MINUT 2026 (8), s. 38–52
ISSN 2719-3063

Wiktor JAROSZ[1]

[1]Student, Faculty of Applied Mathematics, Silesian University of Technology, ul. Kaszubska 23, 44-100 Gliwice

# Arithmetic coding demystified: from theory to practical implementation

**Abstract**. The article discusses the operating principle of arithmetic coding and presents the mathematical theory ensuring its correctness. An important element is also the illustrations, which demonstrate the intuition behind the algorithm. The renormalization mechanism — one of the most crucial parts of arithmetic coding implementation — has been described in detail. The article also goes through an example of message encoding and decoding and discusses the standard method of software implementation.

**Keywords:** information theory, data compression, arithmetic coding

## 1. Preface

Although many do not realize it, lossless data compression accompanies us at every step of our digital lives. Compression algorithms are used during practically every visit to websites. Popular data formats like PNG or PDF contain sophisticated compression methods that make our files take up significantly less disk space than if we stored them in "raw" form. Among dozens of popular lossless compression methods, one stands out for its remarkable mathematical ingenuity, which allows for achieving efficiency extremely close to the theoretical limit. We are speaking, of course, of arithmetic coding.

## 2. Introduction to coding and data compression

The goal of data compression is to reduce the volume of information while preserving its content. In the case of lossless methods, this process must be fully reversible, meaning that after decompression, we obtain exactly the same sequence of bits as the input.

A fundamental model in information theory is the Discrete Memoryless Source (DMS), denoted as a random variable $X$. This variable takes values from a finite alphabet $S = \{s_1, s_2, \ldots, s_n\}$ according to a probability distribution $P = \{p_1, p_2, \ldots, p_n\}$.

Corresponding author: W. Jarosz (wj316047@student.polsl.pl).
Received: 18.03.2026.

The intuition behind effective compression is based on the observation that symbols do not occur in data with equal frequency. Statistical methods, such as Huffman or arithmetic coding, assign shorter bit sequences to frequent symbols (with high $p_i$), and longer ones to rare symbols.

The measure of the "randomness" of symbols from a given source is called Shannon entropy $H(X)$ and is expressed by the formula:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i). \tag{1}$$

The more uncertain the answer to the question "What will the next symbol likely be?", the higher the source entropy. For example, the message *acbcab* has higher entropy than the message *aaaaab*. According to Shannon's source coding theorem [3], entropy determines the theoretical lower bound of the average code length per symbol. No lossless compression algorithm can (on average) encode data using fewer bits per symbol than the source entropy.

# 3. Arithmetic coding

Arithmetic coding is one of the most efficient methods of lossless compression. Many people mistakenly identify Huffman coding [1] with the compression limit. In practice, however, it has a serious limitation — each symbol of the alphabet is encoded using an integer number of bits. This means that, in extreme cases, Huffman coding can waste on average almost 1 bit for each symbol. This is particularly evident in the following example:

**Example 1.** Let us compute the entropy and the Huffman code of a source $X$ with alphabet $S = \{a, b\}$ and probability distribution $P(a) = 0.999$ and $P(b) = 0.001$.

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i) = -(0.999 \log_2(0.999) + 0.001 \log_2(0.001)) \approx 0.011 \text{ bits}. \tag{2}$$

An example Huffman code:

$$a \to 0,$$
$$b \to 1.$$

As can be seen, the entropy of the source (the theoretical minimum average length of a single symbol) is only 0.011 bits. However, when using Huffman coding, we are forced to assign one bit to the symbol $a$ (as well as $b$). In this case, a message consisting of $n$ symbols will be encoded using $n$ bits — much more than the theoretical limit of $0.011n$.

This problem is solved by arithmetic coding, which does not assign codewords to individual symbols. Instead, it encodes the entire input sequence as a single number, which statistically allows symbols to be represented using "fractions" of bits.

## 3.1. How arithmetic coding works

The principle of operation of an arithmetic encoder is based on the recursive subdivision of a numerical interval. The initial interval is $[0, 1)$. This interval is divided proportionally to the probabilities of the symbols. From the resulting subintervals, the subinterval corresponding to the currently encoded symbol

is selected. This procedure is repeated until the entire message has been encoded. Figure 1 illustrates the intuition behind the operation of the encoder.

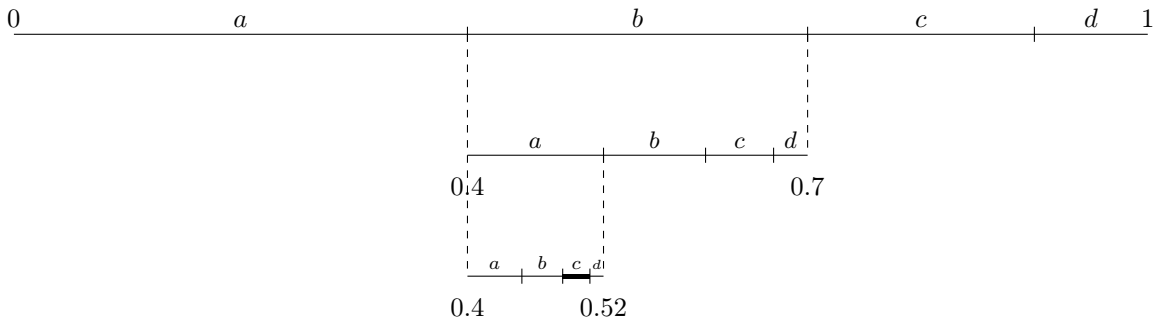



Fig. 1. A graphical representation of the encoding process of the sequence "bac" for a source with probability distribution $0.4, 0.3, 0.2, 0.1$. Each successive level illustrates the physical narrowing of the coding interval. The final code of the message is any number from the interval $[0.484, 0.508)$.

Let $S = \{s_1, s_2, \ldots, s_n\}$ be an alphabet and $P = \{p_1, p_2, \ldots, p_n\}$ be its associated probability distribution. In order to precisely determine the boundaries of the subintervals for each symbol, it is necessary to introduce the definition of cumulative probability.

**Definition 1.** *Let $C_k$ be the sum of the probabilities of the symbols $s_1, s_2, \ldots, s_{k-1}$. We call $C_k$ the cumulative probability of the k-th symbol of the alphabet S:*

$$C_k = \sum_{i=1}^{k-1} p_i \quad \textit{for } k > 1, \quad \textit{and } C_1 = 0. \tag{3}$$

**Example 2.** For a source with probability distribution $P = \{0.4, 0.3, 0.2, 0.1\}$, the cumulative probabilities are:

$$\begin{aligned} C_1 &= 0, \\ C_2 &= 0.4, \\ C_3 &= 0.7, \\ C_4 &= 0.9. \end{aligned}$$

The algorithm maintains two state variables: $L$ (low), representing the lower bound of the current interval, and $H$ (high), representing the upper bound. Initially, $L = 0$ and $H = 1$, so the interval is $[0, 1)$. The process of encoding the next symbol consists of narrowing the interval $[L, H)$ to a new interval corresponding to that symbol. The encoding operation of the $k$-th symbol is described by the following recursive formulas:

$$\text{width} = H_{old} - L_{old}, \tag{4}$$

$$L_{new} = L_{old} + \text{width} \times C_k, \tag{5}$$

$$H_{new} = L_{old} + \text{width} \times (C_k + p_k). \tag{6}$$

The code of the message is any real number lying in the final interval $[L_n, H_n)$. Each subsequent step of the algorithm causes the interval to become narrower. The narrower the interval becomes, the greater the number of digits (bits) needed to represent a number contained within it. In this way, arithmetic coding "encodes" symbols with high probabilities using a smaller number of bits. Such symbols have a smaller influence on the width of the interval. On the other hand, a symbol with probability 0.01 will shrink the interval by a factor of 100.

Decoding the message proceeds analogously to encoding. The decoder receives the code (a number from the interval) and the source (the alphabet and the probability distribution). It is worth noting that each subsequent interval is contained within the previous one: $[L_n, H_n) \subseteq [L_{n-1}, H_{n-1})$. This means that if the decoder receives the number 0.242, and the initial interval $[0, 1)$ contains a subinterval $[0, 0.4)$ corresponding to the symbol 'a', then the decoder will output the symbol 'a'. Choosing any other subinterval (e.g. $[0.4, 0.7)$) would not allow it to reach the number 0.242.

## 3.2. Termination of encoding

In practice, it is necessary to inform the decoder about the end of the message. Note that the number 0.0 may be the code of the message *a*, but also of *aa, aaa, aaaa, ...*. There are several ways to resolve this ambiguity. One of them is to transmit the number of symbols that the decoder should read. However, this approach works only when the size of the transmitted data is known in advance, which is often impossible in the case of streaming transmission. Instead, one of the most popular implementations of arithmetic coding [4] uses a special EOF symbol (End Of File).

**Definition 2.** *Let # be a symbol denoting the end of the message. Then # $\in S$ with some probability $P(\#)$. The encoder terminates the encoding of the message with the symbol #. Encountering the symbol # by the decoder means that the entire message has already been decoded.*

The symbol # is added to the alphabet $S$ with some low probability $P(\#)$, e.g. 0.1. The probabilities of the remaining symbols must be normalized so that the sum of all probabilities equals 1.

## 3.3. Renormalization

The coding model presented so far assumed the use of a machine capable of operating on real numbers with infinite precision. In reality, however, numerical precision is limited. Since the width of the interval decreases exponentially with each symbol, it is easy to observe that the interval would quickly "disappear". The lower bound $L$ and the upper bound $H$ would become so close to each other that the processor would no longer be able to distinguish them (underflow). The solution to this problem is **renormalization**.

The idea behind renormalization is based on the observation that certain bits of the code are known even before the final interval is computed. If the subinterval of the first symbol is $[0, 0.4)$, then we can be sure that the message code will start with the bit 0. This is because the binary representation of every number in this interval begins with $0.0\ldots$. Analogously, if the selected subinterval is $[0.7, 0.9)$, then we

can be sure that the next bit of the message code will be 1, because the binary representation of every number $0.7 \leq x < 0.9$ begins with $0.1\ldots$. In such a situation, we can write the known bit to the buffer (or send it to the receiver) and "stretch" the interval by shifting all bits to the left.

Formally, after selecting the subinterval $[L, H)$ of the next encoded symbol, we have three cases that require renormalization:

1. The subinterval lies entirely in the lower half of the interval: $[L, H) \subseteq [0, 0.5)$.
2. The subinterval lies entirely in the upper half of the interval: $[L, H) \subseteq [0.5, 1.0)$.
3. The subinterval lies on the boundary between the lower and upper halves of the interval: $[L, H) \subseteq [0.25, 0.75)$.

The first two cases have already been discussed. In the first one, we know that the next bit of the encoded message will be 0. In order to "stretch" the interval, we must write this bit to the buffer and then shift the bits of $L$ and $H$ to the left. As a result, this bit "drops out", and we are left with a wider interval. The bit-shifting operation is achieved by multiplying the numbers by 2. In the second case, we know that the next bit of the message will be 1. However, in order to obtain an interval contained in $[0, 1)$ after shifting, we must first subtract 0.5. This is equivalent to changing the first fractional bit from 1 to 0. We thus obtain the following functions, which will be used respectively in the first and second cases:

$$E_1 : [0, 0.5) \to [0, 1), \qquad E_1(x) = 2x; \tag{7}$$

$$E_2 : [0.5, 1) \to [0, 1), \qquad E_2(x) = 2(x - 0.5). \tag{8}$$

The third case is the most complicated. The value of the first fractional bit of $L$ (0) differs from the value of the first fractional bit of $H$ (1). Therefore, we cannot unambiguously write it to the buffer, because we do not yet know whether it is 0 or 1. At the same time, we cannot remain idle, because this interval may become arbitrarily small within $[0.499\ldots, 0.500\ldots)$. The first step in solving this problem is to observe that the final code number will lie either in $[0.25, 0.5)$ or in $[0.5, 0.75)$. In the first case, the bit to be written will be 0, and in the second case it will be 1. However, we do not yet know which of these situations will occur. We therefore postpone writing the bit and define the following scaling function:

$$E_3 : [0.25, 0.75) \to [0, 1), \qquad E_3(x) = 2(x - 0.25). \tag{9}$$

It is worth noting that the interval $[0.25, 0.5)$ becomes the interval $[0, 0.5)$ after mapping. Analogously, the interval $[0.5, 0.75)$ becomes the interval $[0.5, 1.0)$. Thus, if after selecting the subinterval of the next encoded symbol we end up in the lower half of the interval (case $E_1$), then we will know that before scaling with function $E_3$, we should have "entered" $[0.25, 0.5)$. According to the earlier analysis, this means that the first bit to be written is 0. Similarly, if the subinterval of the next symbol falls into the upper half of the interval (case $E_2$), then we will know that before scaling with $E_3$ we were in $[0.5, 0.75)$, so the correct bit to be written was 1. Suppose that we landed in $[0.25, 0.5)$ before scaling. We cannot write only the bit 0, because this would be equivalent to the subinterval $[0, 0.5)$. Note, however, that the fractional part of every number from the interval $[0.25, 0.5)$ begins with the bits 01. We therefore write these bits to the message code. If, on the other hand, the subinterval turned out to be $[0.5, 0.75)$, we would write 10.

But what if we encounter case $E_3$ more than once? Suppose that we select the subinterval $[0.4, 0.6)$. After the first scaling, we obtain $[0.3, 0.7)$. This interval still lies within $[0.25, 0.75)$, so we scale once more and obtain $[0.1, 0.9)$. At this point, we divide the interval proportionally to the probabilities and select the subinterval of the next encoded symbol. Assume that the selected subinterval is $[0.1, 0.4)$. It lies in the lower half, which, according to the explanation in the previous paragraph, means that before scaling we would have landed in $[0.25, 0.5)$. However, we scaled the interval twice using function $E_3$. The interval $[0.25, 0.5)$ is the interval after the first scaling:

$$\begin{aligned} E_3([L, H)) &= 2[L - 0.25, H - 0.25) \\ &= [2L - 0.5, 2H - 0.5] \\ &= [0.25, 0.5). \end{aligned} \tag{10}$$

We must "go back" one more level up. We thus obtain the boundaries of the interval before the first $E_3$ mapping:

$$\begin{aligned} 2L - 0.5 &= 0.25 \\ 2L &= 0.75 \\ L &= 0.375, \end{aligned} \tag{11}$$

$$\begin{aligned} 2H - 0.5 &= 0.5 \\ 2H &= 1 \\ H &= 0.5. \end{aligned} \tag{12}$$

Recall that we started with the interval $[0.4, 0.6)$. After two scalings with $E_3$, we determined that the final number will lie in $[0.375, 0.5)$, that is, in $[0.4, 0.5)$. Note that the binary representation of the fractional part of every number in this interval begins with 011. We therefore write these bits to the output buffer. If scaling with $E_3$ were necessary three times, then the final subinterval would be $[0.4375, 0.5)$, in which the fractional part of every number begins with the bits 0111. At this point, it is worth noting the structure of these numbers. If case $E_3$ occurs $n$ times in a row, then we write the bit imposed by the last selected subinterval (case $E_1$ or $E_2$), and then write $n$ opposite bits. A formal proof of this statement can be found in Appendix A.

We therefore introduce a new state variable: *counter*. This variable will store the information about how many times we have applied the mapping $E_3$. Each time we write bits to the buffer, we will also write *counter* opposite bits and reset *counter* to zero. We thus formally define the following renormalization rules, which will be executed in a loop:

1. $[L, H) \subseteq [0, 0.5)$: Write 0, then write 1 *counter* times. Set $counter = 0$. Scale the interval using function $E_1$.

2. $[L, H) \subseteq [0.5, 1.0)$: Write 1, then write 0 *counter* times. Set $counter = 0$. Scale the interval using function $E_2$.

3. $[L, H) \subseteq [0.25, 0.75)$: Increment the counter: $counter = counter + 1$. Scale the interval using function $E_3$.

**Remark 1.** It is worth noting that the above conditions do not apply to every interval. An example is the interval $[0.1, 0.9)$. The key here is to understand that renormalization and symbol encoding are two separate processes. The purpose of renormalization is solely to expand the interval in order to avoid problems related to numerical precision. We achieve this by writing bits that are already certain (cases 1-3 above). The "encoding" of a new symbol takes place only when the interval does not require renormalization, for example in the case of $[0.1, 0.9)$. Then we divide the interval into subintervals and select the subinterval corresponding to the next encoded symbol.

#### 3.3.1. Flush sequence

The last encoded symbol is # (EOF). After completing the renormalization loop, we obtain an interval $[L, H)$ that satisfies at least one of the following conditions:

$$L < 0.25 < 0.5 \leq H \tag{13}$$

or

$$L < 0.5 < 0.75 \leq H. \tag{14}$$

Finally, we must choose some number from this interval that will unambiguously lead the decoder to the subinterval #. Most practical implementations use the following rules [4]:

**1.** If $L < 0.25$, send 0, then send 1 $(counter + 1)$ times.

**2.** Otherwise, send 1, then send 0 $(counter + 1)$ times.

Let us consider what these conditions mean. The first one is similar to the situation in which we fall into case $E_1$. We send 0 and then send 1 *counter* times. In this case, we additionally send one more 1. This is necessary in order to narrow the final interval from $[0, 0.5)$ to $[0.25, 0.5)$. Let us see what would happen if we did not do this. Suppose that the interval of # is $[0.3, 0.7)$. Mapping with $E_3$ increases the counter to 1 and scales the interval to $[0.1, 0.9)$. If at this point we were to send only the bits 01, their fractional decimal value would be 0.25. Note that this number lies outside the interval $[0.3, 0.7)$, which would lead the decoder to an incorrect symbol. We therefore must send one additional bit with value 1 in order to obtain the number 0.375.

The second condition is similar to case $E_2$. Symmetrically to the previous condition, here we also send $(counter + 1)$ opposite bits. In this case, the visualization is somewhat simpler, because the transmitted number is always equal to 0.5. According to the second condition, 0.5 must be part of the final interval, which ensures that the decoder unambiguously decodes the symbol #.

### 3.4. Decoding the message

Decoding consists of performing the same steps as those performed by the encoder. The only difference is that instead of generating bits, we read them from an already available stream. The decoder proceeds according to the following algorithm:

1. Set $L = 0$, $H = 1$.

2. Read the first $n$ bits of the message into the variable *value*.

3. Repeat until EOF is decoded:

   3.1. Find the symbol whose subinterval contains *value*.

   3.2. Output this symbol.

   3.3. Update $L$ and $H$ by narrowing them to the boundaries of the found subinterval.

   3.4. Execute the renormalization loop for $L$, $H$, and *value* for as long as the interval requires it. After each renormalization, read the next bit of the message into *value*.

It is worth noting that the current message code (*value*) is scaled together with the interval. If the encoder shifts the bits to the left at this point, the decoder must do the same in order for their operations to remain synchronized. Another important detail is the fact that we operate on only $n$ bits of the code. This is necessary because the computer may not be able to keep the entire message in memory. Instead, after each renormalization (left bit shift), we read the next bit from the stream. A more detailed description of the operation of real implementations can be found in Section 5.

# 4. Examples of calculations

For the purposes of the examples below, we assume that the computer is able to keep full numerical values in memory.

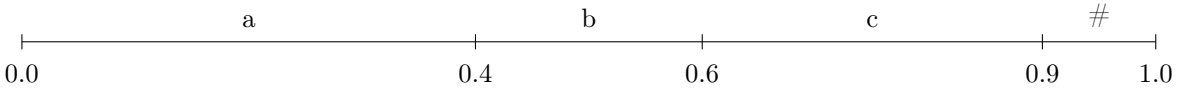
## 4.1. Encoding example

In order to fully understand the operation of the algorithm together with the renormalization mechanism and the flush sequence, let us trace the process of encoding a short message. We assume an alphabet consisting of four symbols $\{a, b, c, \#\}$, for which the following occurrence probabilities are defined: $P(a) = 0.4$, $P(b) = 0.2$, $P(c) = 0.3$, and $P(\#) = 0.1$. The symbol # denotes the end of file (EOF).

Our goal is to encode the message "ba#". The initial state of the encoder is the interval $[0.0, 1.0)$ and the variable $counter = 0$.

**Step 1: Encoding the symbol $b$**

At the beginning, we map the symbols onto the base interval $[0.0, 1.0)$ according to their probabilities:




We select the subinterval corresponding to the symbol $b$, that is $[0.4, 0.6)$. Next, we check the renormalization conditions:

1. The interval $[0.4, 0.6)$ lies entirely within the middle half $[0.25, 0.75)$, which means case $E_3$.

   - We scale: $L = 2(0.4 - 0.25) = 0.3$, $H = 2(0.6 - 0.25) = 0.7$.
   - We increment the counter: $counter = 1$.

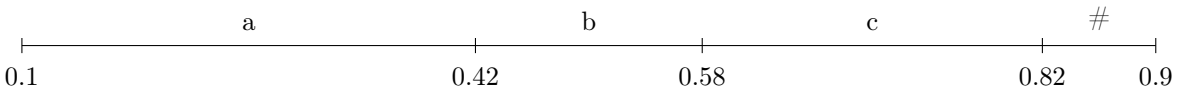
2. The new interval $[0.3, 0.7)$ again satisfies condition $E_3$.

   - We scale: $L = 2(0.3 - 0.25) = 0.1$, $H = 2(0.7 - 0.25) = 0.9$.
   - We increment the counter: $counter = 2$.

The interval $[0.1, 0.9)$ does not require renormalization. We have finished processing the first symbol. The output buffer is still empty.

**Step 2: Encoding the symbol $a$**

We divide the current interval $[0.1, 0.9)$ (with width 0.8) into new subintervals:




We select the subinterval corresponding to the symbol $a$, that is $[0.1, 0.42)$. We check renormalization:

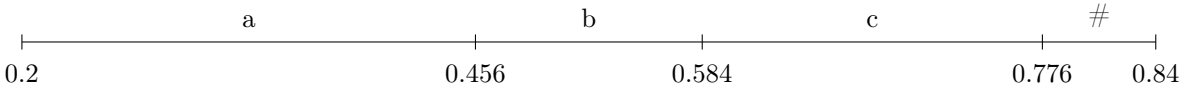
1. The interval $[0.1, 0.42)$ lies in the lower half $[0, 0.5)$, which means case $E_1$.

   - We output bits: the algorithm instructs us to output 0, followed by $counter$ ones. $counter = 2$, so we output 011.
   - We reset the counter: $counter = 0$.
   - We scale: $L = 2(0.1) = 0.2$, $H = 2(0.42) = 0.84$.

The interval $[0.2, 0.84)$ does not require renormalization. The buffer contents are `011`.

**Step 3: Encoding the symbol #**

We divide the interval $[0.2, 0.84)$ (with width 0.64) into new subintervals:




We select the subinterval corresponding to the symbol #, that is $[0.776, 0.84)$. We check renormalization:

1. The interval $[0.776, 0.84)$ lies in the upper half $[0.5, 1.0)$ (case $E_2$).

   - We output the bit: `1` (the counter is 0, so we do not output zeros). Buffer: `0111`.
   - We scale: $L = 2(0.776 - 0.5) = 0.552$, $H = 2(0.84 - 0.5) = 0.68$.

2. The new interval $[0.552, 0.68)$ still lies in $E_2$.

   - We output the bit: `1`. Buffer: `01111`.
   - We scale: $L = 2(0.552 - 0.5) = 0.104$, $H = 2(0.68 - 0.5) = 0.36$.

3. The new interval $[0.104, 0.36)$ lies in the lower half, which means case $E_1$.

- We output the bit: `0`. Buffer: `011110`.
- We scale: $L = 2(0.104) = 0.208$, $H = 2(0.36) = 0.72$.

The interval $[0.208, 0.72)$ does not require renormalization. The buffer contents are `011110`.

**Step 4: The flush sequence**

We analyze the final interval $[0.208, 0.72)$. Since the lower bound $L = 0.208$ is less than 0.25, we apply the first condition of the flush sequence. We therefore output the bit 0, followed by $(counter + 1)$ ones, where $counter = 0$. This yields the sequence `01`, which we append to the buffer.

The final binary code for the message "ba#" is **01111001**.

## 4.2. Decoding example

We are given a source with alphabet $\{a, b, c, \#\}$ and occurrence probabilities $P(a) = 0.4$, $P(b) = 0.2$, $P(c) = 0.3$, and $P(\#) = 0.1$. We are also given the encoded message 01111001. Our goal is to decode it.

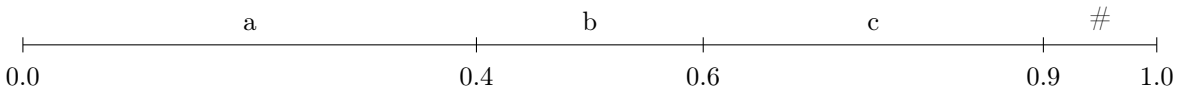
At the beginning, we read the message code and interpret it as a binary fraction $V$:

$$V = 0.01111001_2 = \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{256} = 0.47265625. \tag{15}$$

The initial interval is $L = 0.0$ and $H = 1.0$.

**Step 1: Decoding the first symbol**

We divide the interval $[0.0, 1.0)$ into subintervals:




We compare our value $V \approx 0.4726$ with the boundaries of the subintervals. This value lies in the range $[0.4, 0.6)$. Therefore, the first symbol of the message is $b$. We update the interval to $[0.4, 0.6)$.

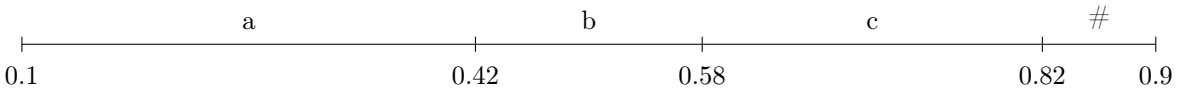
We check the renormalization conditions (remembering that the value $V$ must be transformed synchronously with the boundaries $L$ and $H$):

1. The interval $[0.4, 0.6)$ lies in the middle of the range, which means case $E_3$.
   - We scale the boundaries: $L = 2(0.4 - 0.25) = 0.3$, $H = 2(0.6 - 0.25) = 0.7$.
   - We scale $V$: $V = 2(0.47265625 - 0.25) = 0.4453125$.
2. The new interval $[0.3, 0.7)$ again corresponds to case $E_3$.
   - We scale the boundaries: $L = 2(0.3 - 0.25) = 0.1$, $H = 2(0.7 - 0.25) = 0.9$.
   - We scale $V$: $V = 2(0.4453125 - 0.25) = 0.390625$.

The interval $[0.1, 0.9)$ does not require renormalization. The current code value is $V = 0.390625$. The currently decoded message is "b".

**Step 2: Decoding the second symbol**

We divide the current interval $[0.1, 0.9)$ (with width 0.8) into subintervals:




The value $V = 0.390625$ falls into the range $[0.1, 0.42)$. The second symbol is therefore $a$. We update the interval to $[0.1, 0.42)$.

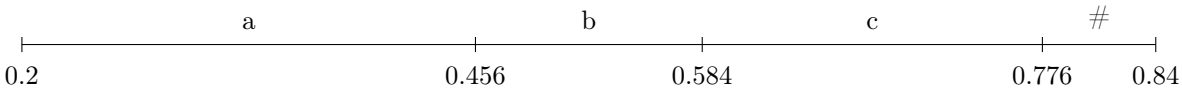
We check the renormalization conditions:

1. The interval $[0.1, 0.42)$ lies entirely in the lower half, which means case $E_1$.
   - We scale the boundaries: $L = 2(0.1) = 0.2$, $H = 2(0.42) = 0.84$.
   - We scale $V$: $V = 2(0.390625) = 0.78125$.

The interval $[0.2, 0.84)$ does not require renormalization. The current code value is $V = 0.78125$. The currently decoded message is "ba".

**Step 3: Decoding the third symbol**

We divide the interval $[0.2, 0.84)$ (with width 0.64) into subintervals:




The value $V = 0.78125$ falls into the range $[0.776, 0.84)$. The third symbol is therefore #. Since this symbol denotes the end of the code, the decoder loop is terminated. The final decoded message is "ba#" (or "ba" after removing #).

**Remark 2.** An observant reader may notice that the message "ba" could have been encoded using a smaller number of bits with a different code. Indeed, it appears that the use of arithmetic coding may have actually increased the number of bits required to represent the message. This problem arises from the fact that the probability distribution used bears little relation to the actual frequencies of symbols in the message "ba#". This shows that even more important than the coding method itself is a correct representation of the source for the encoded message.

## 5. Implementation using integers

One of the most important scientific papers in the history of arithmetic coding is "Arithmetic coding for data compression" [4], published in 1987 by Ian H. Witten, Radford M. Neal, and John G. Cleary. It is the first paper that presented a reliable and efficient implementation of arithmetic coding, containing complete source code of both the encoder and the decoder written in the C language. This implementation quickly became the standard for arithmetic coding. As an interesting historical note, it is worth mentioning that the first scientific works describing arithmetic coding were published in 1976 — as much as 11 years before the paper discussed here.

Instead of storing values as floating-point numbers, the authors used integers. The number of bits determining the precision is fixed in advance for both the interval bounds and the code (originally, this

was 16 bits). The initial value of $L$ is the number 0, and the initial value of $H$ is a number whose binary representation consists entirely of ones at the chosen precision (for 16 bits, this is $2^{16}-1$). The counterpart of the mathematical value 0.5 in this system is exactly half of the range (that is, $2^{15}$ for 16-bit precision). An important detail is the fact that both the encoder and the decoder use the same precision. The use of integers is more efficient, more accurate, and less complicated than the use of floating-point numbers. The benefits of this approach are particularly evident when considering renormalization, where "multiplication by 2" is equivalent to a simple left bit shift. A detailed description of the implementation can be found in the original paper, which interested readers are encouraged to consult.

# 6. Summary

This article discussed the theoretical operation of the arithmetic encoder and decoder. Particular attention was devoted to an intuitive explanation of the purpose and mechanism of renormalization. This mechanism is one of the most important details of arithmetic coding, yet it is characterized by a significantly higher level of complexity than the rest of the algorithm. It is also often poorly explained by the authors of similar publications. A useful tool for gaining a deeper understanding of arithmetic coding is provided by the encoding and decoding examples contained in Section 4., calculated step by step with accessible illustrations. The topic of practical implementation of arithmetic coding was also addressed, which differs from the mathematical theory by using integers to simulate the interval.

# A. Proof of correctness of $E_3$

**Lemma 1.** *Let $E_3^n(x)$ denote the recursive application of the function $E_3$ $n$ times: $E_3^n = \underbrace{E_3 \circ \cdots \circ E_3}_{n \text{ times}}$. Then:*

$$E_3^n(x) = 2^n x - \frac{1}{2}(2^n - 1). \tag{16}$$

*Proof.* We will use induction on $n$.

Base case: let $n_0 = 1$. Then $E_3^1(x) = 2^1 x - \frac{1}{2}(2^1 - 1) = 2x - 0.5 = 2(x - 0.25)$.

Suppose that $E_3^n(x) = 2^n x - \frac{1}{2}(2^n - 1)$. Then:

$$\begin{aligned} E_3^{n+1}(x) &= E_3(E_3^n(x)) \\ &= 2(E_3^n(x)) - 0.5 \qquad \text{(by definition of } E_3\text{)} \\ &= 2(2^n x - \frac{1}{2}(2^n - 1)) - \frac{1}{2} \\ &= 2^{n+1}x - (2^n - 1) - \frac{1}{2} \\ &= 2^{n+1}x - 2^n + \frac{1}{2} \\ &= 2^{n+1}x - \frac{1}{2}(2^{n+1} - 1). \end{aligned} \tag{17}$$

□

The function $E_3^n$ allows us to compute the interval boundaries after encountering case $E_3$ $n$ times in a row. In the remainder of the proof, we will also need the inverse of $E_3^n$:

$$\begin{aligned} 2^n x - \frac{1}{2}(2^n - 1) &= y \\ 2^n x &= y + \frac{1}{2}(2^n - 1) \\ x &= \frac{y + \frac{1}{2}(2^n - 1)}{2^n} \\ (E_3^n)^{-1}(y) &= \frac{y + \frac{1}{2}(2^n - 1)}{2^n}. \end{aligned} \tag{18}$$

Before proceeding further, let us consider the structure of this proof. Intuitively, we already understand why case $E_3$ works. We start with some interval lying within the middle region, for example $[0.3, 0.7)$. We then scale it using the function $E_3$ $n$ times (here, once), until it no longer requires renormalization. In this case, we obtain the interval $[0.1, 0.9)$. After scaling, we select the subinterval of the next encoded symbol. If this subinterval lies in $[0, 0.5)$, then "going back up" will show us that we should also enter the lower half of the interval $[0.3, 0.7)$. For example, suppose that the selected symbol corresponds to the subinterval $[0.2, 0.4)$. To "return" to the interval $[0.3, 0.7)$, we must apply the inverse of the scaling $E_3$ to the subinterval $[0.2, 0.4)$:

$$[L, H) = (E_3^1)^{-1}([0.2, 0.4)) = [\frac{0.2}{2} + 0.25, \frac{0.4}{2} + 0.25) = [0.35, 0.45). \tag{19}$$

We now know that the code of our message, with respect to the interval before the first scaling by $E_3$, lies in $[0.35, 0.45)$. It is crucial to understand what is meant by the phrase "with respect to the interval". We started with the interval $[0.3, 0.7)$. We then performed renormalization $E_3$ and selected the symbol corresponding to $[0.2, 0.4)$. The subinterval $[0.2, 0.4)$ is a subinterval in the context after scaling by $E_3$. However, we went back up using the inverse scaling in order to change the context from "the message code lies in subinterval $[0.2, 0.4)$ of the interval $[0.1, 0.9)$" to "the message code lies in subinterval $[0.35, 0.45)$ of the interval $[0.3, 0.7)$". These statements are equivalent. They differ only in that the former refers to the interval after scaling, while the latter refers to the interval before scaling. Understanding the notion of "reference to an interval" is necessary to understand the following proof.

The equations derived and proven in Lemma 1 and Equation 18 allow us to compute an interval and its inverse after $n$ scalings by the function $E_3$. Note, however, that they are valid only for continuous applications of $E_3$. What happens if, after scaling the interval $[0.3, 0.7)$ to $[0.1, 0.9)$, the subinterval of the next encoded symbol is $[0.4, 0.6)$? This subinterval again falls into case $E_3$, but simply increasing the value of $n$ does not allow us to compute the correct interval after the next scaling. Instead, we must first compute $E_3^1([0.3, 0.7))$, and then $E_3^2([0.4, 0.6))$. The equation $E_3^n$ is correct only for $n$ consecutive scalings of a single interval, without encoding a new symbol in between. Nevertheless, in the proof we must show that the correct code consists of 0 followed by 1 repeated $n$ times, regardless of how many symbols are encoded between the first mapping $E_3$ and the first occurrence of case $E_1$. Fortunately, as will soon become clear, the characteristics of the function $E_3^n$ do not pose a serious problem.

To keep the following proof at a reasonable length, we will prove correctness only for the situation in which the first encountered renormalization case that is not $E_3$ is case $E_1$. Our goal is therefore to show that, regardless of the number of encoded symbols, if the first non-$E_3$ case is $E_1$, then the message code, with respect to the interval before the first $E_3$, must begin with the bit 0, followed by the bit 1 repeated $n$ times.

**Lemma 2.** *Suppose that the first non-$E_3$ renormalization case is $E_1$. Let $n$ be the total number of applications of the mapping $E_3$ before the occurrence of this case $E_1$. Let $[L, H)$ be the correct subinterval of the code with respect to the interval before the first scaling by $E_3$. Then:*

$$\frac{1}{2} - \frac{1}{2^{n+1}} \leq L < H < \frac{1}{2}. \tag{20}$$

*Proof.* From the definition of case $E_1$, we know that at the moment it is encountered, the interval lies entirely in the lower half, i.e. it is a subset of the interval $[0, \frac{1}{2})$.

Observe that the process of encoding symbols between mappings $E_3$ consists solely of selecting subintervals, which only narrows the solution space. This means that each encoding step preserves the containment relation. Therefore, to find the final boundaries with respect to the initial interval, it suffices to map the entire target set $[0, \frac{1}{2})$ back through $n$ mappings of $E_3$.

Since the function $E_3(x)$ and its inverse $(E_3^n)^{-1}(x)$ are affine functions with positive slopes (thus they are continuous and strictly increasing), the preimage of the interval $[0, \frac{1}{2})$ is also a continuous interval, and its boundaries can be determined by substituting the extreme values into the inverse formula (Equation 18):

$$(E_3^n)^{-1}(0) = \frac{0 + \frac{1}{2}(2^n - 1)}{2^n} = \frac{2^{n-1} - \frac{1}{2}}{2^n} = \frac{1}{2} - \frac{1}{2^{n+1}},$$

$$\lim_{y \to \frac{1}{2}} (E_3^n)^{-1}(y) = \frac{\frac{1}{2} + \frac{1}{2}(2^n - 1)}{2^n} = \frac{\frac{1}{2} + 2^{n-1} - \frac{1}{2}}{2^n} = \frac{2^{n-1}}{2^n} = \frac{1}{2}.$$

We therefore obtain the preimage equal to $[\frac{1}{2} - \frac{1}{2^{n+1}}, \frac{1}{2})$. Since the final subinterval before case $E_1$ lies within $[0, \frac{1}{2})$, the corresponding initial subinterval $[L, H)$ must lie within its preimage:

$$[L, H) \subseteq \left[\frac{1}{2} - \frac{1}{2^{n+1}}, \frac{1}{2}\right), \tag{21}$$

which proves that $\frac{1}{2} - \frac{1}{2^{n+1}} \leq L < H < \frac{1}{2}$. □

**Theorem 1.** *If the first non-$E_3$ renormalization case is $E_1$, then the binary message code, with respect to the interval before the first scaling by $E_3$, begins with the bits $01^n$, where $1^n$ denotes writing the bit 1 as many times as the scaling $E_3$ was applied.*

*Proof.* Let $[L, H)$ be the correct subinterval of the code with respect to the interval before the first scaling by $E_3$. Then, according to Lemma 2, we have $\frac{1}{2} - \frac{1}{2^{n+1}} \leq L < H < \frac{1}{2}$. The inequality $\frac{1}{2} - \frac{1}{2^{n+1}} \leq L$ is particularly important for us. Note that the lower bound $\frac{1}{2} - \frac{1}{2^{n+1}}$ can be expanded into a sum of powers

of two (assuming a finite binary representation of the number $\frac{1}{2}$):

$$\frac{1}{2} - \frac{1}{2^{n+1}} = \sum_{i=2}^{n+1} 2^{-i}. \tag{22}$$

In the binary system, this value corresponds exactly to the fraction $0.0\underbrace{1\ldots1}_{n \text{ times}}{}_2$, where the bits in fractional positions from 2 to $n+1$ inclusive are equal to 1. Since the upper bound $H < \frac{1}{2}$, all numbers in the interval $[L, H)$ must have a 0 in the first fractional bit position. At the same time, because they are greater than or equal to the lower bound $L$, their next $n$ bits must be equal to 1. This proves that the binary code of every message from the interval $[L, H)$, with respect to the interval before the first scaling by $E_3$, begins with the bits $01^n$. □

The proof of correctness of renormalization $E_3$ ending with case $E_2$ proceeds analogously and is left as an exercise for interested readers.

## References

1. D.A. Huffman, *A Method for the Construction of Minimum-Redundancy Codes*, Proceedings of the IRE 40 (1952), pp. 1098-1101.
2. K. Sayood, *Introduction to Data Compression 3rd edition*, Morgan Kaufmann Publishers, San Francisco 2006.
3. C.E. Shannon, *A Mathematical Theory of Communication*, Bell Syst. Tech. J. 27 (1948), pp. 379-423, 623-656.
4. I.H. Witten, R.M. Neal, J.G. Cleary, *Arithmetic coding for data compression*, Communications of the ACM 30 (1987), pp. 520-540.