McGill University
School of Computer Science
Sable Research Group

Arithmetic Coding revealed [1]

A guided tour from theory to praxis

Sable Technical Report No. 2007-5

Eric Bodden
Sable Research Group
McGill University
Montréal, Québec, Canada

Malte Clasen
Zuse Institut Berlin
Berlin, Germany

Joachim Kneis
Computer Science Department
RWTH Aachen University
Aachen, Germany

May 25, 2007

www.sable.mcgill.ca

[1] This survey was originally established as contribution to the Seminar Data Compression WS 2001/2002 at the RWTH Aachen University, under supervision of Ralf Schlüter.

Contents

List of Tables

List of Figures

Abstract

This document is an updated and translated version of the German paper *Arithmetische Kodierung* [BCK02] from 2002. It tries to be a comprehensive guide to the art of arithmetic coding.

First we give an introduction to the mathematic principles involved. These build the foundation for chapter 3, where we describe the encoding and decoding algorithms for different numerical systems. Here we also mention various problems one can come across as well as solutions for those. This is followed by a proof of uniqueness and an estimation of the efficiency of the algorithm. In the end we briefly mention different kinds of statistical models, which are used to actually gain compression through the encoding. Throughout this paper we occasionally make some comparisons to the related Huffman encoding algorithm. Though, some rudimentary knowledge about Huffman encoding should suffice for the reader to follow the line of reasoning.

This paper is mainly based on [Say00] and [BCW90]. On the latter we base our implementation which is included in the appendix as full C++ source code. We also make use of parts of this code during some of our examples. The mathematical model we use, however, is strongly based on [Say00] and [Fan61]. Also we employ the well-known Shannon-Theorem [WS49], which proofs the entropy to be the bound of feasible lossless compression.

1 Motivation and History

In comparison to the well-known Huffman Coding algorithm, Arithmetic Coding overcomes the constraint that the symbol to be encoded has to be coded by a whole number of bits. This leads to higher efficiency and a better compression ratio in general. Indeed Arithmetic Coding can be proven to almost reach the best compression ratio possible, which is bounded by the entropy of the data being encoded. Though during encoding the algorithm generates one code for the whole input stream, this is done in a fully sequential manner, symbol after symbol.

Arithmetic Coding, though not being very complex to understand, was not know before the late 70's in the form we use it today. It was able to gain more interest in the 80's, due to its high efficiency and the fact that the hardware implementation of Arithmetic Coding is very straightforward. First approaches to the topic were already given by *Abramson* and *Elias* in 1960, however, these days they did not come up yet with an appropriate solution to a problem we are soon going to address: The arithmetic accuracy needs to be increased with the length of the input message. Fortunately, in 1976 *Pasco* and *Rissanen* proved that specific finite-length numbers actually suffice for encoding - without any loss of accuracy. However, these algorithms were still not very memory-efficient. In 1979 and 1980 then, *Rubin*, *Guazzo*, *Rissanen* and *Langdon* published almost simultaneously the basic encoding algorithm as it is still used today. It is based on finite-precision arithmetic, employing a FIFO mechanism. The implementations by *Rissanen* and *Langdon* were also very close to later hardware implementations.

Thus, compared to other fields of Computer Science, Arithmetic Coding is still very young, however already mature and efficient principle for lossless data encoding, which satisfies all the requirements of what people understand of a modern compression algorithm: Data input streams can be compressed symbolwise, enabling on-the-fly data compression. Also Arithmetic Coding works in linear time with only constant use of memory. As mentioned above, finite precision integer arithmetic suffices for all calculations. These and other properties make it straightforward to derive hardware-based solutions. As we will see soon, Arithmetic Coding is also known to reach a best-possible compression ratio, provided the single symbols of the input stream are statistically independent, which should be the case for most data streams. Also it can be enhanced very simple by allowing simple plug-in of optimized statistical models. The decoder uses almost the same source code as the encoder which also makes the implementation straightforward.

Nowadays there are a lot of hidden applications of Arithmetic Coding, such as hardware based codecs as for instance the fax protocols G3 and G4. This kind of application makes Arithmetic Coding maximally efficient by the use of a small alphabet with an unevenly distributed probability.[2]

[2] Note that a fax page usually holds much more white pixels than black ones.

2 Introduction

Before jumping into the fray and starting with the explanation of the encoding algorithm, first we introduce some basic terms commonly used in data compression. They will be used throughout the whole paper.

Our goal is to compress data, which might either be stored on a computer-readable media or be sent over some form of stream. This data could represent anything, reaching from simple text up to graphics, binary executable programs etc.

However, we do not distinguish here between all those data types. We simply see them all as binary input. A group of such input bits is what we will refer to as a symbol. For instance one could think of an input stream being read bytewise, leading to $2^8 = 256$ different input symbols. For raw text compression, it could also suffice to take an alphabet of 128 symbols only, because the ASCII code is based on a 7-byte structure.

2.1 Foundations

DEFINITION 1 (ALPHABET AND SYMBOL)
We call a finite, nonempty set an ALPHABET. *The* LENGTH *or cardinality of an alphabet A will be referred to as $|A|$. The elements $\{a_1,\dots,a_m\}$ of an alphabet are called* SYMBOLS.

Also we assume that A is an ordered set, so giving $\{a_1,\dots,a_m\}$ a distinct order.

We already mentioned above that the Arithmetic Coding algorithm works sequentially. Thus we need some notion of what the sequential input and output of the encode/decoder might look like. This leads us directly to the notion of a SEQUENCE:

DEFINITION 2 (SEQUENCE)
A series $S = (s_1, s_2 \dots)$ of symbols s_i from an alphabet A is called SEQUENCE. *In the latter we will also use the shortcut $S = s_1 s_2 \dots$*

In analogy to the definition of $|A|$, $|S|$ is the symbol for the length of S, provided that S is of finite length. However, $|S| < \infty$ will be a general assumption henceforth, since most of the corrolary would otherwise make no sense.

Please note that this representation of data is somehow natural, since most human-made media can be read in a sequential order. Just think of books, videos, tapes and more.

Also, when looking at a sequence, one can calculate a distinct probability of each symbol of the alphabet to occur in this very sequence. This probability might be very unevenly distributed, a lot depending on the application domain. For instance consider the letter *e*, which is much more common than *z* in the English language.[3] Since Arithmetic Coding depends a lot of such statistical measures in order to achieve compression, we introduce the PROBABILITY of a symbol as follows:

DEFINITION 3 (PROBABILITY)
Let $S = (s_1,\dots,s_n)$ a finite-length sequence with $|S| = n$ over $A = \{a_1,\dots,a_m\}$. Also let $|S|_{a_i}$ the frequency of a_i in S. Then we define $P(a_i) := \frac{|S|_{a_i}}{n}$ as the PROBABILITY *of a_i (in S).*

[3] An elaboration on http://www.simonsingh.net states an average probability of 12.7% for letter *e* and 0,1% for *z*.

From the definition, we can directly conclude that $P(a_i)$ is always contained in the interval $[0,1)$ for any symbol, whereas the sum over all such probabilities is always $\sum_{i=1}^{m} P(a_i) = 1$. Please note that this interval is open-ended, because it would make no sense to encode a constant sequence holding only a symbol of probability 1, simply because in that case the full content of the sequence would have been known beforehand already. We will later on make use of this property in certain conclusions.

Recapturing the example of *e*/*z* however, we would like to emphasize that the probability of a symbol might heavily depend on its context. If one considers *e* and *z* as symbols for bytes in a binary executable for example, they might be rather evenly distributed. Also one could even show that certain symbols are more likely to occur in scientific text than newspaper articles and so forth.

Some data is subject to interpretation: E.g. consider the sequence 1111131311. It could be interpreted as a sequence of symbols $1,3$ or $11,13$. At least this example proves that we need some kind of unambiguous rule of how probabilities are related to symbols. This relation between symbols of an alphabet and their probability is commonly known as a MODEL in terms of data compression.

DEFINITION 4 (MODEL)
Let A an alphabet. A MODEL *M is a function*

$$M : A \longrightarrow [0,1) : a_i \longmapsto P_M(a_i) \ ,$$

which maps a probability $P_M(a_i)$ to each symbol $a_i \in A$.

This probability might be estimated / calculated and does not necessarily have to match the real probability of the symbol, $P(a_i)$. Indeed in most cases it does not. Please also note that an alphabet is not restricted to only hold symbols of length 1. In the example above, employing 11 and 13 as symbols we already got a picture of that. If one estimates the probability of a given symbol not only by looking at the symbol itself but also at the context given by the last *n* symbols seen, one speaks of an $Order-n\ model$. For instance the average probability of the letter *u* to occur in any German text is only about 0.0435. If one considers its probability to occur after the letter *q* however, this value raises to nearly 1! As one can see already now, an increased value of *n* might lead to better predictions of probabilities.

As already briefly mentioned above, the probability distribution that is given by the interpretation of a sequence under a certain model, matches the real probability distribution at best by chance. Usually this will not be the case. For instance there will be almost no German text fulfilling the distribution given by Table 1 exactly, but rather approximately or even worse. To distinguish the probability induced by the model from the real one, we label the former with $P_M(a_i)$ in order to emphasize the dependency of the model and in order to distinguish from the latter, given by $P(a_i)$.

So we conclude that a model can be seen as an interpretation of an arbitrary dataset. A simple model could for instance be given by the probability distribution of Table 1. This table shows the probabilities of most letters of the German alphabet to occur in an average German text. Probably the clever reader can already anticipate now, that the compression ration will heavily depend on how good this model matches the reality.

This leads to the need to define some kind of measure of compression, enabling us to actually compare the efficiency of different compression approaches. A natural measure of how much information is contained in a given sequence of data is called the ENTROPY.

a	0,0651	h	0,0476	o	0,0251	v	0,0067
b	0,0189	i	0,0755	p	0,0079	w	0,0189
c	0,0306	j	0,0027	q	0,0002	x	0,0003
d	0,0508	k	0,0121	r	0,0700	y	0,0004
e	0,1740	l	0,0344	s	0,0727	z	0,0113
f	0,0166	m	0,0253	t	0,0615		
g	0,0301	n	0,0978	u	0,0435		

Table 1: Probability of letters in an average German text (taken from [Beu94]).

Definition 5 (Entropy)
Let S a sequence over alphabet $A = \{a_1, ..., a_m\}$. The ENTROPY *$H_M(S)$ of the sequence S under model M is defined as*

$$H_M(S) = \sum_{i=1}^{m} P(a_i)\ ld\ \frac{1}{P_M(a_i)}\ . \tag{1}$$

The unit of the entropy is $[bits/symbol]$ because the formula only refers to probabilities as relative frequencies rather than absolute ones.

By the formula one can easily see that with our definition, the entropy of a sequence depends on the model M being used, since the $P_M(a_i)$ are the probabilities under that model. Here, $ld \frac{1}{P_M(a_i)}$ can be interpreted as the minimal length of a binary symbol for a_i, while the factor $P(a_i)$ (being the *real* probability of a_i) can be interpreted as probability of requiring the encoder to binary encode this very symbol. [4]

Considering a model as perfect, one obtains the *correct* probability distribution leading to the natural form of the entropy:

$$H(S) = \sum_{a \in A} P(a_i)\ ld\ \frac{1}{P(a_i)} \tag{2}$$

This kind of entropy is depended on the input data only and *no* subject to interpretation. However the interested reader might wish to know that most of the literature about Arithmetic Coding sloppily does not distinguish between both kinds of entropy.

2.2 Example: Entropy

Let us have a look at the sequence $S = abaabcda$ over alphabet $\{a, b, c, d\}$. We want to binary encode this sequence. Since we have no clue at all about how the probabilities should be distributed in the first place, we decide for the simple model M_1, which - by chance - leads to the correct probability values $P_{M_1}(a) = 0,5$, $P_{M_1}(b) = 0,25$, $P_{M_1}(c) = 0,125$ and $P_{M_1}(d) = 0,125$. One can easily see that this model is ideal in the sense that the estimated probabilities $P_{M_1}(s)$ match the real ones $P(s)$:

$$P_{M_1}(s) = P(s)\ \forall s \in A := \{a, b, c, d\}\ .$$

[4] If one does not encode binary but rather to a base m, then one only has to replace ld with log_m.

When encoding this sequence, we can do so in a very naive way by simply using 2 bits per symbol, $\{00,01,10,11\}$, which leads to overall costs of $8*2\ bits = 16\ bits$. So what about the entropy of $H_{M_1}(S)$?

$$\begin{aligned}
H_{M_1} &= \sum_{s\in\{a,b,c,d\}} P(s)\ ld\ \frac{1}{P_{M_1}(s)} \\
&= (0,5\cdot\ ld\ \frac{1}{0,5}) + (0,25\cdot\ ld\ \frac{1}{0,25}) \\
&\quad +(0,125\cdot\ ld\ \frac{1}{0,125}) + (0,125\cdot\ ld\ \frac{1}{0,125}) \\
&= 0,5\cdot\ ld\ 2 + 0,25\cdot\ ld\ 4 + 0,125\cdot\ ld\ 8 + 0,125\cdot\ ld\ 8 \\
&= 0,5 + 0,5 + 0,375 + 0,375 \\
&= 1,75\ [Bits/Symbol]
\end{aligned}$$

Note that this is given in $[Bits/Symbol]$, which means that we need a minimum of $8*1,75 = 14$ bits to encode the whole input sequence. We cannot do any better.[5] This gives a saving of $16-14 = 2$ bits.

However, what would have happened if we had not been so lucky to guess the correct probability distribution on advance? Have a look at the following model M_2 with $P_{M_2}(a) = 0,125$, $P_{M_2}(b) = 0,125$, $P_{M_2}(c) = 0,5$ and $P_{M_2}(d) = 0,25$. The entropy under M_2 calculates to:

$$\begin{aligned}
H_{M_2} &= \sum_{s\in\{a,b,c,d\}} P(s)\ ld\ \frac{1}{P_{M_2}(s)} \\
&= (0,5\cdot\ ld\ \frac{1}{0,125}) + (0,25\cdot\ ld\ \frac{1}{0,125}) \\
&\quad +(0,125\cdot\ ld\ \frac{1}{0,5}) + (0,125\cdot\ ld\ \frac{1}{0,25}) \\
&= 0,5\cdot\ ld\ 8 + 0,25\cdot\ ld\ 8 + 0,125\cdot\ ld\ 2 + 0,125\cdot\ ld\ 4 \\
&= 1,5 + 0,75 + 0,125 + 0,25 \\
&= 2,625\ [Bits/Symbol]
\end{aligned}$$

We should see this example as a warning. A warning, not to mix up the notion of *coding* with *compression*. The reason for this is that we can see that under the model M_2, we would be required to use $2,625*8 = 21$ bits to encode the input sequence. However, this would be no compression at all, if one remembers that our naive encoding with 2 bits per symbol employed 16 bits altogether only. Also we can conclude that the compression ration can only be as good as the underlying model allows. The better the model matches the reality, the better the compression will be.

However, in the following chapters we will prove, that given any particular model (that on its own might be as optimal as it can be), Arithmetic Coding achieves the absolutely best compression ratio, meaning that no other algorithm could do any better under the very same model.

[5] Note that we do not prove the entropy as measure of optimality here. This fact is commonly known as the *Shannon Theorem*[WS49].

Since we now stirred up your interest so much, we are now going to describe the actual encoding and decoding algorithms.

2.3 Encoder and decoder

DEFINITION 6 (ENCODER & DECODER)
An algorithm which encodes a sequence is called an ENCODER. *The appropriate algorithm decoding the sequence again is called a* DECODER.

In opposite to the input sequence S we refer to the encoded sequence which is output of the encoder and input for the decoder by $Code(S)$ or $C(S)$ for short.The application of both algorithms is referred to as ENCODING respectively DECODING.

We want to emphasize that we use the notion of an algorithm in its most natural way, meaning a general sequence of steps performed by any arbitrary computer. By purpose we do not limit ourselves to a certain implementation at this stage. An encoder could be any algorithm transforming the input in such a way that there is a decoder to reproduce the raw input data. However at the end of this paper we present the full C++ source code of a encoder/decoder pair (also referred to as CODEC), which employs Arithmetic Coding. The following code examples are taken from this reference implementation.

In the theory of data compression one often distinguishes between lossy and lossless compression algorithms. Especially analogous signals are often encoded in a lossy way because such data is in the end meant to be interpreted by some kind of human organ (eye, ear,...) and such organs are very limited in a sense that they simply do not recognize certain levels of noise or distortion at all. Of course lossy compression algorithms can reach better compression ratios by losing some accuracy. However we are not going to consider any lossy compression in this article and rather concentrate on lossless compression, that can be applied to all kinds of data in general. Thus we are only going to consider codecs that are able to reproduce the input data up to the last symbol. In a nutshell our resulting $Code(S)$ will be proven lossless and optimal.

2.4 The notions of uniqueness and efficiency

DEFINITION 7 (UNIQUE DECODABILITY)
We call a code UNIQUELY DECODABLE, *if any sequence is mapped to its code in an injective way. If this is the case one can determine the unique input symbol for any given code.*

A special class of uniquely decodable codes are so-called prefix codes. These can be characterized by the property that no codeword is a prefix of any other codeword:

DEFINITION 8 (PREFIX CODE)
We call a given code C a PREFIX CODE, *if for no pair (x,y) of symbols of the alphabet, $C(x)$ is prefix of $C(y)$.*

Prefix codes have the big advantage that as soon as the decoder has read $C(x)$ for a certain x, it knows at ones that the code is terminated and that symbol x was encoded. In the case of an arbitrary code, it could be the case that the decoder would have to read on in order to see if $C(x)$ was probably only the prefix of another code $C(y)$. Thus, prefix codes are known to be a class of uniquely decodable codes. The diligent reader can find a constructive proof of this property in [Say00] p.31.

Now we are fully equipped to start with the actual coding algorithm. The following chapter introduces the general method of Arithmetic Coding. The subsequent chapters evolve this method, address some of the problems one comes across and discuss the actual implementation.

3 Encoding to real numbers

Huffman-coding was considered to be almost optimal until arithmetic coding was developed in the 70s. The resulting code is usually very close to the entropy and reaches it in some special cases. Its disadvantages are the relatively complex generation of the code tree and the limitation to encode symbols or groups of symbols as such. The binary code in Huffman-coding is looked up in a balanced binary tree that approximates the symbol probabilities: One starts at the root and searches for the appropriate node for the given symbol. The branches are labeled binary, so the resulting code word is the sequence of passed branches. Since the number of passed branches in one pass is always a whole number, each symbol is always encoded in a sequence of full bits. We will show that this is an unnecessary constraint.

Arithmetic Coding uses a one-dimensional table of probabilities instead of a tree. It always encodes *the whole message* at once. This way it is possible to encode symbols using fragments of bits. However, one have cannot access the code word randomly. Using Huffman-coding, one can specify marks that allow decoding starting within the bit stream. Of course one can also split messages in arithmetic coding, but this limits the efficiency since use of bit-fragments on the boundaries is impossible.

What we are looking for is a proper way to encode a message without assigning a fixed binary code to each symbol. So let's take a look at the probabilities of the symbols: All probabilities fall into the range $[0,1)$ while their sum equals 1 in every case. This interval contains an infinite amount of real numbers, so it is possible to encode every possible sequence to a number in $[0,1)$. One partitions the interval according to the probability of the symbols. By iterating this step for each symbol in the message, one refines the interval to a unique result that represents the message. Any number in this interval would be a valid code.

Let M be a model that assigns a probability $P_M(a_i)$ to each symbol a_i that appears in the message. Now we can split the interval $[0,1)$ using these values since the sum always equals 1. The size of the $i-$th sub-interval corresponds to the probability of the symbol a_i.

3.1 Example: interval creation

Let M be a model using the alphabet $A = a,b,c,d$. Let the probabilities of the symbols in the message be

$$P_M(a) = 0.5, P_M(b) = 0.25, P_M(c) = 0.125, P_M(d) = 0.125.$$

Now the interval $[0,1)$ would be split as emphasized in Figure 1.

3.2 Upper and lower bounds

Henceforth we call the upper and lower bounds of the entire current interval *high* and *low*. The bounds of the sub-intervals are calculated from the cumulative probabilities:

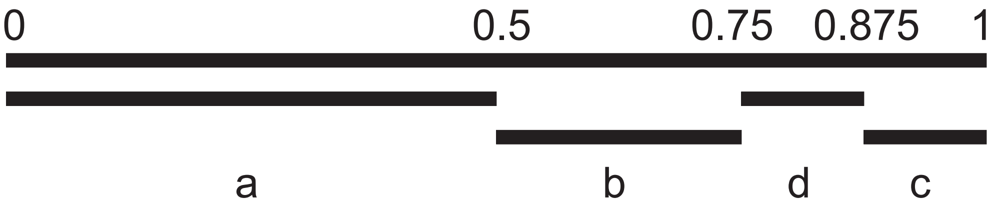

Figure 1: Creating an interval using given model

$$K(a_k) = \sum_{i=1}^{k} P_M(a_i) \; .$$

The values *high* and *low* change during the encoding process whereas the cumulative probabilities remain constant[6]. They are used to update *high* and *low*. With respect to the previous example, we get the following values:

high	1.0	$K(0)$	0.0	$K(2)$	0.75
low	0.0	$K(1)$	0.5	$K(3)$	0.875

We will see that this subdivision depends on the model. However, for now we assume that it is given by a constant table containing the cumulative probabilities $K(a_i)$. This type of model also exists in real applications and is called *static*.

3.3 Encoding

The first step in encoding is the initialization of the interval $I := [low, high)$ by $low = 0$ and $high = 1$. When the first symbol s_1 is read, the interval I can be resized to a new interval I' according to the symbol. The boundaries of I' are also called *low* and *high*. We choose I' to equal the boundaries of s_1 in the model. However, how are these boundaries calculated? Let $s_1 = a_k$ be the kth symbol of the alphabet. Then the lower bound is

$$low := \sum_{i=1}^{k-1} P_M(a_i) = K(a_{k-1})$$

and the upper bound is

$$high := \sum_{i=1}^{k} P_M(a_i) = K(a_k)$$

The new interval I' is set to $[low, high)$. This calculation is nothing new, it just corresponds to the mathematical method of the construction of Figure 1. The most relevant aspect of this method is

[6] provided we are using a constant model

that the sub-interval I' becomes larger for more probable symbols s_1. The larger the interval the lower the number of fractional places which results in shorter code words. All following numbers generated by the next iterations will be located in the interval I' since we use it as base interval as we did used $[0,1)$ before.

We proceed with the second symbol $s_2 = a_j$. However, now we have the problem that our model M describes a partition[7] of the interval $[0,1)$, not of I' which was calculated in the previous step. We have to scale and shift the boundaries to match the new interval. Scaling is accomplished by a multiplication with $high - low$, the length of the interval. Shifting is performed by adding low. This results in the equation

$$
\begin{aligned}
low' &:= low + \sum_{i=1}^{j-1} P_M(a_i) \cdot (high - low) = low + K(a_{j-1}) \cdot (high - low) \; ; & (3) \\
high' &:= low + \sum_{i=1}^{j} P_M(a_i) \cdot (high - low) = low + K(a_j) \cdot (high - low) \; . & (4)
\end{aligned}
$$

This rule is valid for all steps, especially the first one with $low = 0$ and $high - low = 1$. Since we do not need the old boundaries any more for the next iterations, we can overwrite them:

$$
\begin{aligned}
low &:= low' \; ; \\
high &:= high' \; .
\end{aligned}
$$

This iteration might look complicated, but we will give an example resembling the setting in 2.2. Figure 3 on page 16 gives a picture of this. Let S be the sequence *abaabcda* using our ideal model M_1.

We start with the interval $[0,1)$ and the first element of S. Since s_1 is an **a**, the new boundaries are calculated as follows:

$$
\begin{aligned}
low &= 0 \\
high &= 0 + 0.5 \cdot 1 = 0.5 \; .
\end{aligned}
$$

The resulting interval is $[0 \ldots 0.5)$. The next iteration encodes a **b**

$$
\begin{aligned}
low &= 0 + 0.5 \cdot (0.5 - 0) = 0.25 \\
high &= 0 + 0.5 \cdot (0.5 - 0) + 0.25 \cdot (0.5 - 0) = 0.375 \; .
\end{aligned}
$$

followed by a second **a**

$$
\begin{aligned}
low &= 0.25 \\
high &= 0.25 + 0.5 \cdot (0.375 - 0.25) = 0.3125 \; .
\end{aligned}
$$

[7] A *partition* is a disjoint union of sets called *classes*. All classes have empty intersections and the union of all classes results in the base set.

and a third **a**

$$\begin{aligned} low &= 0.25 \\ high &= 0.25 + 0.5 \cdot (0.3125 - 0.25) = 0.28125\ , \end{aligned}$$

The fifth character is a **b**

$$\begin{aligned} low &= 0.25 + 0.5 \cdot (0.28125 - 0.25) = 0.265625 \\ high &= 0.25 + 0.5 \cdot (0.28125 - 0.25) + 0.25 \cdot (0.28125 - 0.25) = 0.2734375\ . \end{aligned}$$

followed by a **c**

$$\begin{aligned} low &= 0.265625 + 0.5 \cdot (0.2734375 - 0.265625) + 0.25 \cdot (0.2734375 - 0.265625) \\ &= 0.271484375 \\ high &= 0.265625 + 0.5 \cdot (0.2734375 - 0.265625) + 0.25 \cdot (0.2734375 - 0.265625) \\ &\quad + 0.125 \cdot 0.25 \cdot (0.2734375 - 0.265625) \\ &= 0.2724609375\ , \end{aligned}$$

a **d**

$$\begin{aligned} low &= 0.271484375 + (0.5 + 0.25 + 0.125) \cdot (0.2724609375 - 0.271484375) \\ &= 0.2723388672 \\ high &= 0.2724609375\ , \end{aligned}$$

and at last another **a**

$$\begin{aligned} low &= 0.2723388672 \\ high &= 0.2723388672 + 0.5 \cdot (0.2724609375 - 0.2723388672) \\ &= 0.2723999024\ . \end{aligned}$$

So the resulting interval is $[0.2723388672; 0.2723999024)$.

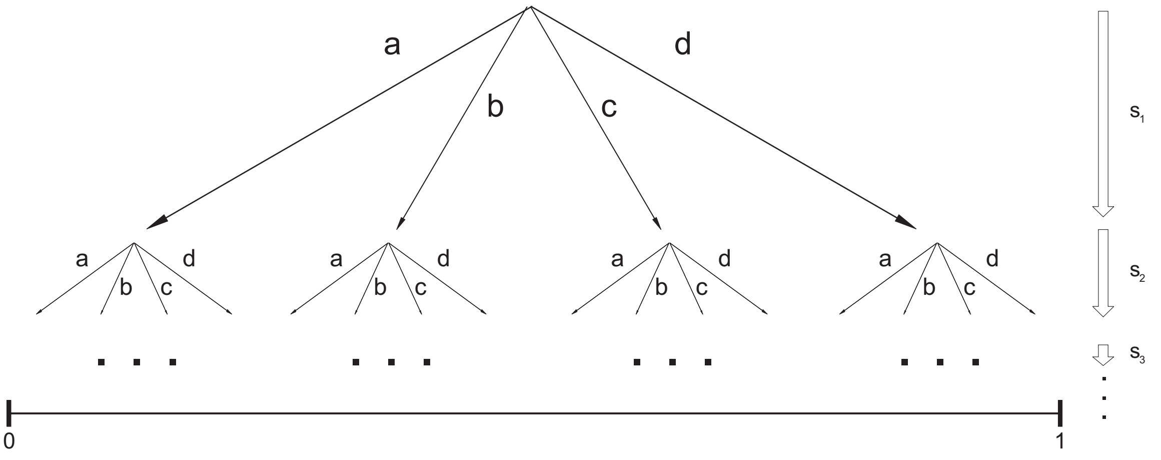

Figure 2: Iterated partitioning of the interval $[0, 1)$ (uniform distribution)

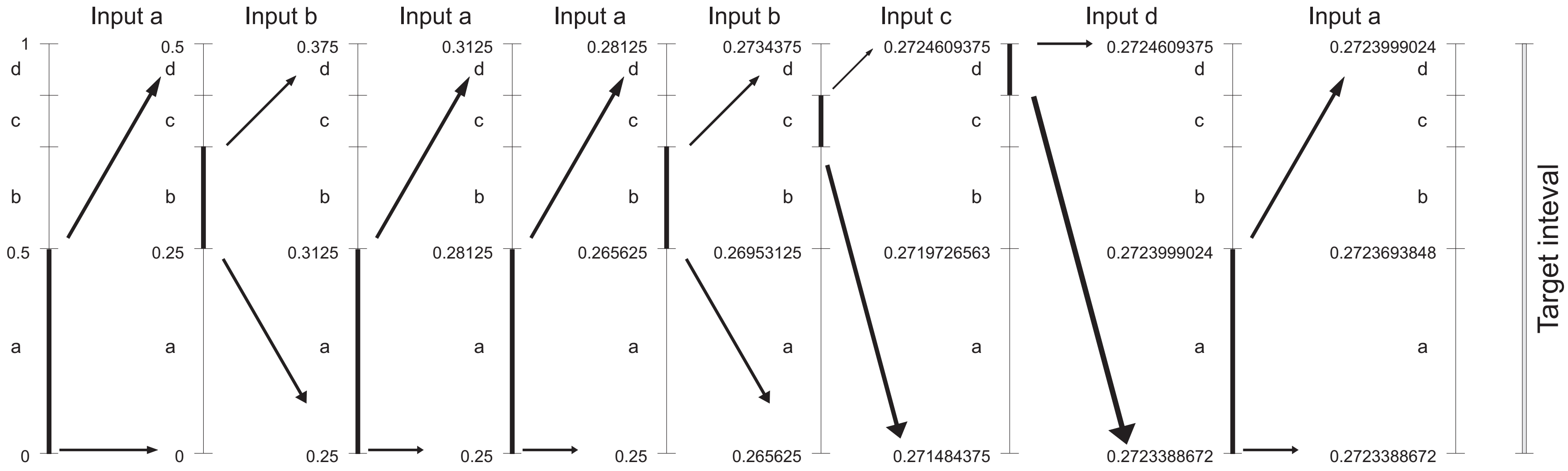

Figure 3: Function of the encoder

The next matter is the actual code. We have to specify the calculated interval. So we could simply save the upper and lower bound, but this is rather inefficient. Knowing that the whole interval is unique for this message, we can safely store only a single value inside the interval. The following lemma should clarify this technique.

LEMMA 1 *The codes of all messages with the same length form a partition of the interval* $I := [0,1)$.

This results clearly from Figure 2. A direct conclusion of the lemma is the fact that the classes of the partition become infinitely small for infinitely long messages. There are no infinitely long messages in practice, but there are very large messages and the corresponding small partitions would cause problems on common computers using finite arithmetics. A solution it the rescaling presented in section 5.

In the last example we can for instance store 0.27234 or any other value in the interval. Here we still assume that we know when the message ends, although this is usually not the case (think of remote data transmissions). End-of-message handling is discussed later, for now we will proceed with a short summary on encoding:

```
low =0;
high=1;
do {
    temp = read_symbol();
    ival = model->get_interval(temp); \\ returns the interval containing temp
    low =  calculate_lower_bound(ival);
    high = calculate_upper_bound(ival);
} while (!end_of_sequence());
return(value_in_interval(low,high));
```

3.4 Decoding

To decode a sequence, one somewhat have to apply the encoder backwards. The value $V := Code(S)$ is given and we have to restore the original sequence S. We assume that the message length is known and equals l. In the first iteration we compare V with each interval $I' := [K(a_k - 1), K(a_k))$ to find the one that contains V. It corresponds to the first symbol of the sequence, s_1. To compute the next symbol, we have to modify the probability partition in the same way we did while encoding:

$$\begin{aligned} low' &:= low + K(a_{i-1}) \cdot (high - low) \\ high' &:= low + K(a_i) \cdot (high - low) \ , \end{aligned}$$

where i has to comply

$$low \leq V \leq high$$

a_i is the next symbol in the encoded sequence. This time, the start case is again a special case of the general formula. The iteration is very similar to the encoder, so from its implemention should arise no further problems.

3.5 Decoding example

We illustrate the decoder using the same data as in the previous examples. The resulting code was $V = 0.27234$ and we assume that we know the length $l = 8$. Starting with $low = 0$ and $high = 1$ we see that V lies inside the first intervall $[0 \dots 0.5)$. The corresponding symbol is an **a** and we set

$$\begin{aligned} low &= 0 \\ high &= 0.5 \end{aligned}$$

In the next iteration we see that $0,27234$ lies between the boundaries

$$\begin{aligned} low &= 0 + \underline{0.5} \cdot (0.5 - 0) = 0.25 \\ high &= 0 + \underline{0.75} \cdot (0.5 - 0) = 0.3125 \end{aligned}$$

and decode a **b**. The relevant boundaries are underlined. The next iteration

$$\begin{aligned} low &= 0.25 + \underline{0} \cdot (0.3125 - 0.25) = 0.25 \\ high &= 0.25 + \underline{0.5} \cdot (0.3125 - 0.25) = 0.28125 \end{aligned}$$

results in an **a**. Since the next iterations are very similar, we skip them and take a look at the last iteration:

$$\begin{aligned} low &= 0.2723388672 + \underline{0} \cdot (0.2724609375 - 0.2723388672) = 0.2723388672 \\ high &= 0.2723388672 + \underline{0.5} \cdot (0.2724609375 - 0.2723388672) = 0.2723999024 \end{aligned}$$

This is the final **a** in the sequence *abaabcda*. Because of the similarities, one can use Figure 3 in this case, too. The decoding algorithm can be summarized as follows:

```
seq = '';
low = 0;
high = 1;
do {
    low'  = model->lower_bound(Value,low,high);
    high' = model->upper_bound (Value,low,high);
    low  = low';
    high = high';
    seq .= model->symbol_in_interval(low,high);
} while ( !end_of_sequence() );
return(seq);
```

We used floating point arithmetic to calculate the boundaries, but without further methods, this results in a large number of fractional places. In particular, it is possible that infinite numbers of fractional places appear (consider $1/3$). The circumvention of this problem is covered by the next subsection. Note that the for the implementation of those methods it makes no difference if one works over symbols or sequences. One can see this by working with the probability distributions of sequences (see [Say00]).

3.6 Uniqueness of representation

Let $C(a_i)$ be a code for a_i:

$$C(a_i) := K(a_{i-1}) + \frac{1}{2}\, P_M(a_i)\ .$$

$C(a_i)$ is the center of the interval of a_i. One can replace $C(a_i)$ by a shortened code of the length

$$l(a_i) = \lceil\, ld\ \frac{1}{P_M(a_i)} \rceil + 1\ .$$

$\lfloor C(a_i) \rfloor_{l(a_i)}$ is defined as the binary code for a_i shortened to $l(a_i)$ digits.

3.6.1 Example

Let S be the sequence

$$S = s_1 s_2 s_3 s_4$$

over the alphabet $A = \{a_1, \ldots, a_4\}$. Let the probabilities computed by the model M be

$$P_M(a_1) = \frac{1}{2},\ P_M(a_2) = \frac{1}{4},\ P_M(a_3) = \frac{1}{8},\ P_M(a_4) = \frac{1}{8}.$$

The following table shows a possible binary code for this sequence. The binary representation of $C(a_i)$ was shortened to $\lceil\, ld\ \frac{1}{P_M(a_i)} \rceil + 1$ which led to the respective code.

i	$K(a_i)$	$C(a_i)$	binary	$l(a_i)$	$\lfloor C(a_i) \rfloor_{l(a_i)}$	Code
1	0.5	0.25	0.0100	2	0.01	01
2	0.75	0.625	0.1010	3	0.101	101
3	0.875	0.8125	0.1101	4	0.1101	1101
4	1.0	0.9375	0.1111	4	0.1111	1111

3.6.2 Proof

We will now show that the code that was generated in the described way is unique. Beforehand we chose the code $C(a_i)$ to represent the symbol a_i. However, any other value in the interval $[K(a_{i-1}), K(a_i))$ would also result in an unique code for a_i. To show that the code $\lfloor C(a_i) \rfloor_{l(a_i)}$ is unique, it is consequently enough to show that the code lies in the interval $[K(a_{i-1}), K(a_i))$. Since we cut off the binary representation of $C(a_i)$ to get $\lfloor C(a_i) \rfloor_{l(a_i)}$, the following equation is satisfied:

$$\lfloor C(a_i) \rfloor_{l(a_i)} \leq C(a_i).$$

Or in detail:

$$0 \leq C(a_i) - \lfloor C(a_i) \rfloor_{l(a_i)} \leq \frac{1}{2^{l(a_i)}}. \tag{5}$$

Since $C(a_i)$ is smaller than $K(a_i)$ by definition, it follows that

$$\lfloor C(a_i) \rfloor_{l(a_i)} < K(a_i).$$

This satisfies the upper bound. The next equation deals with the lower bound $\lfloor C(a_i) \rfloor_{l(a_i)} \geq K(a_{i-1})$:

$$\begin{aligned}
\frac{1}{2^{l(a_i)}} &\overset{\text{def}}{=} \frac{1}{2^{\lceil\, ld\ \frac{1}{P_M(a_i)} \rceil + 1}} \\
&\leq \frac{1}{2^{\ ld\ \frac{1}{P_M(a_i)} + 1}} \\
&= \frac{1}{2 \cdot 2^{\ ld\ \frac{1}{P_M(a_i)}}} \\
&= \frac{1}{2 \frac{1}{P_M(a_i)}} \\
&= \frac{P_M(a_i)}{2}\ .
\end{aligned}$$

By definition of $C(a_i)$, the following is valid:

$$\frac{P_M(a_i)}{2} = C(a_i) - K(a_{i-1})$$

Consequently

$$C(a_i) - K(a_{i-1}) \geq \frac{1}{2^{l(a_i)}} \; . \tag{6}$$

is satisfied. The combination of (5) and (6) results in

$$\lfloor C(a_i) \rfloor_{l(a_i)} \geq K(a_{i-1}) \; . \tag{7}$$

which implies

$$K(a_{i-1}) \leq \lfloor C(a_i) \rfloor_{l(a_i)} < K(a_i) \; ,$$

and thus

$$\lfloor C(a_i) \rfloor_{l(a_i)} \in [K(a_{i-1}), K(a_i)) \; .$$

□

Therewith it is proven that $\lfloor C(a_i) \rfloor_{l(a_i)}$ is a non-ambiguous representation of $C(a_i)$. To show that it is non-ambiguously decodable, it suffices to show that it is a prefix code, since we already know that any prefix code is non-ambiguously decodable.

Given a number a in the interval$[0,1)$ with binary representation of the length n, $[a_1, a_2, ..., a_n]$. It is obvious that any other number b with the prefix $[a_1, a_2, ..., a_n]$ in binary representation lies in the interval $[a, a + \frac{1}{2^n})$. If a_i and a_j are different, we know that the values $\lfloor C(a_i) \rfloor_{l(a_i)}$ and $\lfloor C(a_j) \rfloor_{l(a_j)}$ lie in two disjunct intervals

$$[K(a_{i-1}), K(a_i)), [K(a_{j-1}), K(a_j))$$

If we are able to show that for any symbol a_i the interval

$$[\lfloor C(a_i) \rfloor_{l(a_i)}, \lfloor C(a_i) \rfloor_{l(a_i)} + \frac{1}{2^{l(a_i)}})$$

is contained in $[K(a_{i-1}), K(a_i))$, this implies that the code of symbol a_i cannot be prefix of the code of another symbol a_j.

Equation (7) implies $\lfloor C(a_i) \rfloor_{l(a_i)} \geq K(a_{i-1})$. That proves the assumption for the lower bound, so it is sufficient to show

$$K(a_i) - \lfloor C(a_i) \rfloor_{l(a_i)} > \frac{1}{2^{l(a_i)}}.$$

This is obvious because of

$$\begin{aligned} K(a_i) - \lfloor C(a_i) \rfloor_{l(a_i)} &> K(a_i) - C(a_i) \\ &= \frac{P_M(a_i)}{2} \\ &\geq \frac{1}{2^{l(a_i)}}. \end{aligned}$$

Therefore the code is prefix free. In particular, shortening $C(a_i)$ to $l(a_i)$ Bits results in a non-ambiguously decodable code. Hence we solved the problem of finite arithmetics with floating point numbers.

3.7 Summary

We have got to know the theoretical function of arithmetic coding and have seen several examples. All this was based on floating point arithmetic with infinite precision. We showed that it is actually possible to use this in implementations, but using integers usually results in faster and easier implementations. In the following we show how to use integer arithmetic, which raises new problems with respect to finite arithmetics. The output won't be a real number as in this chapter, instead we will use a sequence of bits. This sequence will have to be terminated properly.

4 Encoding as sequence of bits

4.1 Motivation

To implement arithmetic coding efficiently, we have to make restrictions: There are no (infinite) real numbers, and pure integer implementations are way faster on simple processors as found in fax machines (which actually use arithmetic coding in the G3 protocol).

This chapter covers an implementation with very low memory consumption (only one register for the boundaries, using 32 bits in the examples) and only a few simple integer instructions. The output is a non-ambiguous sequence of bits that can be stored or send on the fly.

4.2 Abstracting from the model

One cannot express probabilities in fractions of 1 using integers. Since probabilities equal the frequencies of symbol occurrences in simple models, one can normalize them to the number of symbols. The lower bound is the sum of frequencies of all lower symbols (in canonical order), the upper bound is this sum plus the frequency of the current symbol:

$$\begin{aligned} low\_count &= \sum_{i=0}^{symbol-1} CumCount[i] \ , \\ high\_count &= low\_count + CumCount[symbol] \ , \end{aligned}$$

where `CumCount` contains the cumulative frequency counts. This resembles the probabilities of the previous section in so far as one does not divide by the total count before adding up. This results in the following equations:

$$\begin{aligned} low\_count &= low \cdot total \ , \\ high\_count &= high \cdot total \ , \end{aligned}$$

where `total` represents the total count.

4.3 Encoding

The encoder consists of a function and static variables that store the current state[8]:

- `mLow` [9] stores the current lower bound. It is initialized with `0`.
- `mHigh` stores the current upper bound. It is initialized with `0x7FFFFFFF`, the maximum value that fits in 31 bits.
- `mStep` stores a step size that is introduced later. It is not necessarily static in the encoder, but the decoder depends on this property.

[8] This implementation follows [WBM94].

[9] The prefix m denotes static variables. This resembles member variables in object oriented programming.

Note that one can use only 31 of 32 bits to prevent overflows. We will go into this later. The function declaration looks as follows:

```
  void Encoder( unsigned int low_count,
                unsigned int high_count,
                unsigned int total );
```

The cumulative probabilities (which are calculated by the model) of the current symbol a_i and the next lower symbol a_{i-1} are passed to the encoder. The encoder computes the new upper and lower bounds from these. At first, the interval from `mLow` to `mHigh` is divided into `total` steps, resulting in a step size of

```
mStep = ( mHigh - mLow + 1 ) / total;
```

One has to add 1 to the difference of `mHigh` and `mLow` since `mHigh` represents the open upper bound. Therefore the interval is larger by 1. An analogy in the common decimal system would be an interval from 0 to $99.\overline{9}$ where the upper bound is stored as 99. The fractional places make the interval larger by 1 compared to $99 - 0 = 99$.

This is also the reason for the limitation to 31 bits: `mHigh` is initialized with the maximum possible value. If one would choose 32 bit, then the addition `mHigh - mLow + 1` would result in an overflow, which might lead to an exception in the best case or even to sporadic encoding errors, which would result in file corruption.

However the upper bound is now updated to

```
mHigh = mLow + mStep * high_count - 1;
```

and the lower bound to

```
mLow = mLow + mStep * low_count;
```

Both calculations rely on the previous value of `mLow`, therefore overwriting it has to be the last step. Since we are dealing with an open interval, we have to decrease `mHigh` by one to reflect this.

4.4 Example: encoding

This time we handle the same input sequence as in all previous examples (see 2.2), but limit ourselves to the first two symbols, `ab`. The model specifies the following data:

At first we initialize the encoder:

```
mBuffer = 0;
mLow = 0;
mHigh = 0x7FFFFFFF;
```

Then we encode the symbols of the sequence. Note that the model is static, so `total` stays constant.

symbol	frequency	low_count	high_count
a	4	0	4
b	2	4	6
c	1	6	7
d	1	7	8

Table 2: Model for the example 2.2

1. ’a’

   ```
   mStep = ( mHigh - mLow + 1 ) / total;
         = ( 0x7FFFFFFF - 0 + 1) / 8
         = 0x80000000 / 8
         = 0x10000000

   mHigh = mLow + mStep * high_count - 1;
         = 0 + 0x10000000 * 4 - 1
         = 0x40000000 - 1
         = 0x3FFFFFFF

   mLow  = mLow + mStep * low_count;
         = 0 + 0x10000000 * 0
         = 0
   ```

2. ’b’

   ```
   mStep = ( mHigh - mLow + 1 ) / total;
         = ( 0x3FFFFFFF - 0 + 1 ) / 8
         = 0x40000000 / 8
         = 0x08000000

   mHigh = mLow + mStep * high_count - 1;
         = 0 + 0x08000000 * 6 - 1
         = 0x30000000 - 1
         = 0x2FFFFFFF

   mLow  = mLow + mStep * low_count;
         = 0 + 0x08000000 * 4
         = 0x20000000
   ```

After these two symbols we can store any value in the interval from `0x20000000` to `0x2FFFFFFF`.

4.5 Decoding

The task of the decoder is to follow the steps of the encoder one by one. Hence we have to determine the first symbol and update the bounds accordingly. This divides the decoder functionality into two functions:

```
unsigned int Decode_Target( unsigned int total );
void Decode( unsigned int low_count, unsigned int high_count );
```

`Decode_Target()` determines the interval that contains the symbol. This is accomplished by calculating the code value of the symbol:

```
mStep = ( mHigh - mLow + 1 ) / total;
value = ( mBuffer - mLow ) / mStep;
```

`mBuffer` is the variable that contains the encoded sequence. The model can use the return value to determine the encoded symbol by comparing it to the cumulative count intervals. As soon as the proper interval is found, the boundaries can be updated like they were during encoding:

```
mHigh = mLow + mStep * high_count - 1;
mLow = mLow + mStep * low_count;
```

Note that `mStep` is reused. That is why it was declared statically in the first place.

4.6 Example: decoder

Now we decode the sequence of bits that was generated in the encoding example 4.4. Let `0x28000000` be the value that was stored by the encoder. We initialize the decoder using the following values:

```
mBuffer = 0x28000000;
mLow = 0;
mHigh = 0x7FFFFFFF;
```

1. ’a’ At first we calculate a value compatible to the model using `Decode_Target()`:

   ```
   mStep = ( mHigh - mLow + 1 ) / total;
         = ( 0x7FFFFFFF - 0 + 1 ) / 8
         = 0x80000000 / 8
         = 0x10000000

   value = ( mBuffer - mLow ) / mStep;
         = ( 0x28000000 - 0 ) / 0x10000000
         = 0x28000000 / 0x10000000
         = 2
   ```

   This 2 is now compared to Table 2 which represents the model. It’s found in the interval $[0,4)$, therefore the encoded symbol is an a. We update the bounds using `Decode()`:

   ```
   mHigh = mLow + mStep * high_count - 1;
         = 0 + 0x10000000 * 4 - 1
         = 0x40000000 - 1
   ```

```
      = 0x3FFFFFFF

mLow  = mLow + mStep * low_count;
      = 0 + 0x10000000 * 0
      = 0
```

2. ’b’

```
Decode_Target():

mStep = ( mHigh - mLow + 1 ) / total;
      = ( 0x3FFFFFFF - 0 + 1 ) / 8
      = 0x40000000 / 8
      = 0x08000000

value = ( mBuffer - mLow ) / mStep;
      = ( 0x28000000 - 0 ) / 0x08000000
      = 0x28000000 / 0x08000000
      = 5

Decode():

mHigh = mLow + mStep * high_count - 1;
      = 0 + 0x08000000 * 6 - 1
      = 0x30000000 - 1
      = 0x2FFFFFFF

mLow  = mLow + mStep * low_count;
      = 0 + 0x08000000 * 4
      = 0x20000000
```

This 5 is located in the interval corresponding to *b*. Now we have decoded the sequence *ab* successfully.

5 Scaling in limited ranges

5.1 Motivation

When we use the presented methods to encode several symbols, a new problem arises: `mLow` and `mHigh` converge more and more and so further encoding will be impossible as soon as the two values coincide. However, there is a simple solution based on the following observation:

5.2 E1 and E2 scaling

As soon as `mLow` and `mHigh` lie in the same half of the range of numbers (in this case $<$ or $\geq$ `0x40000000`), it is guaranteed that they will never leave this range again since the following symbols will shrink the interval. Therefore the information about the half is irrelevant for the following steps and we can already store it and remove it from consideration.

Given the presented implementation, the most significant bits (MSB) of `mLow` and `mHigh` are equal in this case. 0 corresponds to the lower half while 1 represents the upper. As soon as the MSBs match, we can store them in the output sequence and shift them out. This is called $E1$- respective $E2$-scaling:

```
while( ( mHigh < g_Half ) || ( mLow >= g_Half ) ) {
    if( mHigh < g_Half ) // E1
    {
        SetBit( 0 );
        mLow = mLow * 2;
        mHigh = mHigh * 2 + 1;
    }
    else if(mLow >= g_Half ) // E2
    {
        SetBit( 1 );
        mLow = 2 * ( mLow - g_Half );
        mHigh = 2 * ( mHigh - g_Half ) + 1;
    }
}
```

`g_Half` is the global constant `0x40000000` that marks the center of the range. The multiplication by 2 enlarges the interval. The addition of 1 fixes the upper bound as we deal with an open interval. It is equivalent to the more intuitive solution: Whenever a calculation involves an open bound, add 1 before and subtract 1 after it. Adding 1 after the multiplication by 2 produces the same result.

`SetBit()` adds a bit to the output sequence. The complementary function in the decoder is called `GetBit()`. Both functions work sequentially, one can interpret them as FIFO queue. Seeking in the encoded sequence is neither possible nor required, because the algorithm itself works sequentially, too.

5.3 E3 scaling

Though E1 and E2 scaling are a step in the right direction, they are not sufficient on their own. They won't work when mLow and mHigh converge to the center of the interval: Both stay in their halves, but the interval soon becomes too small. The extreme case would be a value of 0x3FFFFFFF for mLow and 0x40000000 for mHigh. They differ in every bit (apart from the one reserved for overflows), but further encoding is impossible.

This is where E3 scaling comes into play: As soon as mLow leaves the lowest quarter (maximum value of the first quarter: g_FirstQuarter) and mHigh the highest (fourth) quarter (maximum value of the third quarter: g_ThirdQuarter), the total range is less than half of the original range and it is guaranteed that this won't change because of the ongoing shrinking. It is not immediately determinable which half will contain the result, but as soon as the next E1 or E2 scaling is possible, one knows the values that one could have stored earlier if one were able to foresee this. This might sound strange, but it's the way E3 scaling works: One enlarges the interval just as one did with E1 or E2 scaling, but instead of storing a bit in the output sequence, one remembers that one did a E3 scaling using the helper variable mScale:

```
while( ( g_FirstQuarter <= mLow ) && ( mHigh < g_ThirdQuarter )) {
    mScale++;
    mLow = 2 * ( mLow - g_FirstQuarter );
    mHigh = 2 * ( mHigh - g_FirstQuarter ) + 1;
}
```

On the next E1 or E2 scaling, one adds the correct bit for each E3 scaling to the output sequence. Using E3 scaling followed by E1 scaling means that the interval would have fit into the range between g_FirstQuarter and g_Half. This is equivalent to an E1 scaling followed by an E2 scaling.[10] The sequence E3-E2 can be interpreted analogous, the same goes for iterated E3 scalings. Hence one has to store E3 scalings after the next E1/E2 scaling, using the inverse bit of that scaling:

```
while( ( mHigh < g_Half ) || ( mLow >= g_Half ) ) {
    if( mHigh < g_Half ) // E1
    {
        SetBit( 0 );
        mLow = mLow * 2;
        mHigh = mHigh * 2 + 1;

        // E3
        for(; mScale > 0; mScale-- )
        SetBit( 1 );
    }
    else if(mLow >= g_Half ) // E2
    {
        SetBit( 1 );
        mLow = 2 * ( mLow - g_Half );
        mHigh = 2 * ( mHigh - g_Half ) + 1;
```

[10] We prove this on page 30.

```
        // E3
        for(; mScale > 0; mScale-- )
        SetBit( 0 );
    }
}
```

This coherence is illustrated by the figures 4 and 5 on page 32. Let A be the alphabet $A := \{a, b, c, d, e\}$ using uniformly distributed probabilities. Figure 4 shows a c as the first symbol. The corresponding interval is $[0.4, 0.6)$ that covers the second and third quarter. Therefore we can apply E3 scaling and the resulting interval covers the second and third quarter again. After the next E3 scaling, the interval covers more than two quarters, so we have to proceed with the next symbol b. The resulting interval is $[0.375, 0.5)$ which is contained completely in the lower half. The E1 scaling stores a 0 in the output sequence, followed by two 1 bits for the E3 scalings.

Figure 5 illustrates why storing 011 was correct. Starting with the intervall $[0, 1)$, we apply E1 and E2 scalings according to the stored bits, meaning one E1 and two E2 scalings. The resulting interval is the same as in 4 which shows the result of two E3 scalings followed by one E1 scaling.

This is valid in general. Let f and g be two functions and $g \circ f$ the consecutive application of f and g. Then we can express the method as follows:

LEMMA 2 *Applied to any sequence, the following equations are valid:*
$E1 \circ (E3)^n = (E2)^n \circ E1,$
$E2 \circ (E3)^n = (E1)^n \circ E2.$

Proof:
Let $a := low$, $b := high$ and $I := [0, 1)$ be the interval we are working with. The scaling functions can be expressed as follows:

$$\begin{aligned} E1\begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} 2a \\ 2b \end{pmatrix} \\ E2\begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} 2a-1 \\ 2b-1 \end{pmatrix} \\ E3\begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} 2a-\frac{1}{2} \\ 2b-\frac{1}{2} \end{pmatrix} \end{aligned}$$

The nth iteration results in

$$\begin{aligned} E1^n\begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} 2^n a \\ 2^n b \end{pmatrix} \\ E2^n\begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} 2^n a - 2^n + 1 \\ 2^n b - 2^n + 1 \end{pmatrix} \\ E3^n\begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} 2^n a - 2^{n-1} + \frac{1}{2} \\ 2^n b - 2^{n-1} + \frac{1}{2} \end{pmatrix} \end{aligned}$$

The proof by induction can be done by the reader with little effort. This results in the following equation:

$$(E1 \circ (E3)^n)\binom{a}{b} \quad = \quad E1\begin{pmatrix} 2^n a - 2^{n-1} + \frac{1}{2} \\ 2^n b - 2^{n-1} + \frac{1}{2} \end{pmatrix} = \begin{pmatrix} 2^{n+1} a - 2^n + 1 \\ 2^{n+1} b - 2^n + 1 \end{pmatrix} \tag{8}$$

$$((E2)^n \circ E1)\binom{a}{b} \quad = \quad (E2)^n \binom{2a}{2b} = \begin{pmatrix} 2^{n+1} a - 2^n + 1 \\ 2^{n+1} b - 2^n + 1 \end{pmatrix} \tag{9}$$

Equating (8) and (9) implies:

$$E1 \circ (E3)^n = (E2)^n \circ E1$$

The second identity can be proven in an analogous way. □

5.4 Example encoding

We encode the input sequence `abccedac` over $A = a, b, c, d, e$ for further illustration of the E3 scaling. The model has to be adjusted according to the Table 3 on page 31. The example is presented in Table 4 to improve readability. The first column contains the symbol that should be encoded next. The following three columns show the parameters that are passed to `Encode()`. They are followed by the computed bounds `mLow` and `mHigh`. The next columns contain E1 and E2 scalings together with the resulting output bits. Underlined bits represent bits of E3 scalings. The next columns show further E3 scalings and the updated bounds, followed by the required bits to chose a value inside these bounds.

This example is limited to 7 bit integers. This is sufficient for our sequence and far easier to read than 31 bit.

symbol	frequency	`low_count`	`high_count`
a	2	0	2
b	1	2	3
c	3	3	6
d	1	6	7
e	1	7	8

Table 3: Model for the example of scaling functions

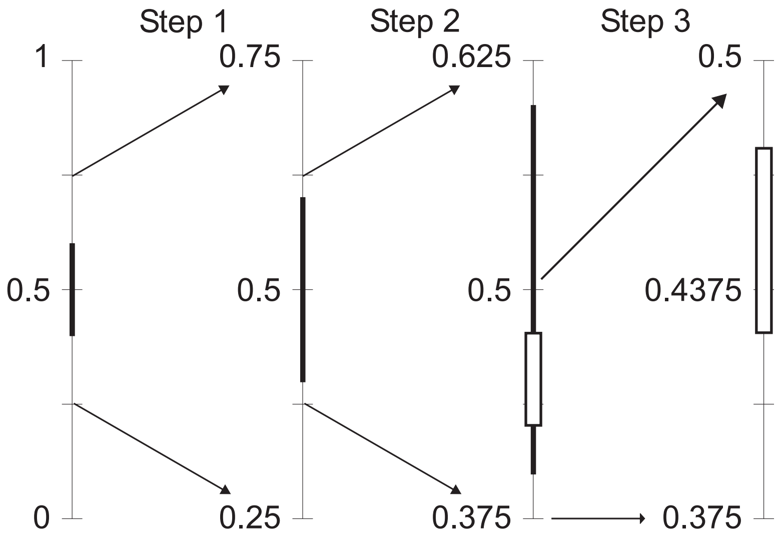

Figure 4: Application of E3 scaling

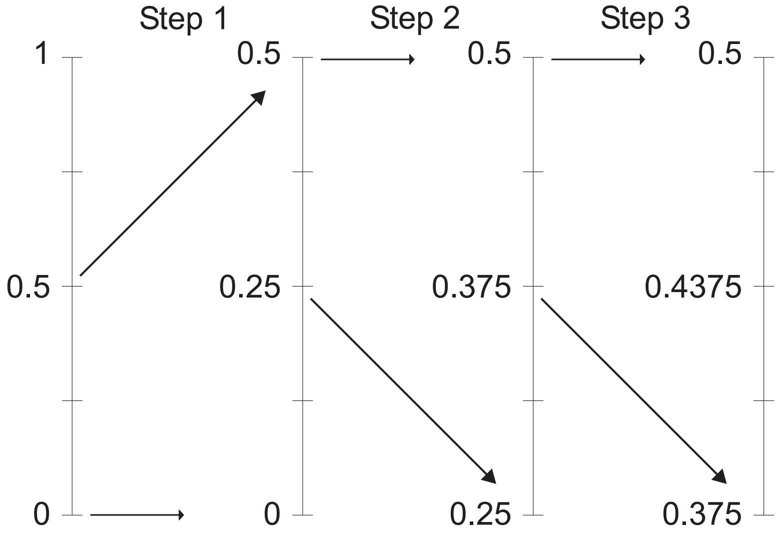

Figure 5: For comparison - without E3 scaling

Sym	l_c	h_c	t	mStep	mLow	mHigh	Bits	E1/2-mLow	E1/2-mHigh	mScale	E3-mLow	E3-mHigh
a	0	2	8	16	0000000[0]	0011111[31]	00	0000000[0]	1111111[127]	0		
b	2	3	8	16	0100000[32]	0101111[47]	010	0000000[0]	1111111[127]	0		
c	3	6	8	16	0110000[48]	1011111[95]				1	0100000[32]	1111111[127]
c	3	6	8	12	1000100[68]	1100111[103]	$1\underline{0}$			0		
e	7	8	8	9	1001111[71]	1001111[79]	100	0111000[56]	1111111[127]	0		
d	6	7	8	9	1101110[110]	1110110[118]	11	0111000[56]	1011011[91]	1	0110000[48]	1110111[119]
a	0	2	8	9	0110000[48]	1000001[65]				3	0000000[0]	1000111[71]
c	3	6	8	9	0011011[27]	0110101[53]	$0\underline{111}$	0110110[54]	1101011[107]	0		
rest							1					

Table 4: Example of scaling functions in the encoder

Sym	current symbol
l_c	low_count, lower bound of the cumulative frequency of the symbol
h_c	high_count, upper bound of the cumulative frequency of the symbol
t	total, total frequency count
mStep	step size
mLow	lower bound of the new interval
mHigh	upper bound of the new interval
Bits	Bits that are stored and removed by E1/E2 scalings
E1/2-mLow	lower bound after E1/E2 scaling
E1/2-mHigh	upper bound after E1/E2 scaling
mScale	sum of the new and the remaining E3 scalings
E3-mLow	lower bound after E3 scaling
E3-mHigh	upper bound after E3 scaling

Table 5: Explanation of columns

5.5 Decoding

Since the decoder follows the steps of the encoder, the scalings work the same. However, note that one has to update the buffer mBuffer, too. This works the same way the bounds are updated, one just does not generate the subsequent bits but rather take them from the encoded sequence.

```
// E1 scaling
mLow = mLow * 2;
mHigh = mHigh * 2 + 1;
mBuffer = 2 * mBuffer + GetBit();

// E2 scaling
mLow = 2 * ( mLow - g_Half );
mHigh = 2 * ( mHigh - g_Half ) + 1;
mBuffer = 2 * ( mBuffer - g_Half ) + GetBit();

// E3 scaling
mLow = 2 * ( mLow - g_FirstQuarter );
mHigh = 2 * ( mHigh - g_FirstQuarter ) + 1;
mBuffer = 2 * ( mBuffer - g_FirstQuarter ) + GetBit();
```

5.6 Example decoder

In the next example we decode the sequence that has been encoded in the last one. The input for the decoder is 00010101001101111. The first 7 bits of this sequence are loaded into mBuffer. The next bits are omitted in the table to improve readability. Note that even on E3 scalings the buffer is updated although no bits would be sent in the encoder at this stage. We had to cut off some column names: St is mStep, Sy is Sym and Sc is mScale.

St	mBuffer	l_c	h_c	Sy	mLow	mHigh	Bits	E1/2-mLow	E1/2-mHigh	Sc	E3-mLow	E3-mHigh
16	0001010[10]	0	2	a	0000000[0]	0011111[31]	00	0000000[0]	1111111[127]	0		
16	0101010[42]	2	3	b	0100000[32]	0101111[47]	010	0000000[0]	1111111[127]	0		
16	1010011[83]	3	6	c	0110000[48]	1011111[95]				1	0100000[32]	1111111[127]
12	0100110[38]	3	6	c	1000100[68]	1100111[103]	1$\underline{0}$			0		
9	1001101[77]	7	8	e	1001111[71]	1001111[79]	100	0111000[56]	1111111[127]	0		
9	1101111[111]	6	7	d	1101110[110]	1110110[118]	11	0111000[56]	1011011[91]	1	0110000[48]	1110111[119]
9	1110000[112]	0	2	a	0110000[48]	1000001[65]				3	0000000[0]	1000111[71]
9	1000000[64]	3	6	c	0011011[27]	0110101[53]	0$\underline{111}$	0110110[54]	1101011[107]	0		

Table 6: Example of scaling functions in the decoder

6 Ranges

6.1 Interval size

Since all necessary methods have been presented by now, it should be clear that the values mLow and mHigh can fall into two ranges when one iteration by the encoder or decoder is finished:

- $mLow < FirstQuarter < Half \leq mHigh$,
- $mLow < Half < ThirdQuarter \leq mHigh$.

This interval contains at least one complete quarter. More is possible but not guaranteed.

The calculation of mStep involves a division of the interval size by total. If total is larger than the interval, this integer division results in 0. The algorithm cannot proceed with this value, so the model has to assure that total stays always below the minimum guaranteed size of the interval, in our case one quarter of the base range. Since we use 31 bits in the examples, a quarter equals to 29 bits, sufficing for models with less than 2^{29} symbols (=512 Mbyte at 1 byte/symbol).

6.2 Alternative calculation

Literature[11] sometimes mentions another method to calculate the bounds. In our algorithm the step size is computed first, followed by a multiplication with the cumulative frequency counts of the model. Sometimes this might result in quite large unused intervals:

Let the interval be of the size 7 and the model returns a value of 4 for total. Then the step size calculation results in `7 / 4 = 1` because of the integer arithmetic. This way the upper bound mHigh will not equal the previous upper bound when the last symbol is encoded (high_count equals 4). Instead it is cut down to 4, hence almost one half of the interval remains unused. To circumvent this limitation one can exchange the order of arithmetic operations:

```
range = mHigh - mLow + 1;
mHigh = mLow + ( high_count * range ) / total;
mLow = mLow + ( low_count * range ) / total;
```

Using this method results in mHigh $(4*7)/4 = 28/4 = 7$, so one can use the whole interval. However, the new order provokes overflows due to the multiplication. Let the range be $[0,15)$ (4 bits). The alternative method would result in $4*7 \equiv 12\ (mod\ 16)$ [12], an unusable value for further calculations. Using our method results in $7/4 \equiv 1\ (mod\ 16)$ respective $1*4 \equiv 4\ (mod\ 16)$ which is the expected value. To run the encoder on 32 bit registers, one has to limit the width of the factors: $\lceil ld(a*b+1)\rceil \leq \lceil ld(a+1)\rceil + \lceil ld(b+1)\rceil$.

Since total must not be larger than the minimal interval available (a quarter of the base interval), it follows that

[11] See [BCW90], chapter 5.2.5, page 118.

[12] $4*7 = 0100*0111 = 0010*1110 = 0001*1100 = 1100 = 12$

$$\begin{aligned} ld(total) &\overset{!}{\leq} ld(range) - 2 \ , \\ ld(total) + ld(range) &\overset{!}{\leq} ld(register) \ . \end{aligned}$$

This means in practice that one is limited to 17 bits respectively 15 bits for `total`. Because of the lower precision and the additional division, this alternative method is usually less efficient than the method presented here.

7 Summary of encoder and decoder implementation

En- and decoder can be pooled in one class. The only public methods are those required by users of the en-/decoder, i.e. `Encode`, internal functions like `GetBit` can be private. Static variables can be implemented as member variables.

7.1 Encoder

The encoder can be implemented with the following interface:

```
void Encode( const unsigned int low_count,
     const unsigned int high_count,
     const unsigned int total );

void EncodeFinish();
```

`EncodeFinish()` terminates the code correctly. At first it has to be ensured that the following bits determine a value inside the final interval. Since we know that the interval always contains at least one quarter, we can simply use the lower bound of that quarter. There are two cases:

1. second quarter

   $$mLow < FirstQuarter < Half \leq mHigh \ .$$

   It is sufficient to store a 0 followed by a 1. That means that we select the lower half first, followed by the upper. Since the decoder adds 0s to the input stream automatically at the end of the stored file, this marks the lower bound of the second quarter. If there is an unhandled E3 scaling, one also has to add `mScale` 1 bits. One can combine this with the last 1 to a loop over `mScale+1` bits.

2. third quarter

   $$mLow < Half < ThirdQuarter \leq mHigh \ .$$

   The second case is a bit easier to encode: One would have to write a 1 followed by `mScale+1` 0 bits, but because these are added automatically, terminating with a 1 is sufficient. Therefore no loop is required.

```
if( mLow < g_FirstQuarter ) // mLow < FirstQuarter < Half <= mHigh
{
    SetBit( 0 );

    for( int i=0; i<mScale+1; i++ ) // 1 + e3 scaling
        SetBit(1);
}
else // mLow < Half < ThirdQuarter <= mHigh
{
    SetBit( 1 ); // decoder adds zeros automatically
}
```

7.2 Decoding

The decoder consists of the following three methods:

```
void DecodeStart();

unsigned int DecodeTarget( const unsigned int total );

void Decode( const unsigned int low_count,
     const unsigned int high_count );
```

`DecodeStart()` initializes the buffer by reading the first bits of the encoded input sequence.

```
for( int i=0; i<31; i++ ) // only use the last 31 bits
    mBuffer = ( mBuffer << 1 ) | GetBit();
```

There are no further functions needed and the presentation of encoding and decoding is finished.

We showed that overflows in integer arithmetics can be circumvented using E1, E2 and E3 scaling. A positive side effect is that one can send already the stored bits in sequential data transmissions like remote data transfer. Since the decoder takes only takes into account the bits found in the buffer, it can start decoding as soon as 31 bits are received. Note that errors in the encoded stream corrupt the whole transmission. One has to split the message or add more redundancy to get a robust implementation.

7.3 Termination of the decoding process

Since the bit sequence does not imply an end of the encoded message, one has to add additional information.

The simplest way is to add a file header that contains the length of the file. A disadvantage is that one can only encode files of a fixed length or one has to have random access on the output file. Both is not available for example in fax machines that use special end symbols instead. This symbol is encoded using the *minimal* probability and must not appear in the regular data stream. The decoder terminates as soon as this symbol is read.

The following chapter provides a closer look to the efficiency of arithmetic coding and gives a comparison with Huffman coding.

8 Efficiency

8.1 Looking at the efficiency

In chapter 3.6 we demonstrated, that a sequence x cannot be stored using less than $l(x)$ space without any loss. From that we can derive the *average* length of an Arithmetic Code for a sequence $S^{(m)}$ of length m:

$$\begin{aligned} l_{A^{(m)}} &= \sum_x P_M(x)\ l(x) && (10)\\ &= \sum_x P_M(x)\left[\lceil ld\ \frac{1}{P_M(x)}\rceil + 1\right] && (11)\\ &\leq \sum_x P_M(x)\left[ld\ \frac{1}{P_M(x)} + 1 + 1\right] && (12)\\ &= -\sum_x P_M(x)\ \ ld\ P_M(x) + 2\sum_x P_M(x) && (13)\\ &= H_M(S^{(m)}) + 2. && (14) \end{aligned}$$

And since we already know that the average length is always greater or equal to the entropy, it turns out that

$$H_M(S^{(m)}) \leq l_{A^{(m)}} \leq H_M(S^{(m)}) + 2. \qquad (15)$$

The average length per symbol l_A, also known as *compression ratio* of the Arithmetic Code, is $l_A = \frac{l_{A^{(m)}}}{m}$. So we get the following bounds for l_A:

$$\frac{H_M(S^{(m)})}{m} \leq l_A \leq \frac{H_M(S^{(m)})}{m} + \frac{2}{m}. \qquad (16)$$

Also we know that the entropy of the sequence is nothing but the length of the sequence times the average entropy of every symbol:[13]

$$H_M(S^{(m)}) = m \cdot H_M(x) \qquad (17)$$

For the bounds for l_A this means

$$H_M(x) \leq l_A \leq H_M(x) + \frac{2}{m}. \qquad (18)$$

By examining this comparison one can easily see that he compression ratio l_A is guaranteed to come close to the entropy, which itself is just determined by the model M. This is the desired effect.

8.2 Comparison to Huffman Coding

After having pointed out the efficiency of Arithmetic Coding in the last subsection, we now want to compare this efficiency to the one of the well known Huffman Code. Let us recall example 3.6.1. The average length of the code can be calculated as

$$\begin{aligned} l &= 0,5 \cdot 2 + 0,25 \cdot 3 + 0,125 \cdot 4 + 0,125 \cdot 4\\ &= 2,75\ [bits/symbol]\ . \end{aligned}$$

[13] Proof in [Say00] p.50

But the entropy of this sequence is rather:

$$\begin{aligned}
H_M(x) &= \sum_{i=1}^{4} P(a_i)\ \ ld\ \frac{1}{P_M(a_i)} \\
H_M(x) &= -\left(\sum_{i=1}^{4} P(a_i)\ \ ld\ P_M(a_i) \right) \\
&= -\left(\frac{1}{4} \cdot\ ld\ \frac{1}{2} + \frac{1}{4} \cdot\ ld\ \frac{1}{4} + \frac{1}{4} \cdot\ ld\ \frac{1}{8} + \frac{1}{4} \cdot\ ld\ \frac{1}{8} \right) \\
&= -\left(\frac{1}{4} \cdot (-1) + \frac{1}{4} \cdot (-2) + \frac{1}{4} \cdot (-3) + \frac{1}{4} \cdot (-3) \right) \\
&= 2,25.
\end{aligned}$$

So it turns out that the length of the code the *symbolwise* Arithmetic Coding produces is here *not* very close to the entropy. And even worse: If one encoded this sequence using Huffman Coding, one would achieve the entropy completely. Why is that? That is simply due to the fact that Huffman Coding is *ideal* if and only if one can assign whole bits for the single probabilities (because the constraint of Huffman Coding is that it cannot use fraction of bits). And this is here obviously the case because because all probabilities are (negative) powers of 2. However, exactly this is almost never the case in practical use - but unfortunately does not prevent many people from using such arguments as justification for Huffman.[14] Apart from that, Arithmetic Coding is not even worse is such cases. However, obviously it cannot perform any better either, because the lower bound is already achieved by Huffman. Another common assumption in comparisons of efficiency is that the sequence of symbols is independent from its context. Also this will actually never be the case for real life data sources. However, use of this assumption leads to much easier equations, which fortunately are not too far from reality again. Equation (17) for instance uses this precondition. Now one can easily see that instead of using the comparison (18) one could also work with (16). However, the latter is just unnecessarily complicated and differs just by an unimportant factor. One can proof that the efficiency of the Huffman Code is constraint as follows[15]:

$$H_M(S) \leq l_S \leq H_M(S) + 1\ . \tag{19}$$

For *Extended Huffman*, which is a special version of Huffman Coding, merging b symbols together to longer, single symbols, the efficiency rises to

$$H_M(S) \leq l_S \leq H_M(S) + \frac{1}{b}\ . \tag{20}$$

This is obviously more efficient for non-utopian sequences ($\exists x \in S : P(x) \neq 2^n\ \forall n \in \mathbb{N}$). If one now considers b approaching m and compares this with equation (18), one could come to the conclusion that Huffman Coding here has an advantage over Arithmetic Coding, although this benefit shrinks with raising length m of the sequence. However, this property is in real life not valid because one must take with into account that b cannot be chosen arbitrarily big. Let us consider working over an alphabet of length k and to group b symbols together then we get a codebook size of k^b. For plausible values of $k = 16$ and $b = 20$ this already leads to the value 16^{20}, which is way too big for every known RAM at the current time. So b is constrained by simple physics, while the length of the sequence m increases more and more. So in a practical view, Arithmetic Coding has also here its advantages.

[14] See also [Say00] ch. 4.5.

[15] Also see [Say00] ch. 3.2.3.

Another probable benefit of Arithmetic Coding depends on the data source. One can show that Huffman Coding never overcomes a compression ratio of $(0,086 + P_{max}) \cdot H_M(S)$ for an arbitrary sequence S with P_{max} being the largest of all occurring symbol probabilities[16]. Obviously, for large alphabets it will turn out that one achieves a relatively small value for P_{max}, leading to better results for the Huffman Code. This gives indeed a good justification for such a code on large alphabets. Compared to that, for small alphabets, which oppositely lead to bigger probabilities, Arithmetic Coding can win the race again. Applications using such small alphabet are for instance the compression standards *G3* and *G4*, which are used for fax transmission. Here we have a binary alphabet (containing two symbols, one for black and one for white) and the probability for a white pixel is usually very high. This leads to a value for P_{max} of nearly 1, which disqualifies Huffman Coding and gives us Arithmetic Coding as first choice.

Considering practical results [Can], it turns out that Arithmetic Coding is a small step ahead for most of the real life data sources. That is due to the fact that Huffman Coding is really just optimal for the almost utopian case that all symbol probabilities are powers of two because in this case the Huffman tree has minimal depth. However, since this is almost never the case, the Huffman Coder is usually forced to assign whole numbers of bits for symbols where an Arithmetic Coder could assign fractions of bits at the same time.

Another benefit of Arithmetic Coding, which we will not investigate any further in this paper, is that it can be adapted to work with various probability models. As we saw in previous chapters, one has just to attach an appropriate optimized model for every data source. The basic coding / decoding algorithm remains unchanged, so that implementation of multiple codecs is relatively straightforward. This is especially an advantage if one considers *adaptive* models, which require complex changes of the tree structure using the Huffman algorithm.

We will now explain such adaptive models in further detail, because compared to the previously used static models, they are usually much more powerful.

[16] [Say00] p.37f

9 Alternative models

In previous chapters we used the cumulative function

$$K(a_k) = \sum_{i=1}^{k} P_M(a_i)$$

to code the symbol a_k, being the k-th symbol of the alphabet A. In reality, the probabilities $P_M(a_i)$, $(i = 1, \dots, |A|)$ are therefore retrieved from the model M [17]. However, until now, we have withheld if this model is capable of determining the probability of a symbol a_k in a sequence S at all. And if it is, how does it work? We will now try to answer these questions.

First of all, we want to note that the entropy $H_M(S)$ is depended on the model M by definition. Therefore, regardless how good or bad our model is, the Arithmetic Coder always achieves the best possible result (neglecting some few bits of overhead). However this lower bound $\big($recall equation (18)$\big)$ can still be lowered further using appropriate models.

9.1 Order-n models

Hitherto we considered all symbols as being independent in a stochastic sense. However, it is actually quite common that probabilities change dependent on the current context. In German texts for example the average probability of the letter 'u' is approximately 4.35%. But if one considers the predecessor being a 'q', the probability for seeing a 'u' increases to almost 100%.

Models which take the context of a symbol with into account are called ORDER-N MODELS, where N stands for the size of the context. So for example an Order-3 model will always return the probability in relation to the last 3 symbols seen so far.

9.2 Adaptive Models

Most implementations are developed for various *different* data sources. This means that usually the exact probability distribution of the data source is unknown. Also it might not always be possible to simply count the occurring symbols. Just consider a fax transmission: The transmission shall already begin when the first page becomes read and the rest of the document(s) and its symbol probabilities are still unknown. So the only useful thing one can do is performing an estimation.

And now it seems obvious that this estimation must be *adapted* to probabilities of the symbols which have already been read by the current position. That is why in this case we speak of an *adaptive model*. Let us have a look at the following example:

9.2.1 Example

As an easy demonstration we choose an *adaptive order-0 model*, where *order-0* means that our model always considers the probability of just the symbol, without any context.

To achieve that, it is sufficient enough to define an array K at the beginning, which has the size of the cardinality of the alphabet Σ. All array values become initialized with the value 0. Now, before

[17] Depending on the implementation, the model may also pass the bound $K(a_k)$ directly.

s	K[a]	K[b]	K[c]	K[d]	z	$P_M^{(z)}(a)$	$P_M^{(z)}(b)$	$P_M^{(z)}(c)$	$P_M^{(z)}(d)$
a	1	0	0	0	1	1	0	0	0
b	1	1	0	0	2	$1/2$	$1/2$	0	0
a	2	1	0	0	3	$2/3$	$1/3$	0	0
d	2	1	0	1	4	$1/2$	$1/4$	0	$1/4$

Table 7: Function of an adaptive order-0 model

each coding step, a symbol s is passed from the input stream to the model and this increments the appropriate array entry as well as the absolute symbol counter z. Afterwards the probabilities are redistributed using the assignment

$$P_M^{(z)}(s) = \frac{K[s]}{z} \ .$$

Let us for example consider the following alphabet

$$A = a, b, c, d$$

and encode the sequence *abad*. Table 7 gives the calculation results for this model. It turns out that the probability values which are assigned after the last symbol was read are equal to the *real* probabilities of the symbols. So it is obvious that the calculated probabilities come pretty close to the real ones for long sequences.

Since the initialization is known in advance and every assignment is done step by step after reading each symbol, the decoder can retrace these steps without any problems. It just updates the model in the very same way. This leads to the advantage that no updated probabilities must be sent over the data stream. They can just be generated from the data which the decoder receives anyway.

9.3 Additional models

For some applications, such as mixed files which consist of very distinct data partitions with different probability distributions, it might be useful to detect rapid changes of the probability distribution and - once such a jump is detected - to reinitialize the array K of the model e.g. with a uniform distribution. This usually leads to a better compression for the following part of the sequence because the model can adapt much faster.

Obviously one can imagine a lot of different additional models which might be better for appropriate data sources. However, we will not go any further into this theme because the implementation of the models is actually independent from the mechanism of Arithmetic Coding and there is already a lot of literature around about stochastic modeling. [Dat] might give some useful hints.

10 Conclusion

After all these consideration, let us now recap and check if we have achieved what we promised in the beginning. With Arithmetic Coding, we have described a coding method, which is suitable for data compression. This was proven by showing that the requirements of a bijective encoding are met. Implementation can nowadays employ integer as well as floating point arithmetics. We have seen, that Arithmetic Coding can work sequentially, encoding symbol per symbol and thus is able to send already encoded parts of a message before it is fully known. This property is exploited when applying the three scaling functions, which enlarge the working interval in such a way that overflows do not occur and even finite arithmetics suffice. Also we showed up the bounds of the efficiency of general encoding and noted, that the average code length for any symbol of an input sequence approaches closer and closer to the model-dependent entropy with raising length of the input sequence. We also demonstrated in what cases Arithmetic Coding is especially efficient and in what cases it is only as efficient as Huffman Coding. We noted that the compression ratio that can be reached by any encoder under a given model is actually bounded by the quality of that model. Here we also realized another advantage of Arithmetic Coding, since it allows the easy exchange of statistical models, that might be optimized for certain input data.

We conclude that we have achieved our goal. In the end of this paper we now want to share some thoughts about fields of improvement and applied techniques.

10.1 Remember: Compression has its bounds

Although Arithmetic Coding has been established and optimized over the past 10 to 20 years, every now and then a new variation appears. Interested readers might want to observe the newsgroup *comp.compression* for new techniques and further insights. However, beware: Sometimes people claim having invented an outstanding algorithm that performs several times better than anything seen before. Frequently they are exaggerating, sometimes simply ignoring that Arithmetic Codingis about *lossless* coding. We know for sure that the Shannon theorem [WS49] guarantees that compression below the entropy of the source is impossible. One can remove as much redundancy from one's data as one likes, but entropy is proven to be a hard limit.

10.2 Methods of Optimization

However one can optimize one's algorithms in at least two dimensions: memory usage and speed.

10.2.1 Memory Usage

Arithmetic Coding is almost optimal in terms of memory usage. It uses only a constant amount of memory for simple models (elaborate models might take some more, but usually less than linear). Furthermore it generates a code that cannot be compressed any further. Note that this code depends on the model: $H(S) \leq H_M(S) \leq |Code(S)|$. We have to differentiate between the natural entropy of the source sequence, $H(S)$, which represents the mathematical lower bound, and the bound that is set by our model, $H_M(S)$. Arithmetic Codingreaches $H_M(S)$, but that might be far from perfect if one's model is incapable of representing the input data very well.

Since input data is not predictable in a general way, one will have to find a model that works for one specific application context. Arithmetic Coding allows a modular design so that the coder can interact with different models, even switching between them while coding. Quite a lot of models have been developed, one of the most popular model families is *PPM (Prediction with partial match)*. They are quite efficient due to varying context length, but most of the advanced ones lack a sound mathematical background. Visit [Dat] for further information.

10.2.2 Speed

The speed of Arithmetic Coding coders has been improved over the years. Integer implementations are common, but with improving floating point power of modern CPUs this way might become an alternative. We showed in 3.6 that an implementation based on floating point arithmetic is possible. A very efficient integer implementation is the *Range Coder* [Mar79], [Cam99]. It performs scaling byte-wise, thus eliminating large parts of the bit-fiddling which is a major performance problem on current CPUs. Speed improvements up to 50% are reported whereas the code size increases only by 0.01%. These numbers have to be seen with caution since they only reflect the performance of the coder, not of the model. However, the bottleneck of todays implementations of Arithmetic Coding is almost always the model. As usual one can get an overview about all this on [Dat].

As one can see, the most interesting research fields in the context of Arithmetic Coding are the models. Code size, memory usage and speed depends mainly on them whereas a well implemented coder can be seen as a minor task, especially since Arithmetic Codingitself is documented very well.

A A reference implementation in C++

also available at: *http://ac.bodden.de*

This implementation should present the whole algorithm in a non-ambiguous way to answer any open questions regarding implementation details. We use a simple adaptive order 0 model as describes in chapter 9.2. Therefore the compression ratio is quite low, but one can exchange the model anytime, just derive a new one from the base class `ModelI`.

A.1 Arithmetic Coder (Header)

```
#ifndef __ARITHMETICCODERC_H__
#define __ARITHMETICCODERC_H__

#include <fstream>
using namespace std;

class ArithmeticCoderC
{
public:
  ArithmeticCoderC();

  void SetFile( fstream *file );

  void Encode( const unsigned int low_count,
               const unsigned int high_count,
               const unsigned int total );
  void EncodeFinish();

  void DecodeStart();
  unsigned int DecodeTarget( const unsigned int total );
  void Decode( const unsigned int low_count,
               const unsigned int high_count );

protected:
  // bit operations
  void SetBit( const unsigned char bit );
  void SetBitFlush();
  unsigned char GetBit();

  unsigned char mBitBuffer;
  unsigned char mBitCount;

  // in-/output stream
  fstream *mFile;

  // encoder & decoder
  unsigned int mLow;
  unsigned int mHigh;
  unsigned int mStep;
  unsigned int mScale;

  // decoder
  unsigned int mBuffer;
};

#endif
```

A.2 Arithmetic Coder

```
#include "ArithmeticCoderC.h"
#include "tools.h"

// constants to split the number space of 32 bit integers
// most significant bit kept free to prevent overflows
const unsigned int g_FirstQuarter = 0x20000000;
const unsigned int g_ThirdQuarter = 0x60000000;
const unsigned int g_Half         = 0x40000000;

ArithmeticCoderC::ArithmeticCoderC()
{
  mBitCount = 0;
  mBitBuffer = 0;

  mLow = 0;
  mHigh = 0x7FFFFFFF; // just work with least significant 31 bits
  mScale = 0;

  mBuffer = 0;
  mStep = 0;
}

void ArithmeticCoderC::SetFile( fstream *file )
{
  mFile = file;
}

void ArithmeticCoderC::SetBit( const unsigned char bit )
{
  // add bit to the buffer
  mBitBuffer = (mBitBuffer << 1) | bit;
  mBitCount++;

  if(mBitCount == 8) // buffer full
  {
    // write
    mFile->write(reinterpret_cast<char*>(&mBitBuffer),sizeof(mBitBuffer));
    mBitCount = 0;
  }
}

void ArithmeticCoderC::SetBitFlush()
{
  // fill buffer with 0 up to the next byte
  while( mBitCount != 0 )
    SetBit( 0 );
}

unsigned char ArithmeticCoderC::GetBit()
{
  if(mBitCount == 0) // buffer empty
  {
    if( !( mFile->eof() ) ) // file read completely?
      mFile->read(reinterpret_cast<char*>(&mBitBuffer),sizeof(mBitBuffer));
    else
      mBitBuffer = 0; // append zeros

    mBitCount = 8;
  }

  // extract bit from buffer
  unsigned char bit = mBitBuffer >> 7;
  mBitBuffer <<= 1;
  mBitCount--;

  return bit;
}

void ArithmeticCoderC::Encode( const unsigned int low_count,
                               const unsigned int high_count,
                               const unsigned int total )
// total < 2ˆ29
{
  // partition number space into single steps
  mStep = ( mHigh - mLow + 1 ) / total; // interval open at the top => +1

  // update upper bound
  mHigh = mLow + mStep * high_count - 1; // interval open at the top => -1

  // update lower bound
  mLow = mLow + mStep * low_count;

  // apply e1/e2 scaling
  while( ( mHigh < g_Half ) || ( mLow >= g_Half ) )
    {
      if( mHigh < g_Half )
      {
        SetBit( 0 );
        mLow = mLow * 2;
        mHigh = mHigh * 2 + 1;

        // perform e3 scalings
        for(; mScale > 0; mScale-- )
          SetBit( 1 );
      }
      else if( mLow >= g_Half )
      {
        SetBit( 1 );
        mLow = 2 * ( mLow - g_Half );
        mHigh = 2 * ( mHigh - g_Half ) + 1;

        // perform e3 scalings
        for(; mScale > 0; mScale-- )
          SetBit( 0 );
      }
    }

  // e3
  while( ( g_FirstQuarter <= mLow ) && ( mHigh < g_ThirdQuarter ) )
  {
    // keep necessary e3 scalings in mind
    mScale++;
    mLow = 2 * ( mLow - g_FirstQuarter );
    mHigh = 2 * ( mHigh - g_FirstQuarter ) + 1;
  }
}

void ArithmeticCoderC::EncodeFinish()
{
  // There are two possibilities of how mLow and mHigh can be distributed,
  // which means that two bits are enough to distinguish them.

  if( mLow < g_FirstQuarter ) // mLow < FirstQuarter < Half <= mHigh
  {
    SetBit( 0 );

    for( int i=0; i<mScale+1; i++ ) // perform e3-scaling
      SetBit(1);
  }
  else // mLow < Half < ThirdQuarter <= mHigh
  {
    SetBit( 1 ); // zeros added automatically by the decoder; no need to send them
  }

  // empty the output buffer
  SetBitFlush();
}

void ArithmeticCoderC::DecodeStart()
{
  // Fill buffer with bits from the input stream
  for( int i=0; i<31; i++ ) // just use the 31 least significant bits
    mBuffer = ( mBuffer << 1 ) | GetBit();
}

unsigned int ArithmeticCoderC::DecodeTarget( const unsigned int total )
// total < 2ˆ29
{
  // split number space into single steps
  mStep = ( mHigh - mLow + 1 ) / total; // interval open at the top => +1

  // return current value
  return ( mBuffer - mLow ) / mStep;
}

void ArithmeticCoderC::Decode( const unsigned int low_count,
                               const unsigned int high_count )
{
  // update upper bound
  mHigh = mLow + mStep * high_count - 1; // interval open at the top => -1

  // update lower bound
  mLow = mLow + mStep * low_count;

  // e1/e2 scaling
  while( ( mHigh < g_Half ) || ( mLow >= g_Half ) )
    {
      if( mHigh < g_Half )
      {
        mLow = mLow * 2;
        mHigh = mHigh * 2 + 1;
        mBuffer = 2 * mBuffer + GetBit();
      }
      else if( mLow >= g_Half )
      {
        mLow = 2 * ( mLow - g_Half );
        mHigh = 2 * ( mHigh - g_Half ) + 1;
        mBuffer = 2 * ( mBuffer - g_Half ) + GetBit();
      }
      mScale = 0;
    }

  // e3 scaling
  while( ( g_FirstQuarter <= mLow ) && ( mHigh < g_ThirdQuarter ) )
  {
    mScale++;
    mLow = 2 * ( mLow - g_FirstQuarter );
    mHigh = 2 * ( mHigh - g_FirstQuarter ) + 1;
    mBuffer = 2 * ( mBuffer - g_FirstQuarter ) + GetBit();
  }
}
```

A.3 Model Base Class (Header)

```
#ifndef __MODELI_H__
#define __MODELI_H__

#include "ArithmeticCoderC.h"

enum ModeE
{
  MODE_ENCODE = 0,
  MODE_DECODE
};

class ModelI
{
public:
  void Process( fstream *source, fstream *target, ModeE mode );

protected:
  virtual void Encode() = 0;
  virtual void Decode() = 0;

  ArithmeticCoderC mAC;
  fstream *mSource;
  fstream *mTarget;
};

#endif
```

A.4 Model Base Class

```
#include "ModelI.h"

void ModelI::Process( fstream *source, fstream *target, ModeE mode )
```

```
{
  mSource = source;
  mTarget = target;

  if( mode == MODE_ENCODE )
  {
    mAC.SetFile( mTarget );

    // encode
    Encode();

    mAC.EncodeFinish();
  }
  else // MODE_DECODE
  {
    mAC.SetFile( mSource );

    mAC.DecodeStart();

    // decode
    Decode();
  }
};
```

A.5 Model Order 0 (Header)

```
#ifndef __MODELORDER0C_H__
#define __MODELORDER0C_H__

#include "ModelI.h"

class ModelOrder0C : public ModelI
{
public:
  ModelOrder0C();

protected:
  void Encode();
  void Decode();

  unsigned int mCumCount[ 257 ];
  unsigned int mTotal;
};

#endif
```

A.6 Model Order 0

```
#include "ModelOrder0C.h"

ModelOrder0C::ModelOrder0C()
{
  // initialize probabilities with 1
  mTotal = 257; // 256 + escape symbol for termination
  for( unsigned int i=0; i<257; i++ )
    mCumCount[i] = 1;
}

void ModelOrder0C::Encode()
{

  while( !mSource->eof() )
  {
    unsigned char symbol;

    // read symbol
    mSource->read( reinterpret_cast<char*>(&symbol), sizeof( symbol ) );

    if( !mSource->eof() )
    {
      // cumulate frequencies
      unsigned int low_count = 0;
      for( unsigned char j=0; j<symbol; j++ )
        low_count += mCumCount[j];

      // encode symbol
      mAC.Encode( low_count, low_count + mCumCount[j], mTotal );

      // update model
      mCumCount[ symbol ]++;
      mTotal++;
    }
  }

  // write escape symbol for termination
  mAC.Encode( mTotal-1, mTotal, mTotal );
}

void ModelOrder0C::Decode()
{
  unsigned int symbol;

  do
  {
    unsigned int value;

    // read value
    value = mAC.DecodeTarget( mTotal );

    unsigned int low_count = 0;

    // determine symbol
    for( symbol=0; low_count + mCumCount[symbol] <= value; symbol++ )
      low_count += mCumCount[symbol];

    // write symbol
    if( symbol < 256 )
      mTarget->write( reinterpret_cast<char*>(&symbol), sizeof( char ) );

    // adapt decoder
    mAC.Decode( low_count, low_count + mCumCount[ symbol ] );

    // update model
    mCumCount[ symbol ]++;
    mTotal++;
  }
  while( symbol != 256 ); // until termination symbol read
}
```

A.7 Tools

```
#ifndef __TOOLS_H__
#define __TOOLS_H__

int inline min( int a, int b )
{
  return a<b?a:b;
};

#endif
```

A.8 Main

```
#include <iostream>
#include <fstream>
using namespace std;

#include "ModelOrder0C.h"

// signature: "ACMC" (0x434D4341, intel byte order)
// (magic number for recognition of encoded files)
```

```
const int g_Signature = 0x434D4341;

int __cdecl main(int argc, char *argv[])
{
  cout << "Arithmetic Coding" << endl;

  if( argc != 3 )
  {
    cout << "Syntax: AC source target" << endl;
    return 1;
  }

  fstream source, target;
  ModelI* model;

  // choose model, here just order-0
  model = new ModelOrder0C;

  source.open( argv[1], ios::in | ios::binary );
  target.open( argv[2], ios::out | ios::binary );

  if( !source.is_open() )
  {
    cout << "Cannot open input stream";
    return 2;
  }
  if( !target.is_open() )
  {
    cout << "Cannot open output stream";
    return 3;
  }

  unsigned int signature;
  source.read(reinterpret_cast<char*>(&signature),sizeof(signature));
  if( signature == g_Signature )
  {
    cout << "Decoding " << argv[1] << " to " << argv[2] << endl;
    model->Process( &source, &target, MODE_DECODE );
  }
  else
  {
    cout << "Encoding " << argv[1] << " to " << argv[2] << endl;
    source.seekg( 0, ios::beg );
    target.write( reinterpret_cast<const char*>(&g_Signature),
                  sizeof(g_Signature) );
    model->Process( &source, &target, MODE_ENCODE );
  }
  source.close();
  target.close();

  return 0;
}
```

Index

References

[BCK02] Eric Bodden, Malte Clasen, and Joachim Kneis. Arithmetic Coding in a nutshell. In *Proseminar Datenkompression 2001*. University of Technology Aachen, 2002. English version available: Arithmetic coding, introduction, source code, example, applications.

[BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text compression*. Prentice-Hall, Inc., 1990.

[Beu94] A. Beutelspacher. *Cryptology*. Mathematical Association of America, 1994.

[Cam99] Arturo Campos. Range coder implementation, 1999.
http://www.arturocampos.com/ac_range.html.

[Can] The canterbury corpus.
http://corpus.canterbury.ac.nz/summary.html.

[Dat] The data compression library.
http://dogma.net/DataCompression/ArithmeticCoding.shtml.

[Fan61] R. Fano. *Transmission of Information*. MIT Press, Cambridge, 1961.

[Mar79] G. N. N. Martin. Range encoding, an algorithm for removing redundancy from a digitised message. In *Video and Data Recording Conference, Southampton, July 24-27, 1979*, 1979.

[Say00] Khalid Sayood. *Introduction to data compression (2nd ed.)*. Morgan Kaufmann Publishers Inc., 2000.

[WBM94] Ian H. Witten, Timothy C. Bell, and Alistair Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images*. John Wiley & Sons, Inc., 1994.

[WS49] W. Weaver and C.E. Shannon. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, Illinois, 1949. republished in paperback 1963.