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].

Symbol	Probability	Code	Binary fraction
z	0.5	1	0.1
y	0.25	01	0.01
x	0.125	001	0.001
w	0.125	000	0.000

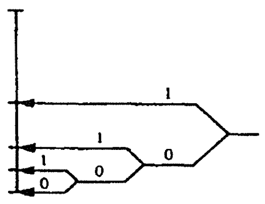


Figure 1

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

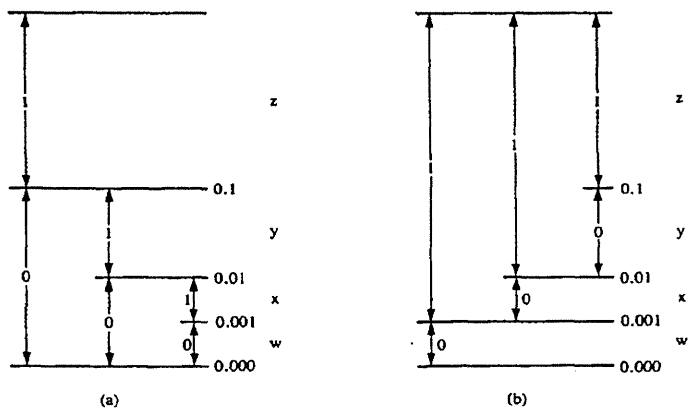


Figure 2

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

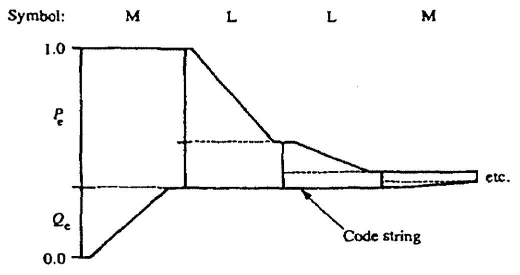


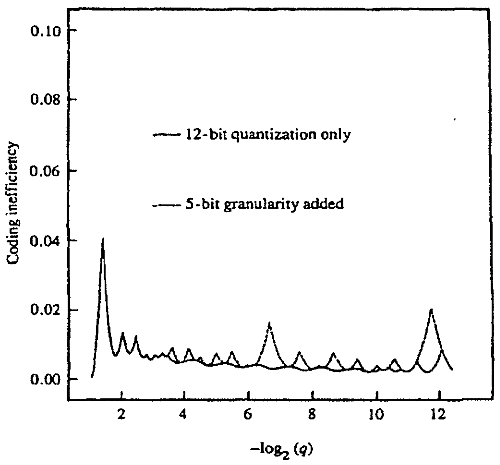
Figure 3

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.

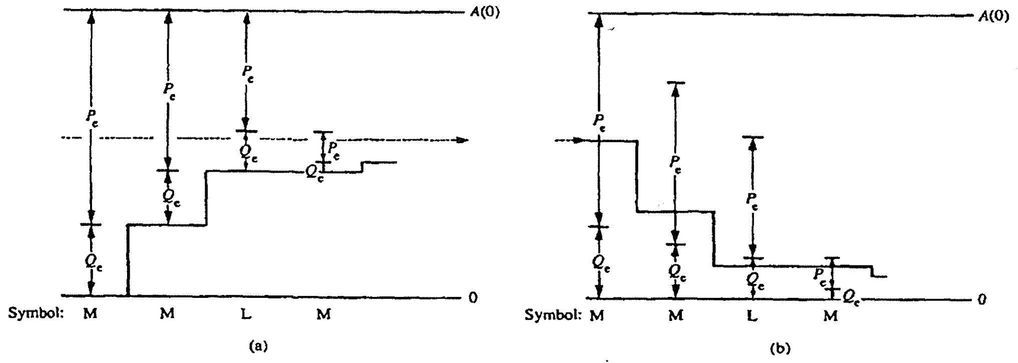


Figure 5

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

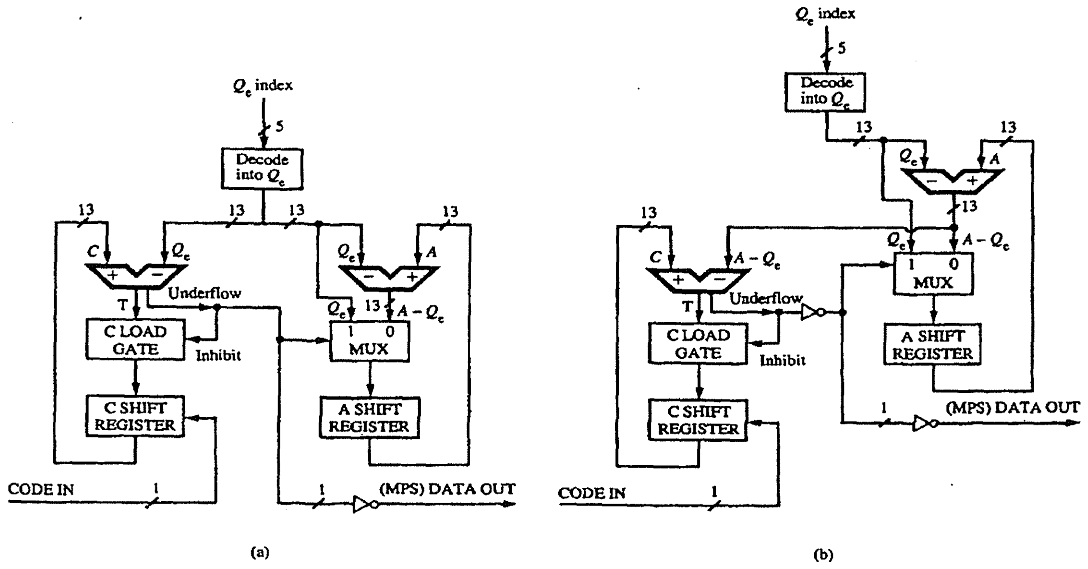


Figure 6

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

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase® Smarter legal research.