithmetic operations within the required bounds. Decoder renormalization must be the same as in the encoder.

Another problem to be resolved for implementation in fixed-precision arithmetic is a carry propagation problem. It is possible to generate a code string with a consecutive sequence of 1-bits of arbitrary length. If a bit is added to the least significant bit of this sequence, a carry will propagate until a 0-bit is encountered. Langdon and Rissanen [8] resolved this problem by "bit stuffing." If a sequence of 1-bits of a predefined maximum length is detected, an extra 0-bit is stuffed into the code string. The stuffed 0-bit acts as a carry trap for any carry from future coding operations. The decoder, after detecting this same sequence, removes the stuffed bit, adding any carry contained in it to the code-string remainder. The Q-Coder follows this general scheme, but with the additional constraints that the string of 1-bits is eight bits in length and is byte-aligned.

One final practical problem needs to be resolved. In general, arithmetic coding requires a multiply operation to scale the interval after each coding operation. Generally, multiplication is a costly operation in both hardware and software implementations. An early implementation of adaptive binary arithmetic coding avoided multiplication [8]. However, the Skew Coder [7] uses an even simpler approximation to avoid the multiply; the same approximation is used in the Q-Coder. If renormalizations are used to keep the current interval, A, of order unity, i.e., in the range $1.5 > A \geq 0.75$, the multiplications required to



Coding inefficiency as a function of $\log_2(q)$, where q is the less probable symbol probability. The solid line is for the 12-bit integer representation of Q_e. The dashed line shows the effect of restricting the set of allowed Q_e to the values in Table 1.

subdivide the interval can be approximated as follows:

$$A \times Q_e \simeq Q_e,$$

$$A \times P_e = A \times (1 - Q_e) \simeq A - Q_e.$$

Both the code string and the current interval are periodically renormalized such that the value of A is in the desired range relative to Q_e for the next decision. The true interval is obtained by scaling A by the current renormalization factor. The idea that A should be kept in the range from 0.75 to 1.5 is due to Rissanen.[2]

A calculation of the instantaneous coding inefficiency introduced by this approximation is shown in **Figure 4**. The abscissa is a log scale of the true (not the estimated) LPS probability, q. The ordinate is the coding inefficiency relative to the ideal code length, assuming that the best possible integer value of q is selected. The coding inefficiency is dominated on the left side of the plot by the approximation to the multiply. On the right side of the plot, the coding inefficiency is dominated by quantization effects. The solid line gives the coding efficiency for the 12-bit integer arithmetic precision used in the Q-Coder. When the allowed LPS probability estimates are further restricted to the small subset of 30 values actually used in the Q-Coder, the dashed curve labeled "5-bit granularity added" results.

[2] J. J. Rissanen, IBM Almaden Research Center, San Jose, CA, private communication.

3. Q-Coder hardware and software coding conventions

The description of the arithmetic coder and decoder in the preceding section is precisely that of a hardware-optimized implementation of the arithmetic coder in the Q-Coder. It uses the same hardware optimizations developed for the earlier Skew Coder [7]. A sketch of the unrenormalized code-string development is shown in **Figure 5(a)**, and a sketch of the corresponding decoding sequence is found in **Figure 5(b)**. Note that the coding (and decoding) process requires that both the current code string and the current interval be adjusted on the more probable symbol path. On the less probable symbol path only the current interval must be changed.

The extra operations for the MPS path do not affect hardware speed, in that the reduction in the interval size and the addition to (or subtraction from) the code string can be done in parallel. The illustration of the hardware decoder implementation in **Figure 6(a)** shows this parallelism. However, this organization is not as good for software. Having more operations on the more probable path, as seen in the decoder flowchart of **Figure 7(a)**, can be avoided. Software speed can be enhanced by exchanging the location of subintervals representing the MPS and LPS. As illustrated in **Figure 7(b)**, the instructions on the more probable path are reduced to a minimum and, instead, more instructions are needed on the less probable path. Note that this organization gives slower hardware, in that two serial arithmetic operations must be done on the LPS path [see **Figure 6(b)**]. To decode, first the new MPS subinterval size is calculated, then the result is compared to the code string.

If the choices were limited to the two organizations sketched in Figures 6 and 7, there would be a fundamental conflict between optimal hardware and software implementations. However, there are two ways to resolve this conflict [9]. First, it is possible to invert the code string created for one symbol-ordering convention, and achieve a code string identical to that created with the opposite convention. A second (and simpler) technique uses the same symbol-ordering convention for both hardware and software, but assigns different code-string pointer conventions for hardware and software implementations. The code-string convention shown in Figure 5(a), in which the code string is pointed at the *bottom* of the interval, is used for hardware implementations. However, a different code-string convention, illustrated in **Figure 8(a)**, is used for software. In this software code-string convention the code string is pointed at the *top* of the interval. When the software code-string convention is followed, coding an MPS does not change the code string, while coding an LPS does. Figure 8(a) also shows the relationship between hardware and software code strings. Note that the gap between the two code strings is simply the current interval.

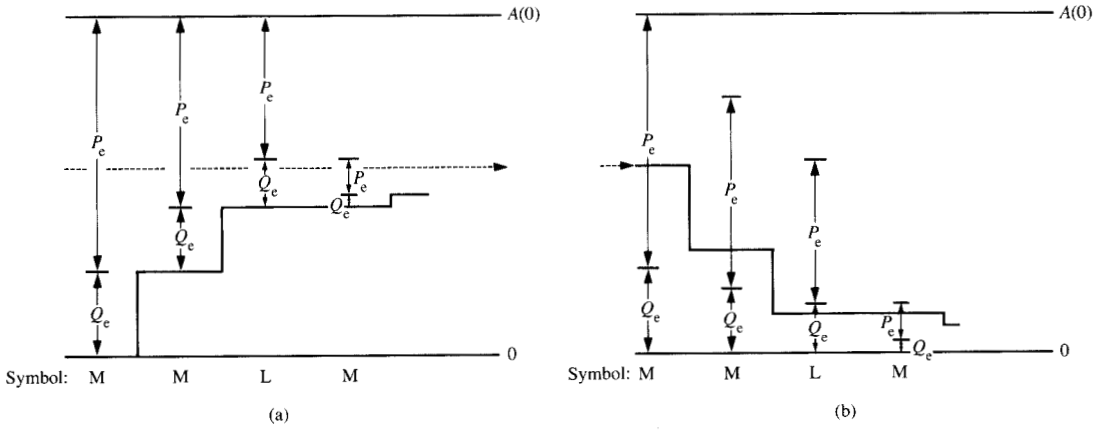


Figure 5

Code-string treatment with hardware-oriented conventions: (a) Hardware encoder code-string development. (b) Hardware decoder code-string remainder.

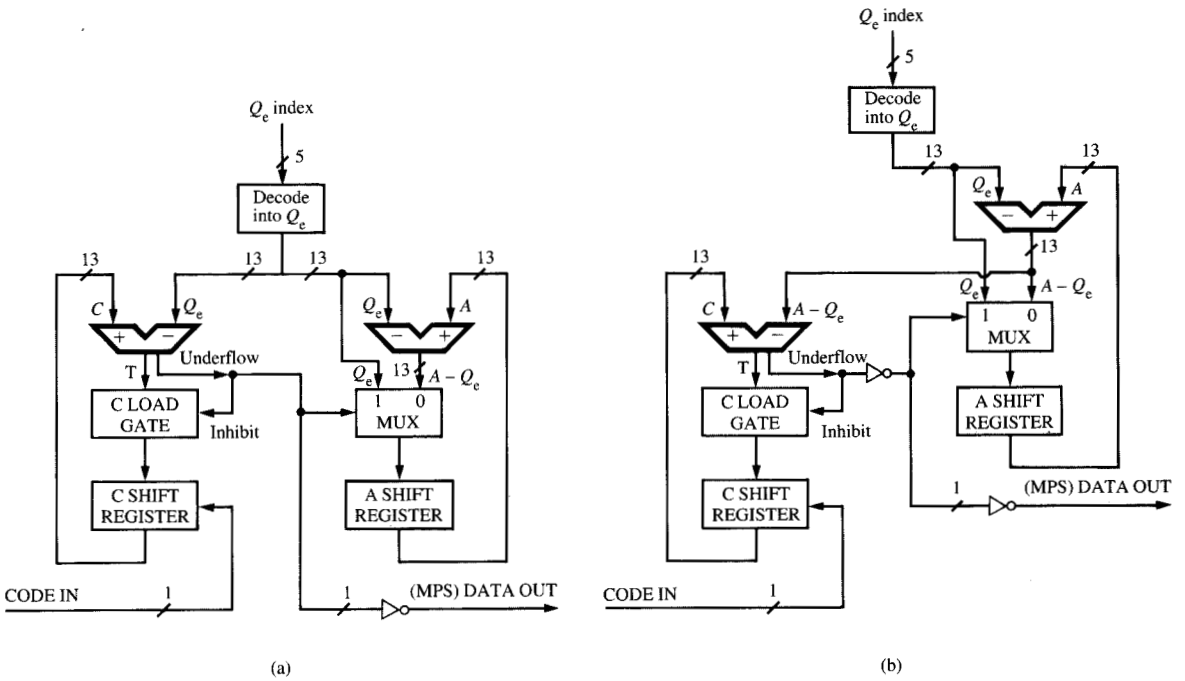


Figure 6

Data path trade-offs: (a) Hardware-optimized MPS and LPS subinterval ordering. (b) Software-optimized MPS and LPS subinterval ordering.

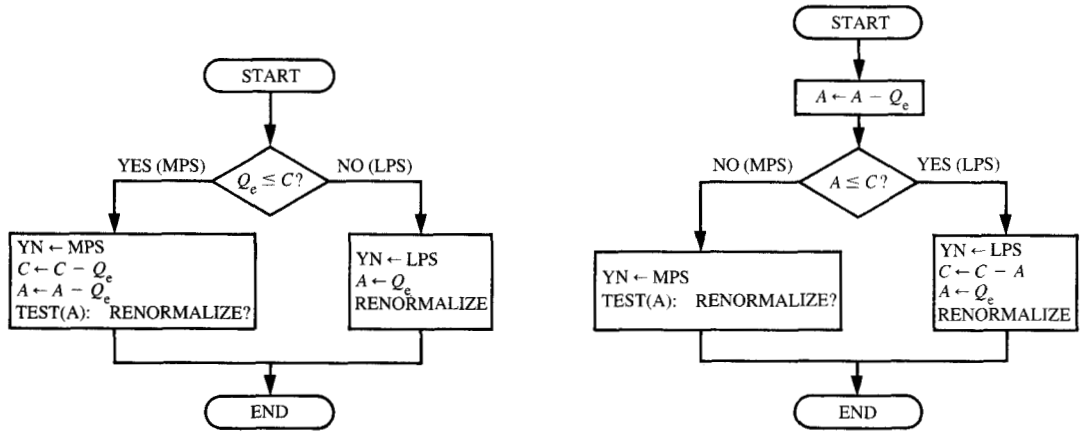


Figure 7

Inner-loop trade-offs: (a) Hardware-optimized MPS and LPS subinterval ordering. (b) Software-optimized MPS and LPS subinterval ordering.

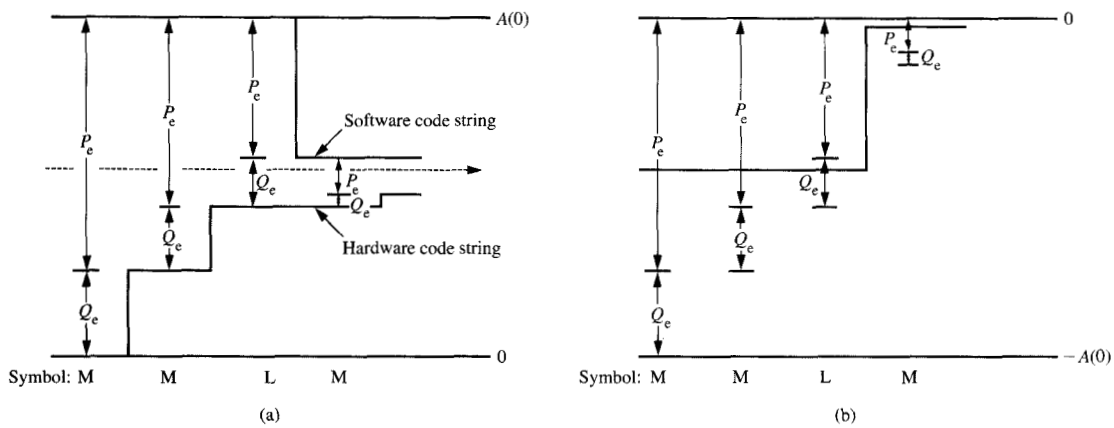


Figure 8

Code-string treatment with software-oriented conventions: (a) Software encoder code-string development. (The hardware code string is shown for comparison.) (b) Software decoder code-string remainder.

The addition operation required when generating a code string following the hardware convention creates a problem with carry propagation. Conversely, the subtraction operation required when following the software convention creates a problem with borrow propagation. Borrow propagation could be blocked by stuffing a 1-bit following a string of 0-bits, but the code strings generated using the two conventions would then be incompatible.

Compatibility (and, in fact, identity) of the hardware and software code strings can be achieved as follows [9]: In both cases the code string is partitioned into two parts, a code *buffer* which contains the high-order bytes of the code string, and a code *register* which contains the low-order bits of the code string. The hardware convention for bit stuffing requires that if R, the byte most recently transferred from the code register to the code buffer, cannot allow the addition of a carry bit (i.e., X′FF′), the leading bit of the next byte must become a stuffed 0-bit.

In software the problem is borrow propagation. The convention is adopted that, by definition, if a borrow is needed from the byte most recently transferred from the code register, it can be taken. The only byte value which cannot supply a borrow is 0. Consider the case where a 0-byte is to be transferred from the code register to the code buffer. Note that the difference between software and hardware code strings is the value in the probability interval register, and this interval value (by definition) must be smaller than the least significant bit of the byte being transferred. Therefore, for the special case where the byte about to be transferred is 0, only two possibilities exist for the software code string relative to the hardware code string:

code buffer	code register	
····, R ,	00, ····	software code string

(a)

····, R ,	00, ····	hardware code string

or

code buffer	code register	
····, R + 1,	00, ····	software code string

(b)

····, R ,	FF, ····	hardware code string

These two cases can be differentiated by comparing the code register with the interval register. If the code register is less than the interval register, case (b) has occurred; otherwise, case (a) has occurred. If case (b), a borrow is taken from the X′00′ byte (converting it to X′FF′) and propagated to $R + 1$ (reducing it to R). For both hardware and software code strings, the X′FF′ byte triggers bit stuffing. However, in hardware a 0-bit is stuffed, while in software a 1-bit is stuffed. As coding proceeds, either a borrow will be subtracted from the software stuff bit or a carry will be added to the hardware stuff bit. The code strings then become identical.

To work with fixed precision, the decoder must "undo" the actions of the encoder. The hardware decoder therefore subtracts any LPS interval the encoder added. Similarly, since for the LPS operation the software encoder subtracts the MPS interval when coding, the software decoder must add this interval. If the software decoder initial conditions simply emulated the hardware decoder, the software code remainder would approach $A(0)$ instead of 0. Since $A(0)$ gets very large relative to the current interval as coding proceeds, a problem with arithmetic precision would quickly occur. Instead, the software decoder remainder is initialized to $-A(0)$ at the start of coding. Adding the MPS interval then moves the remainder toward 0. The decoder comparison is done with a negative remainder and a negative interval, as sketched in **Figure 8(b)**. However, since both remainder and interval approach zero (and are periodically renormalized), there is no problem with precision. Note that the gap between the remainders of the hardware and software decoders is the current interval.

4. Probability estimation

Adaptive arithmetic coding requires that the probability be re-estimated periodically. This very important concept—dynamic probability estimation—was developed in earlier arithmetic coding implementations [7, 8, 10–12]. The current technique, when applied to mixed-context coding, generally falls in the class of so-called approximate counting methods [10, 12, 13].

The probability-estimation technique used in the Q-Coder differs from the earlier techniques in that the estimates are revised only during the interval renormalization required in the arithmetic coder [14]. Estimation only at renormalizations is very important for efficient software implementations. The inner loop of the coder is then minimized. Since each renormalization produces at least one compressed-data bit, the instruction cycles expended on the estimation process are related to the compressed-data code-string length.

By definition, both encoder and decoder renormalize in precisely the same way. Furthermore, renormalization occurs following both the MPS (occasionally) and the LPS (always). The estimation process is implemented as a finite-state machine of 60 states. Half of the states are for an MPS of 1, and the other half are for an MPS of 0. Each state k of the machine has a less probable symbol probability estimate, $Q_e(k)$, associated with it. After the LPS renormalization, the estimate is increased; after the MPS renormalization, the estimate is decreased.

Table 1 gives the set of allowed values for the LPS probability estimate. This particular set of values is the result of a lengthy optimization procedure involving both theoretical modeling [14] and coding of acutal data generated by binary and continuous-tone image-compression models.[3] The values are given as hexadecimal integers, where a scaling is used such that the decimal fraction 0.75 is equivalent to

[3] G. Langdon was the first to show that the estimation process in [14] could be used with such a sparse finite-state machine. He initiated the effort to simplify this algorithm, and contributed significantly to the set of values in Table 1.

Table 1 Q-Coder LPS probability estimate values and associated LPS renormalization index changes.

Q_e (hex)	Q_e (decimal)	dk	Q_e (hex)	Q_e (decimal)	dk
X'0AC1'	0.50409	1	X'0181'	0.07050	2
X'0A81'	0.49237	1	X'0121'	0.05292	2
X'0A01'	0.46893	1	X'00E1'	0.04120	2
X'0901'	0.42206	1	X'00A1'	0.02948	2
X'0701'	0.32831	1	X'0071'	0.02069	2
X'0681'	0.30487	1	X'0059'	0.01630	2
X'0601'	0.28143	1	X'0053'	0.01520	2
X'0501'	0.23456	2	X'0027'	0.00714	2
X'0481'	0.21112	2	X'0017'	0.00421	2
X'0441'	0.19940	2	X'0013'	0.00348	3
X'0381'	0.16425	2	X'000B'	0.00201	2
X'0301'	0.14081	2	X'0007'	0.00128	3
X'02C1'	0.12909	2	X'0005'	0.00092	2
X'0281'	0.11737	2	X'0003'	0.00055	3
X'0241'	0.10565	2	X'0001'	0.00018	2

hexadecimal 1000. This particular scaling provides ease of hardware implementation and leads to a convenient range for the interval register A of hexadecimal 1000 to 1FFF (corresponding to the decimal range of $0.75 \leq A < 1.5$). For simplicity, a convention was adopted that MPS renormalization always shifts the LPS estimate to the next smaller value; the column labeled dk, which gives the change in table position (to larger Q_e) following LPS renormalization, was then derived as part of the optimization procedure.

Each state in the finite-state-machine estimation process must have a unique index. A five-bit index which selects one of 30 Q_e values from Table 1, together with one bit which defines the sense of the MPS, completely describes the current state of the estimator. This six-bit index is all that is required in a hardware implementation.

One particularly efficient software implementation of this finite-state machine uses two tables, one for the MPS renormalization and one for the LPS renormalization. Each table entry is four bytes, where the first two bytes are the new estimate Q_e and the next two bytes are the index to be used at the next renormalization. The estimation process is, therefore, nothing more than an indexed lookup of a four-byte quantity from one of two tables.

An approximate understanding of the principles of operation of this estimation process can be gained from the following argument.[4] Unlike the earlier estimation techniques, which require some estimation mechanism external to the arithmetic coder, the Q-Coder's approximate symbol counting is imbedded in the arithmetic coding process itself. The counter for the MPS is the interval register A. The LPS counter is a one-bit counter which is set by the LPS renormalization.

[4] In developing this explanation, the complications introduced by the exchange of MPS and LPS definitions when the estimate would exceed 0.5 are ignored; the truncation of the finite-state machine at the minimum LPS value is also ignored.

The interval register is decremented by the current LPS probability estimate Q_e following each MPS. After N_{mps} symbols, the MPS renormalization occurs, giving

$$N_{mps} = dA/Q_e ,$$

where dA is the change in the interval register between renormalizations. Define the finite-state-machine organization such that increasing state index, k, corresponds to decreasing Q_e. The finite-state machine is also restricted such that on the MPS renormalization path only a unit change in k is allowed. On the LPS path, however, the change in index, dk, can be 1, 2, or 3. For balance of the estimator, dk MPS renormalizations must occur for each LPS renormalization. Thus, for one LPS event, the number of MPSs must be

$$N_{mps} = [A - 0.75 + 0.75(dk - 1)]/Q_e ,$$

where A is the starting interval after the LPS renormalization. We assume Q_e does not change very much from one state to the next if $dk > 1$. The estimator will be balanced when the true probability q is approximately equal to the estimate

$$q = 1/(N_{mps} + 1) = Q_e .$$

Thus, near $q = 0.5$, $dk = 1$ for a good balance (A is of the order of 1.125 on the average). For $q \ll 0.5$, $dk = 2$ for a good balance. The values for dk in Table 1 show this qualitative behavior. At very small values of Q_e, $dk = 3$ for some entries, but this is a result of optimization in real compression experiments, where the increased estimation rate for $dk = 3$ becomes important.

Figure 9 shows the coding inefficiency for the finite-state machine corresponding to Table 1. The solid curve is the result of an exact modeling of this state machine [9]; the data points are from a Monte Carlo simulation using a real coder and pseudorandom data sets. The coding inefficiency is of the order of 5–6%, the inefficiency being dominated by the granularity of the set of allowed probability estimates (only 30), rather than by any inaccuracy in the estimation process. This granularity represents the best compromise we could find between good coding of statistically stable systems and rapid estimation of dynamically varying systems.

In most data-compression systems, conditioning states or contexts are used with independent probability estimates for each context. In the Q-Coder, a different six-bit index (or, in software, a four-byte storage unit) is kept for each context. However, the renormalization of the interval register is used to generate individual estimates for all of these separate contexts. The approximate analysis above ignores the effects of these independent contexts on the estimation process. An approximate model and experimental data for the influence of multiple contexts are given in [14].

5. Some experimental results

Table 2 lists some experimental results for the Q-Coder for two important classes of image models. The results for gray-

scale compression show that the Q-Coder provides good compression, slightly improving upon the performance achieved with a version of arithmetic coding which used a full multiplication in both arithmetic coding and probability estimation [15].

The results for facsimile image compression using a 7-pel-neighborhood model [8] and the Q-Coder are also summarized. This algorithm for adaptive bilevel image compression (ABIC) has been successfully implemented in a high-speed VLSI chip [16], using the hardware-optimized form of the Q-Coder. Here, the results are much better than can be achieved with the CCITT G-4 facsimile algorithm, and are even better than the stationary entropy for the model. Compressing to better than the stationary entropy is a direct result of the adaptive dynamic probability estimation. The compression achieved with the Skew Coder is almost identical to that listed here for the facsimile documents PTT1–PTT8. However, the Q-Coder fares better on the gray-scale images and also on the statistically unstable binary halftone images.

6. Summary

A brief overview of the Q-Coder has been presented. The same material is covered in much greater depth in the companion papers of this issue of the *IBM Journal of Research and Development* [9, 14, 16, 17] and in previous tutorials [1, 18]. This paper is intended to provide a broad overview of the concepts involved. In addition to the tutorial and the description of the parts of the Q-Coder derived from earlier arithmetic coders such as the Skew Coder, two main points have been summarized—the compatible coding conventions for hardware and software implementations, and the probability-estimation technique.

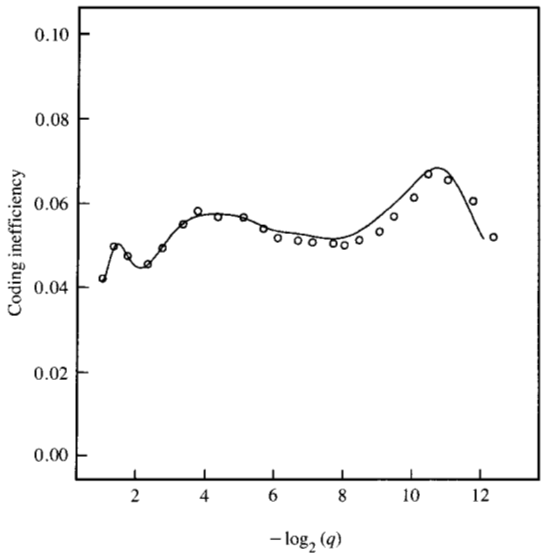


Figure 9

Coding inefficiency of the finite-state machine listed in Table 1 versus negative log of less probable symbol probability q. The solid curve is calculated [13]; the data points are for a Monte Carlo simulation using a real coder and pseudorandom data sets.

Table 2 Compression performance relative to alternative techniques.

	DPCM/Q-Coder (bits)	*Entropy* (bits)	*Reference [15]* (bits)
	Gray-scale image compression		
Total, 13 images	2 120 288	2 074 047	2 138 864
	Bilevel image compression		
	ABIC (bits)	*Entropy* (bits)	*CCITT G-4* (bits)
Total, CCITT 1–8	1 747 008	1 841 921	2 113 128
Total, 6 halftones	757 952		2 333 312

References

1. G. G. Langdon, "An Introduction to Arithmetic Coding," *IBM J. Res. Develop.* **28,** 135 (1984).
2. D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proc. IRE* **40,** 1098 (1952).
3. C. E. Shannon, "A Mathematical Theory of Communication," *Bell Syst. Tech. J.* **27,** 379 (1948).
4. P. Elias, in N. Abramson, *Information Theory and Coding*, McGraw-Hill Book Co., Inc., New York, 1963.
5. J. J. Rissanen, "Generalized Kraft Inequality and Arithmetic Coding," *IBM J. Res. Develop.* **20**, 198 (1976); first published as *IBM Research Report RJ-1591*, June 1975.
6. R. C. Pasco, "Source Coding Algorithms for Fast Data Compression," Ph.D. Thesis, Department of Electrical Engineering, Stanford University, Stanford, CA, May 1976.
7. G. G. Langdon and J. J. Rissanen, "A Simple General Binary Source Code," *IEEE Trans. Info. Theory* **IT-28,** 800 (1982).
8. G. G. Langdon and J. J. Rissanen, "Compression of Black-White Images with Arithmetic Coding," *IEEE Trans. Commun.* **COM-29,** 858 (1981).
9. J. L. Mitchell and and W. B. Pennebaker, "Optimal Hardware and Software Arithmetic Coding Procedures for the Q-Coder," *IBM J. Res. Develop.* **32,** 727 (1988, this issue).
10. D. R. Helman, G. G. Langdon, N. Martin, and S. J. P. Todd, "Statistics Collection for Compression Coding with Randomizing Feature," *IBM Tech. Disclosure Bull.* **24**, 4917 (1982).
11. G. G. Langdon and J. J. Rissanen, "A Double-Adaptive File Compression Algorithm," *IEEE Trans. Commun.* **COM-31,** 1253 (1983).
12. P. Flajolet, "Approximate Counting: A Detailed Analysis," *BIT* **25,** 113 (1985).
13. S. J. P. Todd, G. G. Langdon, D. R. Helman, and N. Martin, "Statistics Collection for Compression Coding with Randomizing Feature," *Research Report RJ-6414*, IBM Almaden Research Center, San Jose, CA, August 1988.
14. W. B. Pennebaker and J. L. Mitchell, "Probability Estimation for the Q-Coder," *IBM J. Res. Develop.* **32,** 737 (1988, this issue).

15. D. Anastassiou, J. L. Mitchell, and W. B. Pennebaker, "Gray-Scale Image Coding for Free-Frame Videoconferencing," *IEEE Trans. Commun.* **COM-34,** 382 (1986).
16. R. Arps, T. K. Truong, D. J. Lu, R. C. Pasco, and T. D. Friedman, "A Multi-Purpose VLSI Chip for Adaptive Data Compression of Bilevel Images," *IBM J. Res. Develop.* **32,** 775 (1988, this issue).
17. J. L. Mitchell and W. B. Pennebaker, "Software Implementations of the Q-Coder," *IBM J. Res. Develop.* **32,** 753 (1988, this issue).
18. G. G. Langdon, W. B. Pennebaker, J. L. Mitchell, R. B. Arps, and J. J. Rissanen, "A Tutorial on the Adaptive Q-Coder," *Research Report RJ-5736,* IBM Almaden Research Center, San Jose, CA, July 1987.

Received March 11, 1988; accepted for publication September 2, 1988

William B. Pennebaker *IBM Research Division, T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Pennebaker is a Research Staff Member at the IBM T. J. Watson Research Center and currently manages a group doing research in areas related to image processing and compression. He joined IBM's Research Division in 1962 and has worked in areas related to low-temperature physics, thin films, display technology, printing technology, and image processing. Dr. Pennebaker has received an Outstanding Contribution Award for work on strontium titanate films, an Outstanding Invention Award for work on silicon nitride films, and an Outstanding Innovation Award for work on image processing and compression. He has received ten IBM Invention Achievement Awards. Dr. Pennebaker received his B.S. in engineering physics from Lehigh University, Bethlehem, Pennyslvania, in 1957, and his Ph.D. in physics from Rutgers University, New Brunswick, New Jersey, in 1962. He is a member of the American Association for the Advancement of Science, the American Institute of Physics, the Institute of Electrical and Electronics Engineers, and the Society for Information Display.

Joan L. Mitchell *IBM Research Division, T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Mitchell graduated from Stanford University with a B.S. in physics in 1969. She received her M.S. and Ph.D. degrees in physics from the University of Illinois at Champaign-Urbana in 1971 and 1974, respectively. She joined the Exploratory Printing Technologies group at the IBM T. J. Watson Research Center immediately after completing her Ph.D., and since 1976 has worked in the field of data compression. Dr. Mitchell received IBM Outstanding Innovation Awards for two-dimensional data compression in 1978, for teleconferencing in 1982, and for the Image View Facility and resistive ribbon thermal transfer printing technology in 1985. She is co-inventor on fifteen patents.

Glen G. Langdon, Jr. *University of California at Santa Cruz, Santa Cruz, California 95064.* Dr. Langdon received the B.S. from Washington State University, Pullman, in 1957, the M.S. from the University of Pittsburgh, Pennsylvania, in 1963, and the Ph.D. from Syracuse University, New York, in 1968, all in electrical engineering. He worked for Westinghouse on instrumentation and data logging from 1961 to 1962 and was an application programmer for the PRODAC computer for process control for most of 1963. In 1963 he joined IBM at the Endicott, New York, development laboratory, where he did logic design on small computers. In 1965 he received an IBM Resident Study Fellowship. On his return from Syracuse University, he was involved in future systems architectures and storage subsystem design. During 1971, 1972, and part of 1973, he was a Visiting Professor at the University of São Paulo, Brazil, where he developed graduate courses on computer design, design automation, microprogramming, operating systems, and MOS technology. The first Brazilian computer, called Patinho Feio (Ugly Duckling), was developed by his students at the University of São Paulo during his stay. He joined the IBM Research laboratory in 1974 to work on distributed systems and later on stand-alone color graphic systems. In 1987 he retired from the Computer Science Department of the IBM Almaden Research Center in San Jose, California, where he worked on data-compression algorithms. He has taught graduate courses on logic and computer design at the University of Santa Clara, California, and is currently a Professor of Computer Engineering at the University of California, Santa Cruz. Dr. Langdon is author of *Logic Design: A Review of Theory and Practice,* an ACM monograph, and coauthor of the Brazilian text *Projecto de Sistemas Digitais.* In 1980 he received the IBM Outstanding Innovation Award for his contributions to arithmetic coding compression techniques; he holds several patents. Dr. Langdon is an accreditation evaluator for computer-related programs for ABET (Accreditation Board for Engineering and Technology), and a CSAB (Computer Sciences Accreditation Board) visitor for computer science programs.

Ronald B. Arps *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120.* Dr. Arps received the B.S., M.S., and Ph.D. degrees in electrical engineering from the California Institute of Technology, Pasadena, in 1960; Oregon State University, Corvallis, in 1963; and Stanford University, Stanford, in 1969, respectively. From 1960 to 1962 he was with the Electrodata Division, Burroughs Co., Pasadena. He has been with IBM since 1963, starting in its Advanced Systems Development Division at Los Gatos and currently in its Research Division at San Jose. His assignments have included exploratory studies on processing and compressing binary images, advanced development of computer peripherals and systems, and research into hardware-optimized adaptive compression and implementation of algorithms in VLSI microsystems. Dr. Arps received a Resident Study Award to Stanford University in 1967–69 and taught as an IBM Visiting Scientist at the Swiss Federal Institute of Technology, Zurich, during 1970–71. During 1977–78, he was on leave as a visiting associate professor at Linkoping University in Sweden. In 1979, Dr. Arps published a chapter entitled "Binary Image Compression" in *Image Transmission Techniques* (Academic Press, New York, 1979). Dr. Arps is currently manager of the VLSI-Oriented Algorithms project at the IBM Almaden Research Center and architect of the ABIC VLSI chip for Adaptive Binary Image Compression. His research interests include adaptive data-compression algorithms as well as image processing, office automation, and computer-aided design of LSI.