A Faster Arithmetic Coder

Barney Stratford*
Oxford University Computing Laboratory

January 13, 2006

Abstract

Arithmetic coding is a highly effective method for data compression. One of the criticisms levelled at it is that it runs fairly slowly. We propose a novel method for arithmetic coding that runs at a significantly higher speed and can be implemented using simple hardware if required.

1 Introduction

Arithmetic coding is a technique for data compression. The origins of the method come from the work of Claude Shannon, although the first recognisable arithmetic coder was developed by Elias some time in 1963. The ideas were further refined until, in 1987, Witten, Neal and Cleary published a practical implementation [13] of the method including C source code. People could then try their own experiments, and the method began to take off.

One of the criticisms often levelled at arithmetic coding is that it tends to run very slowly. Several authors have attempted to remedy this, with limited success. Moffat et al. [6] managed to produce a coder that is faster on systems that have to perform multiplications and divisions in software, although there is a significant cost in the algorithm's compression effectiveness and the speedup completely vanishes when hardware-based arithmetic is available. The new algorithm described in this paper can compete even with hardware-based arithmetic, as it uses only addition, comparison and bit-shift operations. Furthermore, it can easily be implemented in cheap hardware.

2 Mathematical Preliminaries

2.1 Definition. For real numbers a and b such that $a < b$, we use the notation $\langle a, b\rangle$ to represent the interval[1] $\{x \in \mathbb{R} : a \leq x < b\}$. We let $\mathcal{I} = \{\langle a, b\rangle : a, b \in \mathbb{R} \wedge a < b\}$. □

We will need to use some results about the floor function, $x \mapsto \lfloor x \rfloor$. Many published proofs involving arithmetic coding have been made more difficult than necessary because the authors were unaware of these useful characterisations. These results are used extensively throughout this paper. Because the proofs are obvious, they are omitted.

2.2 Lemma. (Rule of Floors) Let $x \in \mathbb{R}$ and $n \in \mathbb{Z}$. Then

- $n \leq \lfloor x \rfloor \iff n \leq x$
- $\lfloor x \rfloor < n \iff x < n$. □

*Supported by a grant from the EPSRC.

[1] The standard notation for such intervals is $[a, b)$. We have avoided this notation owing to the possible confusion between interval delimiters, list brackets and parentheses.

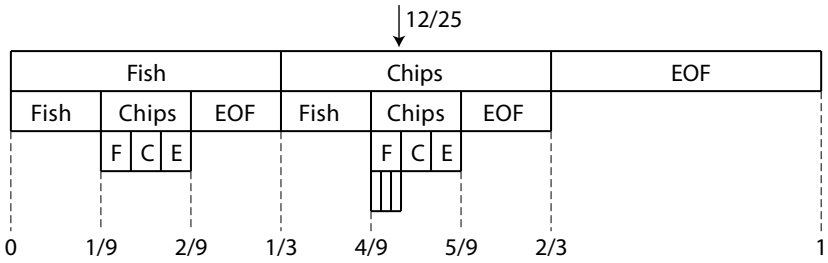

Figure 1: The Arithmetic Coder of Example 3.1.

2.3 Lemma. (Plus 1 Rule) Let $n \in \mathbb{Z}$ and $m \in \mathbb{Z}$. Then

- $n \leq m \iff n < m + 1$
- $n < m \iff n + 1 \leq m$ □

For the most part, functions will be specified using the notation of the Haskell programming language [1]. This is a particularly intuitive language that makes it very easy to carry out formal reasoning about programs. Following the lead of Witten, Neal and Cleary, we translate our faster coder into C to enable the reader to experiment.

3 What is Arithmetic Coding?

A *message* can be considered as a finite list of symbols drawn from some alphabet. In arithmetic coding, each message is represented by an interval of real numbers in $\langle 0, 1\rangle$. Messages containing more information are represented by narrower intervals; the empty message is always encoded as $\langle 0, 1\rangle$.

Every interval that represents some message xs can be partitioned into disjoint subintervals, one for each letter x of our alphabet. The subintervals are then taken to represent the messages $xs +\!\!+ [x]$. ($+\!\!+$ is Haskell's list concatenation operator.) By repeating this partitioning process, we can encode messages of any required length.

To decode, all we need to know is a *single* element e of the interval representing the message. Suppose that xs is any message whose corresponding interval I contains e. The messages $xs +\!\!+ [x]$ will be represented by disjoint subsets of I, only one of which contains e. We can use this fact to find successively longer messages whose corresponding interval contains e. Eventually, we will have recovered the whole message.

The partitions used by these algorithms are fixed at the design stage, so both the encoder and decoder will know how to partition each interval. The particular choice of partition used is known as a *model*, and is discussed further shortly.

3.1 Example. Suppose that our alphabet consists of the three symbols "Fish", "Chips" and "EOF". Every interval that represents a message xs is partitioned into thirds, with the lowest third representing the message $xs +\!\!+ [\text{Fish}]$, the middle third representing the message $xs +\!\!+ [\text{Chips}]$, and the highest third representing the message $xs +\!\!+ [\text{EOF}]$. The EOF symbol occurs at the end of every message we wish to encode, and nowhere else.

Suppose that we need to encode the message $[\text{Fish}, \text{Chips}, \text{EOF}]$. The only message whose encoding we know straight away is the empty message, $[\,]$, which is encoded as $\langle 0, 1\rangle$.

By using our model, we see that $[\,] +\!\!+ [\text{Fish}]$ is encoded as $\langle 0, 1/3\rangle$. The middle third of this, or $\langle 1/9, 2/9\rangle$, is the interval representing $[\text{Fish}] +\!\!+ [\text{Chips}]$. Finally, $[\text{Fish}, \text{Chips}] +\!\!+ [\text{EOF}]$ is represented by the top third of $\langle 1/9, 2/9\rangle$, or $\langle 5/27, 6/27\rangle$, and our complete message has been encoded.

Suppose now that a friend has encoded a message, and we are told that its corresponding interval contains $^{12}/_{25}$. We know straight away that [] is a message whose corresponding interval, $\langle 0,\, 1\rangle$, contains $^{12}/_{25}$.

As before, $\langle 0,\, 1\rangle$ is partitioned into thirds, and the subset containing $^{12}/_{25}$ is $\langle ^1/_3,\, ^2/_3\rangle$. This corresponds to the message [] ++ [Chips]. The middle third of $\langle ^1/_3,\, ^2/_3\rangle$ is $\langle ^4/_9,\, ^5/_9\rangle$, representing the message [Chips] ++ [Chips], and this too contains $^{12}/_{25}$. The message [Chips, Chips] ++ [Fish] is represented by the interval $\langle ^{12}/_{27},\, ^{13}/_{27}\rangle$, which also contains $^{12}/_{25}$; as does $\langle ^{38}/_{81},\, ^{39}/_{81}\rangle$, representing [Chips, Chips, Fish] ++ [EOF]. This message ends in EOF, and so is the same as the message that our friend encoded. We have successfully decoded the message. □

The arithmetic decoder continues to emit symbols until the whole message has been recovered. The question then arises of how the decoder can know when it has decoded a complete message. The commonest method in the literature is to use of a special EOF symbol, as was the case in the preceding example. Another common choice is to record the length of the message and decode precisely that many symbols. Which method is used is largely a matter of taste. The first method may be more useful when the length of the message to be compressed is not known in advance, while the second might be better if one wishes to check in advance that there is sufficient space to decode the entire message. Throughout this paper, we follow the literature and assume that all messages end in the EOF character.

Although the idea behind arithmetic coding is strikingly simple, practical implementations have earned themselves a deserved reputation for being difficult to understand and reason about. Perhaps as a result of this, there have been only a few attempts (eg. [2, 11]) to produce a rigorous proof of the method's correctness. The current work was a result of a formal derivation of the method, although it is presented here in a form that is more accessible to experimentalists.

4 Models

A *model* is an association between symbols and disjoint intervals in $\langle 0,\, 1\rangle$, and is used to calculate the partitions mentioned in the previous section. Suppose that I_{xs} is the interval corresponding to the string xs, and $I_{xs +\!\!+ [x]}$ is the interval corresponding to the message $xs +\!\!+ [x]$. Suppose also that M_x is the interval associated with the letter x in our model. If we stretch the real line so that $\langle 0,\, 1\rangle$ is mapped onto I_{xs}, then M_x will be mapped onto $I_{xs +\!\!+ [x]}$.

4.1 Example. Returning to our previous example, we could take

- $M_{\text{Fish}} = \langle ^0/_3,\, ^1/_3\rangle$
- $M_{\text{Chips}} = \langle ^1/_3,\, ^2/_3\rangle$
- $M_{\text{EOF}} = \langle ^2/_3,\, ^3/_3\rangle$

If we map $\langle 0,\, 1\rangle$ onto I_{xs}, then M_{Fish} will be mapped onto the lowest third of I_{xs}, which is $I_{xs +\!\!+ [\text{Fish}]}$. Similarly, M_{Chips} will be mapped onto the middle third of I_{xs}, which is $I_{xs +\!\!+ [\text{Chips}]}$, and the same goes for the EOF symbol. □

Most models have rational-valued endpoints to their intervals. Therefore we can write $M_x = \langle n_1/d,\, n_2/d\rangle$ for some n_1, n_2 and d. Furthermore, we can assume without loss of generality that d depends only on the model, and not on the letter being encoded. If $I_{xs} = \langle a,\, b\rangle$, then

$$I_{xs +\!\!+ [x]} = \left\langle a + \frac{n_1}{d}(b-a),\, a + \frac{n_2}{d}(b-a) \right\rangle$$

The decoder works in a similar manner. Suppose that e is any member of the interval representing the message, and we have found that $e \in I_{xs}$. We wish to find a letter such that $e \in I_{xs +\!\!+ [x]}$. Note that

$$
\begin{aligned}
& e \in I_{xs +\!\!+ [x]} \\
\Leftrightarrow \quad & \{ \text{Preceding result} \} \\
& e \in \left\langle a + \frac{n_1}{d}(b-a),\, a + \frac{n_2}{d}(b-a) \right\rangle \\
\Leftrightarrow \quad & \{ \text{Set Theory and Arithmetic} \} \\
& n_1 \leq d \times \frac{e-a}{b-a} < n_2 \\
\Leftrightarrow \quad & \{ \text{Rule of Floors} \} \\
& n_1 \leq \left\lfloor d \times \frac{e-a}{b-a} \right\rfloor < n_2
\end{aligned}
$$

Hence, we can find the next letter x by comparing $\lfloor d(e-a)/(b-a) \rfloor$ with the associated values of n_1 and n_2.

Arithmetic coders are not required to use the same model throughout the encoding of a message. Most practical coders adjust the model as encoding proceeds. This enables the coder to closely match the data being encoded, thereby optimising compression effectiveness. All that is required is that the encoder and decoder must both use the same model at the same stage in the process. In the literature, it is said that the encoder and decoder must be "in lockstep".

5 The Coder's Output

We have seen that the output from an arithmetic coder is an interval. Most of the time, this is not useful. Instead, we need to store our encoded data as a sequence of letters from some finite alphabet. For convenience, and without loss of generality, we will assume that the alphabet used is $\{0, 1, \ldots, \mathsf{base} - 1\}$, where base is a fixed constant. We refer to the letters of this alphabet as *digits*.

The analogy with the standard radix notation is intended, since the output of an arithmetic coder will be the digits of some number in the interval I_{xs}. For example, taking $\mathsf{base} = 10$ and $I_{xs} = \langle 0.25,\, 0.5 \rangle$, we could output [3], [4], or even $[2, 5, 0]$. Since we are interested in data compression, it is natural to attempt to minimise the length of the output as much as is convenient.

Suppose that $I_{xs} = \langle a,\, b \rangle$. As a and b approach each other, their initial digits may coincide. For example, taking $\langle a,\, b \rangle = \langle 0.3456,\, 0.3491 \rangle$, the initial digits, 3 and 4, are the same for a and b. Therefore, every number in $\langle a,\, b \rangle$ will begin with the digits $[3, 4]$. We may as well output those digits immediately, rather than waiting for the encoding to be completed. This is known as *incremental encoding*.

We can do better than this, however, by outputting a digit whenever $1/\mathsf{base} \geq b - a$, not just when they have the same initial digits. For example, if $I_{xs} = \langle 0.4998,\, 0.5001 \rangle$, then we can provisionally output the digits $[4, 9, 9]$, although we may need to revise these to $[5, 0, 0]$ at a later stage in the encoding.

This outputting of digits can be made precise. Instead of representing the interval as $I_{xs} = \langle a,\, b \rangle$, we represent it as

$$I_{xs} = \langle \mathsf{digits}\ \mathit{file}\ a,\, \mathsf{digits}\ \mathit{file}\ b \rangle$$

where *file* is a list of digits and

$\mathsf{digits}\ [\,]\ x = x$
$\mathsf{digits}\ ([d] +\!\!+\ ds)\ x = (d + \mathsf{digits}\ ds\ x)/\mathsf{base}$

Whenever a digit is appended to *file*, we must therefore shift a and b one digit to the left to ensure that the value of I_{xs} is unchanged.

Finally, we must consider what happens when the encoding is finished. We can assume that none of the above cases occur, and so we know that

$$\begin{aligned}
& 1/\mathsf{base} < b - a \\
\Leftrightarrow\ & \{\,\text{Arithmetic}\,\} \\
& a \times \mathsf{base} + 1 < b \times \mathsf{base} \\
\Leftrightarrow\ & \{\,\text{Rule of Floors}\,\} \\
& \lfloor a \times \mathsf{base} + 1 \rfloor < b \times \mathsf{base} \\
\Leftrightarrow\ & \{\,\text{Since } a \times \mathsf{base} < \lfloor a \times \mathsf{base} + 1 \rfloor\,\} \\
& a \times \mathsf{base} < \lfloor a \times \mathsf{base} + 1 \rfloor < b \times \mathsf{base} \\
\Leftrightarrow\ & \{\,\text{Arithmetic}\,\} \\
& a < \lfloor a \times \mathsf{base} + 1 \rfloor / \mathsf{base} < b \\
\Rightarrow\ & \{\,\text{Set theory}\,\} \\
& \lfloor a \times \mathsf{base} + 1 \rfloor / \mathsf{base} \in \langle a,\, b \rangle
\end{aligned}$$

Therefore, we need only output one final digit, which is $\lfloor a \times \mathsf{base} + 1 \rfloor$.

6 Integer Arithmetic and Approximate Coding

The work so far requires the use of arbitrary-precision arithmetic. This greatly slows the compression and decompression algorithms, and costs a great deal of memory. Therefore, we would like to introduce some alterations to enable us to use only finite-precision integers to represent intervals. Since we can't represent all possible intervals using such integers (there aren't enough integers) we will have to make some approximations.

Of course, an approximate arithmetic coder will be slightly less effective than the exact version, but this is a small price to pay for practicality.

Let N be some integer that is a multiple of base, and suppose that

$$I_{xs} = \langle \mathsf{digits}\ \mathit{file}\ (a/N),\ \mathsf{digits}\ \mathit{file}\ (b/N) \rangle$$

for some integers a and b. Then we can approximately encode the next symbol by taking

$$\begin{aligned}
I_{xs +\!\!+ [x]} &= \left\langle \mathsf{digits}\ \mathit{file}\ \left(\frac{\lfloor a + (n_1/d)(b-a) \rfloor}{N}\right),\ \mathsf{digits}\ \mathit{file}\ \left(\frac{\lfloor a + (n_2/d)(b-a) \rfloor}{N}\right)\right\rangle \\
&\approx \left\langle \mathsf{digits}\ \mathit{file}\ \left(\frac{a + (n_1/d)(b-a)}{N}\right),\ \mathsf{digits}\ \mathit{file}\ \left(\frac{a + (n_2/d)(b-a)}{N}\right)\right\rangle
\end{aligned}$$

Although we are now encoding approximately, the decoder can still recover the original message since the $I_{xs +\!\!+ [x]}$ intervals are still disjoint subsets of I_{xs}. We will soon see how this can be used to produce an alternative approximation to division that requires significantly less time to evaluate than the obvious one given here.

There is one small problem with approximate arithmetic coding. Suppose that $b - a < d$, $n_1 = 0$ and $n_2 = 1$. Then

$$\left\lfloor a + \frac{n_1}{d}(b-a) \right\rfloor = a = \left\lfloor a + \frac{n_2}{d}(b-a) \right\rfloor$$

and so $I_{xs{+\!\!+}[x]}$ is the empty set. This is not a well-defined state of the arithmetic coder (which requires its intervals to be non-empty) and in practice, a coder would probably crash or spew out infinite rubbish if this were to happen. This is called the *underflow problem*.

The solution is to use incremental encoding to ensure that a and b are always widely-separated, for then this problem can only occur when $d > N/\mathsf{base}$. By choosing N to be sufficiently large and preventing d from growing too much, we can ensure that this never happens.

7 Incremental Decoding

Suppose that $\mathsf{digits}\ \mathit{file}\ (e/N) \in I_{xs}$ for some list *file* and real number e, and we wish to find a letter x such that $\mathsf{digits}\ \mathit{file}\ (e/N) \in I_{xs{+\!\!+}[x]}$. As we did with the exact coder, we can derive a method for finding x rigorously.

$$\mathsf{digits}\ \mathit{file}\ (e/N) \in I_{xs{+\!\!+}[x]}$$

$\Leftrightarrow$ { Definitions of $I_{xs{+\!\!+}[x]}$ and digits }

$$e/N \in \left\langle \left\lfloor a + \frac{n_1}{d}(b-a) \right\rfloor \Big/ N,\ \left\lfloor a + \frac{n_2}{d}(b-a) \right\rfloor \Big/ N \right\rangle$$

$\Leftrightarrow$ { Arithmetic and Set Theory }

$$\left\lfloor a + \frac{n_1}{d}(b-a) \right\rfloor \le e < \left\lfloor a + \frac{n_2}{d}(b-a) \right\rfloor$$

$\Leftrightarrow$ { Rule of Floors }

$$\left\lfloor a + \frac{n_1}{d}(b-a) \right\rfloor \le \lfloor e \rfloor < \left\lfloor a + \frac{n_2}{d}(b-a) \right\rfloor$$

$\Leftrightarrow$ { Plus 1 Rule }

$$\left\lfloor a + \frac{n_1}{d}(b-a) \right\rfloor < \lfloor e \rfloor + 1 \le \left\lfloor a + \frac{n_2}{d}(b-a) \right\rfloor$$

$\Leftrightarrow$ { Rule of Floors }

$$a + \frac{n_1}{d}(b-a) < \lfloor e \rfloor + 1 \le a + \frac{n_2}{d}(b-a)$$

$\Leftrightarrow$ { Arithmetic }

$$n_1(b-a) < d(\lfloor e \rfloor - a + 1) \le n_2(b-a)$$

$\Leftrightarrow$ { Plus 1 Rule }

$$n_1(b-a) \le d(\lfloor e \rfloor - a + 1) - 1 < n_2(b-a)$$

$\Leftrightarrow$ { Arithmetic }

$$n_1 \le \frac{d(\lfloor e \rfloor - a + 1) - 1}{b-a} < n_2$$

$\Leftrightarrow$ { Rule of Floors }

$$n_1 \le \left\lfloor \frac{d(\lfloor e \rfloor - a + 1) - 1}{b-a} \right\rfloor < n_2$$

This result was first proved by Witten, Neal and Cleary [13], except their proof was rather more complicated because they didn't use the Rule of Floors, preferring to reason directly from the definition of the floor function.

The use of integer arithmetic in the encoder has made it possible for us to use incremental decoding, since we only need to use the integer part of e to decode each symbol. The fractional part of e (representing the remainder of the message that has yet to be decoded) can be left in the compressed data stream until it is needed.

Since the value of I_{xs} and digit *file* (e/N) both depend on *file*, it will be necessary to update the value of e whenever *file* gets updated. As before, we simply shift digits to the left. This time, however, the digit immediately to the right of the decimal point gets shifted into the integer part of e, which is equivalent to reading the next digit from the compressed data stream.

8 Approximate Interval Shrinking

In the Witten, Neal and Cleary coder, we require to calculate $a + \lfloor n_1(b-a)/d \rfloor$ and $a + \lfloor n_2(b-a)/d \rfloor$, where $0 \le n_1 < n_2 \le d$ and, to prevent the underflow problem rearing its ugly head, $d \le b - a$. We give an approximation to

$$\left\lfloor \frac{n_i(b-a)}{d} \right\rfloor \quad \text{when} \quad 0 \le n_i \le d \le b-a$$

where $n_i = n_1$ or n_2. Previous work on such approximations can be found in [3, 4, 8, 9]. For brevity, we let $w = b - a$.

Let $k \in \mathbb{N}$ be such that $\frac{1}{2}w < 2^k d \le w$, that is $k = \lfloor \log_2(w/d) \rfloor$. Then

$$\left\lfloor \frac{n_i w}{d} \right\rfloor = \left\lfloor \frac{(2^k n_i)w}{(2^k d)} \right\rfloor$$

and so we can assume without loss of generality that $\frac{1}{2}w < d \le w$. Now,

$$\begin{aligned}
& \tfrac{1}{2}w < d \le w \\
\Leftrightarrow \quad & \{ \text{Arithmetic} \} \\
& 1 \le \frac{w}{d} < 2 \\
\Rightarrow \quad & \{ \text{Since } 0 \le x - \lfloor x \rfloor < 1 \text{ for all } x \} \\
& 1 \le \frac{n_i w}{d} - \left\lfloor \frac{n_i w}{d} \right\rfloor + \frac{w}{d} < 3 \\
\Leftrightarrow \quad & \{ \text{Arithmetic} \} \\
& 1 + \left\lfloor \frac{n_i w}{d} \right\rfloor \le \frac{(n_i+1)w}{d} < 3 + \left\lfloor \frac{n_i w}{d} \right\rfloor \\
\Leftrightarrow \quad & \{ \text{Rule of Floors} \} \\
& 1 + \left\lfloor \frac{n_i w}{d} \right\rfloor \le \left\lfloor \frac{(n_i+1)w}{d} \right\rfloor < 3 + \left\lfloor \frac{nw}{d} \right\rfloor \\
\Leftrightarrow \quad & \{ \text{Arithmetic} \} \\
& 1 \le \left\lfloor \frac{(n_i+1)w}{d} \right\rfloor - \left\lfloor \frac{n_i w}{d} \right\rfloor < 3 \\
\Leftrightarrow \quad & \{ \text{Arithmetic} \} \\
& \left\lfloor \frac{(n_i+1)w}{d} \right\rfloor - \left\lfloor \frac{n_i w}{d} \right\rfloor = 1 \text{ or } 2
\end{aligned}$$

Observe that

$$\sum_{n_i=0}^{d-1} \left(\left\lfloor \frac{(n_i+1)w}{d} \right\rfloor - \left\lfloor \frac{n_i w}{d} \right\rfloor \right) = \left\lfloor \frac{dw}{d} \right\rfloor - \left\lfloor \frac{0w}{d} \right\rfloor = w.$$

But each of the terms in this sum evaluates to either 1 or 2. Suppose that p of them evaluate to 1 and q of them evaluate to 2. There are d terms in total. Therefore,

$$\begin{aligned} 1p + 2q &= w \text{ (writing the above sum in a different way) and} \\ p + q &= d \text{ (since the total number of terms is } d). \end{aligned}$$

This pair of equations has solution

$$\begin{aligned} p &= 2d - w, \text{ and} \\ q &= w - d. \end{aligned}$$

We therefore make the following approximation:

$$\left\lfloor \frac{(n_i+1)w}{d} \right\rfloor - \left\lfloor \frac{n_i w}{d} \right\rfloor \approx \begin{cases} 1 & \text{if } n_i \geq w - d \\ 2 & \text{if } n_i < w - d \end{cases}$$

Note that there are $p = 2d - w$ values of n_i for which this approximation is 1, and $q = w - d$ for which it is 2. Furthermore, we can easily see that

$$\left\lfloor \frac{n_i w}{d} \right\rfloor \approx \begin{cases} n_i + w - d & \text{if } n_i \geq w - d \\ 2n_i & \text{if } n_i < w - d \end{cases}$$

Guided by this reasoning, we make the following definition:

$$\begin{array}{l} \mathsf{approx}\ w\ n_i\ d \\ \quad \mid 2^k n_i \geq w - 2^k d \quad = 2^k n_i + w - 2^k d \\ \quad \mid 2^k n_i < w - 2^k d \quad = 2(2^k n_i) \\ \quad\quad \textbf{where } k = \lfloor \log_2(w/d) \rfloor \end{array}$$

for then

$$\left\lfloor \frac{n_i w}{d} \right\rfloor \approx \mathsf{approx}\ w\ n_i\ d.$$

For the purposes of decoding, we will need to be able to reverse this approximation. Suppose that, for some n,

$$\mathsf{approx}\ w\ n_i\ d \leq n < \mathsf{approx}\ w\ (n_i + 1)\ d.$$

What is the value of n_i? To answer this question, we would like to define another function, $\mathsf{unapprox}$, so that $n_i = \mathsf{unapprox}\ w\ n\ d$ or, to put it another way,

$$n_i \leq \mathsf{unapprox}\ w\ n\ d < n_i + 1.$$

This is equivalent to saying that

$$\mathsf{approx}\ w\ n_i\ d \leq n \iff n_i \leq \mathsf{unapprox}\ w\ n\ d.$$

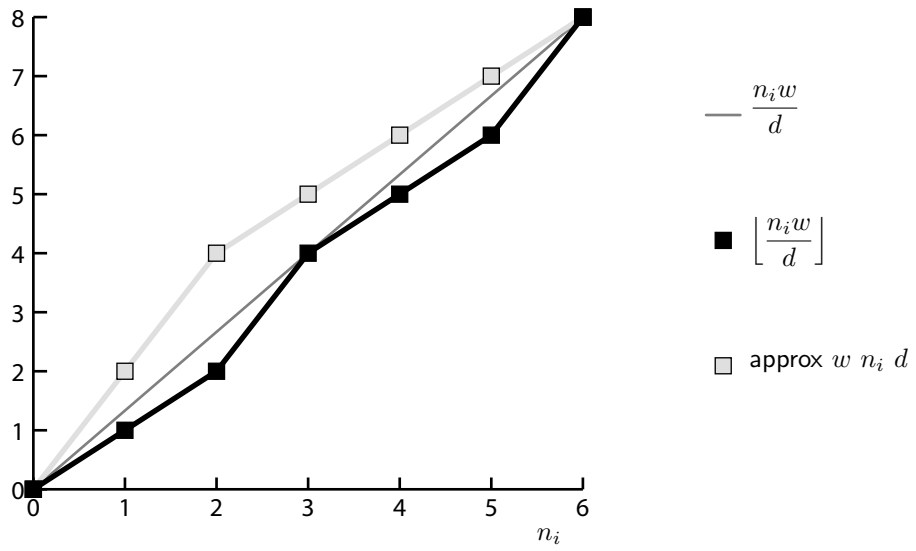

Figure 2: Integer Division and Approximate Division with $d = 6$ and $w = 8$

A little bit of easy work gives us that

$$\begin{array}{l}\mathsf{unapprox}\ w\ n\ d \\ \quad \mid n \geq 2(w - 2^k d) \quad = \lfloor 2^{-k}(n - w + 2^k d) \rfloor \\ \quad \mid n < 2(w - 2^k d) \quad = \lfloor 2^{-k}(1/2 n) \rfloor \\ \quad\ \textbf{where}\ k = \lfloor \log_2(w/d) \rfloor\end{array}$$

Since the approximation requires at most a small constant number of additions, subtractions and bit-shifts, it is clear that it will be much quicker to evaluate than the original function, which used multiplication and division. In effect, we have approximated integer division by putting a kink into the graph: see Figure 2. This approximation also lends itself to hardware implementations [3] of arithmetic coding, since multiplication and division require a significantly larger quantity of silicon than these simple operations. Not only is our approximation reversible, but the operations required to reverse it are also extremely cheap. We can therefore make a very fast decoder as well.

The approximation's cost is that the compression effectiveness is reduced by about 1%. This compares favourably with the cost of 5% that Moffat was prepared to accept in his improved coder [6].

Using this approximation to division, we can update our coder. As before, suppose that

$$I_{xs} = \langle \mathsf{digits}\ \mathit{file}\ (a/N), \mathsf{digits}\ \mathit{file}\ (b/N) \rangle$$

for some integers a and b. Then we take

$$I_{xs +\!\!+ [x]} = \left\langle \mathsf{digits}\ \mathit{file}\ \left(\frac{a + \mathsf{approx}\ (b-a)\ n_1\ d}{N}\right), \mathsf{digits}\ \mathit{file}\ \left(\frac{a + \mathsf{approx}\ (b-a)\ n_2\ d}{N}\right) \right\rangle$$

Suppose we know that $\mathsf{digits}\ \mathit{file}\ (e/N) \in I_{xs}$, and we wish to find the next letter in the encoded data stream. Then

$$\begin{array}{ll} & \mathsf{digits}\ \mathit{file}\ (e/N) \in I_{xs +\!\!+ [x]} \\ \Leftrightarrow & \{\, \text{Definition of } \mathsf{digits} \text{ and } I_{xs +\!\!+ [x]} \,\} \\ & \dfrac{e}{N} \in \left\langle \dfrac{a + \mathsf{approx}\ (b-a)\ n_1\ d}{N}, \dfrac{a + \mathsf{approx}\ (b-a)\ n_2\ d}{N} \right\rangle \end{array}$$

Encoding	Encoded Size	Encoding Time	Decoding Time
None	3,251,493 bytes		
Witten	1,795,707 bytes	4.181 s	4.378 s
Moffat	2,180,261 bytes	3.583 s	4.712 s
Fast	1,818,799 bytes	1.335 s	1.710 s

Table 1: Comparison of the Different Arithmetic Coders

$\Leftrightarrow$ { Arithmetic and Set Theory }

$$\mathsf{approx}\ (b-a)\ n_1\ d \le e-a < \mathsf{approx}\ (b-a)\ n_2\ d$$

$\Leftrightarrow$ { Rule of Floors }

$$\mathsf{approx}\ (b-a)\ n_1\ d \le \lfloor e-a \rfloor < \mathsf{approx}\ (b-a)\ n_2\ d$$

$\Leftrightarrow$ { Property of approx and unapprox }

$$n_1 \le \mathsf{unapprox}\ (b-a)\ \lfloor e-a \rfloor\ d < n_2$$

We can therefore use the unapprox function to help recover the next letter.

9 Conclusion

Table 1 shows the performances of the various arithmetic coders discussed so far, measured on a SunBlade 100 machine with a 500MHz Sparc 9 processor running Solaris. The test file is the concatenation of the files in the Calgary Corpus [12]. The fast coder (given in C in the appendix) produces an output file that is about 101% of the output of Witten's coder, while encoding and decoding take only 32% and 39% of the time respectively. The model used in all cases was identical to the adaptive model given in Witten's original paper [13]. It is interesting to note that the time required for modelling in the encoder was 0.814 seconds, meaning that the faster encoder runs at about 6.5 times the speed when modelling time is discounted.

A A C Implementation of the Fast Arithmetic Coder

A.1 Introduction

The arithmetic coding program given here is based on the fast coder discussed in previous sections. It is optimised for readability as well as speed, and is intended to be used as a general-purpose arithmetic coding library—experience has shown that it's very hard to get arithmetic coding algorithms to run correctly, so it is much better to reuse tried and tested code!

The encoded data produced by this code contains a few dummy bytes in addition to the useful data. This is to make the program more flexible in practical applications. In particular, the encoder and decoder will both leave the file pointer in the same location after compression / decompression has been completed. Therefore, one can simply append further data to the file without having to re-position the file pointer after decoding. This convenience was not present in other arithmetic coders.

Before using this code, the reader should be aware that arithmetic coding was covered by a number of patents. Many of these have now expired, since the method is over 20 years old; others may still be in force. You need to perform your own checks. None of the novel ideas in this paper are covered by additional patents, so you can use them as you wish.

These files can be downloaded from

`http://barney_stratford.fastmail.fm/coder.zip`

which also includes examples of how to use the library.

A.2 ac.h

The programmers' interface for this arithmetic coder is fairly self-explanatory. The denominator used by `AC_Encode` and `AC_Decode` is that passed to the *previous* call to those functions. Initially, the denominator is set to 1, so you will almost certainly want to say `AC_Encode (0, 1, new_denom)` before beginning the encoding of the file, and equivalently `value = AC_Decode (0, 1, new_denom)` before decoding.

In order to ensure correct operation, both the compressor and decompressor must make matching calls to `AC_Encode` and `AC_Decode`. If the compression software makes a call to `AC_Encode (low, high, new_denom)` then the decompressor must make a matching call to `AC_Decode (low, high, new_denom)`.

```
/*
 *  ac.h
 *  By Barney Stratford
 *  Version 12, 13th January 2005
 */

#ifndef AC_H
#define AC_H

#define MAX_DENOM 65536

void AC_InitialiseEncoder (FILE *compressed);
void AC_InitialiseDecoder (FILE *compressed);

void AC_Encode (register long low, register long high, long new_denom);
long AC_Decode (register long low, register long high, long new_denom);

void AC_FinaliseEncoder (void);
void AC_FinaliseDecoder (void);

#endif
```

A.3 encode.c

Throughout this file, the `left` variable contains the value of a and `width` contains $b-a$. The value of k in the definition of `approx` is contained in `bits`, while `kink` contains the location of the kink in the graph of the approximation to integer division.

The `denom` variable contains the denominator to be use next time `AC_Encode` is called. It is not used directly, other than in the assertions at the beginning of `AC_Encode`.

```
/*
 *  encode.c
 *  By Barney Stratford
 *  Version 12, 13th January 2005
 */
```

```
#include <assert.h>
#include <stdio.h>
#include "ac.h"

#define BITS 16

static FILE *file = NULL;
static long left, kink, bits;
static long denom;
extern const int ac_shift[];

static void write_byte (long byte);



void AC_InitialiseEncoder (FILE *compressed)
{
#ifdef DEBUG
    assert (compressed != NULL);
    assert (file == NULL);
#endif

    /* Initialise state */
    file = compressed;
    left = 0;
    kink = 0;
    bits = BITS + 8;
    denom = 1;
}

void AC_Encode (register long low, register long high, long new_denom)
{
    register long width;

#ifdef DEBUG
    assert (1 <= new_denom);
    assert (new_denom <= MAX_DENOM);
    assert (0 <= low);
    assert (low < high);
    assert (high <= denom);
    assert (file != NULL);
#endif

    /* Shrink the interval */
    low <<= bits;
    high <<= bits;
    if (low < kink) low <<= 1;
    else low += kink;
    if (high < kink) high <<= 1;
    else high += kink;
    left += low;
    width = high - low;
```

```
    /* If necessary, write out some bytes */
    while (width <= 1L << BITS)
    {
        write_byte (left >> BITS);
        left = (left & ((1L << BITS) - 1)) << 8;
        width <<= 8;
    }

    /* Adjust the denominator */
    if (new_denom <= 256)
        bits = BITS + ac_shift[new_denom] - ac_shift[width >> BITS];
    else
        bits = BITS - 8 + ac_shift[new_denom >> 8] - ac_shift[width >> BITS];
    if (new_denom << bits > width) bits--;
    kink = width - (new_denom << bits);
    denom = new_denom;
}

void AC_FinaliseEncoder (void)
{
#ifdef DEBUG
    assert (file != NULL);
#endif

    /* Write out any remaining data in the buffer */
    write_byte (left >> BITS);
    write_byte (256);
    write_byte (0);
    write_byte (0);
    file = NULL;
}



static void write_byte (long byte)
{
    static int buffer = 0, carry = 0;

    if (byte < 255)
    {
        putc (buffer, file);
        while (carry)
        {
            putc (255, file);
            carry--;
        }
        buffer = byte;
    }
```

```
    else if (byte > 255)
    {
        putc (buffer + 1, file);
        while (carry)
        {
            putc (0, file);
            carry--;
        }
        buffer = byte - 256;
    }
    else carry++;
}
```

A.4 decode.c

All the variables in the decoder are analogous to those in the encoder, with the exception of `left`, which now contains $\lfloor e-a \rfloor$. The fractional part of e is stored in the incoming encoded data-stream.

The `value` variable is used only to facilitate the assertion-checking that occurs at the beginning of `AC_Decode`.

```
/*
 *  decode.c
 *  By Barney Stratford
 *  Version 12, 13th January 2005
 */

#include <stdio.h>
#include <assert.h>
#include "ac.h"

#define BITS 16

static FILE *file = NULL;
static long left, kink, bits;
static long denom, value;
extern const int ac_shift[];

static long read_byte (void);



void AC_InitialiseDecoder (FILE *compressed)
{
#ifdef DEBUG
    assert (compressed != NULL);
    assert (file == NULL);
#endif
```

```
    /* Initialise state */
    file = compressed;
    read_byte ();
    left = (read_byte () << 16) + (read_byte () << 8) + read_byte ();
    kink = 0;
    bits = BITS + 8;
    denom = 1;
    value = 0;
}

long AC_Decode (register long low, register long high, long new_denom)
{
    register long width;

#ifdef DEBUG
    assert (1 <= new_denom);
    assert (new_denom <= MAX_DENOM);
    assert (0 <= low);
    assert (low <= value);
    assert (value < high);
    assert (high <= denom);
    assert (file != NULL);
#endif


    /* Shrink the interval */
    low <<= bits;
    high <<= bits;
    if (low < kink) low <<= 1;
    else low += kink;
    if (high < kink) high <<= 1;
    else high += kink;
    left -= low;
    width = high - low;

    /* If necessary, read in some bytes */
    while (width <= 1L << BITS)
    {
        left = (left << 8) + read_byte ();
        width <<= 8;
    }

    /* Adjust the denominator */
    if (new_denom <= 256)
        bits = BITS + ac_shift[new_denom] - ac_shift[width >> BITS];
    else
        bits = BITS - 8 + ac_shift[new_denom >> 8] - ac_shift[width >> BITS];
    if (new_denom << bits > width) bits--;
    kink = width - (new_denom << bits);
    denom = new_denom;
```

```
    /* Find the next symbol */
    if (left < kink << 1) return (value = (left >> 1) >> bits);
    else return (value = (left - kink) >> bits);
}

void AC_FinaliseDecoder (void)
{
#ifdef DEBUG
    assert (file != NULL);
#endif

    file = NULL;
}

static long read_byte (void)
{
    int byte = getc (file);
    return (byte == EOF ? 0 : byte);
}
```

A.5 lookup.c

This lookup table satisfies `256 <= n << ac_shift[n] && n << ac_shift[n] < 512` whenever `0 < n && n <= 256`. We use a lookup table for this task because it's faster than performing the calculation.

```
/*
 *  lookup.c
 *  By Barney Stratford
 *  Version 12, 13th January 2005
 */

const int ac_shift[] =
{
    9, 8, 7, 7, 6, 6, 6, 6, 5, 5, 5, 5, 5, 5, 5, 5,
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0
};
```

References

[1] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition, 1998.

[2] Richard Bird and Jeremy Gibbons. Arithmetic coding with folds and unfolds. In Johan Jeuring and Simon Peyton Jones, editors, Advanced Functional Programming 4, volume 2638 of LNCS. Springer-Verlag, 2003.

[3] Gennady Feygin, P. Glenn Gulak, and Paul Chow. Minimizing error and VLSI complexity in the multiplication free approximation of arithmetic coding. In *Proceedings of the Data Compression Conference*, March 1993.

[4] Hannes Hassler and Naofumi Takagi. Function evaluation by table look-up and addition. Technical Report KUIS-95-0003, Department of Information Science, Kyoto University, January 1995.

[5] Glen G. Langdon, Jr. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2):135–149, March 1984.

[6] Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, July 1998.

[7] Mark Nelson. Arithmetic coding + statistical modelling = data compression. *Dr Dobb's Journal*, February 1991.

[8] Stuart F. Oberman and Michael J. Flynn. Fast IEEE rounding for division by functional iteration. Technical Report CSL-TR-96-700, Computer Systems Laboratory, Stanford University, July 1996.

[9] Stuart F. Oberman and Michael J. Flynn. Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8):833–854, August 1997.

[10] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423,623–656, July and October 1948.

[11] Barney Stratford. *A Formal Treatment of Lossless Data Compression Algorithms*. PhD thesis, Oxford University Computing Laboratory, May 2005.

[12] Ian H. Witten and Timothy C. Bell. The Calgary Corpus. Available from `ftp://ftp.cpcs.ucalgary.ca/pub/projects/text.compression.corpus`.

[13] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.