**Tutorial: Arithmetic Coding**Yu-Yun Chang (張又允)

E-mail: r01942029@ntu.edu.tw

Graduate Institute of Communication Engineering

National Taiwan University, Taipei, Taiwan, ROC

**ABSTACT**

Data compression, or called source coding, involves encoding information using fewer bits than the original representation. Data compression is useful since it helps reduce resources usage, which often has some constraints, such as data storage space or transmission capacity. Therefore, many compression techniques are proposed, and one of them is arithmetic coding.

Arithmetic coding is a data compression technique that encodes data by creating a code which represents a fraction in the unit interval [0, 1]. The algorithm is recursive. On each recursion, the algorithm successively partitions subintervals of the unit interval [0, 1]. This differs from other forms of entropy encoding such as Huffman coding, which separates the input into component symbols and replacing each with a code.

In this tutorial, we will discuss some important concepts of arithmetic coding and learn how to encode by means of some examples.

**INTRODUCTION**

Arithmetic coding, which is a method of generating variable-length codes, is useful when dealing with sources with small alphabets such as binary sources. In order to explain, let us compare arithmetic coding with Huffman coding.

Huffman coding, which is the most famous method for source coding or data compression, guarantees a coding rate within 1 bit of the entropy That is, the Huffman code for a source with an average codeword length satisfies

Moreover, if we encode the output of the source in longer blocks of symbols, we are guaranteed the average codeword length per input symbol closer to the entropy. In other words, suppose we encode the sequence by generating a codeword for every symbols, and then we have

where denotes the average codeword length per input symbol.

However, there is still another problem. The latter approach becomes impractical since it causes an exponential growth in the size of the codebook when we try to obtain Huffman codes for long sequences of symbols. In other words, the complexity of this approach increases exponentially with block length.

Arithmetic coding is a method of encoding without this inefficiency. In arithmetic coding, instead of using a sequence of bits to represent a symbol, we represent it by a subinterval of the unit interval . In other words, we encode the data into a number in the unit interval , and this technique can be implemented by separating the unit interval into several segments according to the number of distinct symbols. The length of each segment is proportional to the probability of each symbol, and then the output data is located in the corresponding segment according to the input symbol.

This provides a way of assigning codewords to particular sequences without having to generate codewords for all sequences and alleviates the inefficiency and the complexity. Moreover, the code for a sequence of symbols is an interval whose length decreases as we add more symbols to the sequence. This property allows us to have a coding scheme that is incremental, that is, the code for an extension to a sequence can be calculated simply from the code for the original sequence.

In the following chapters, we discuss basic concepts about arithmetic coding in Chapter 1, the motivation and the coding theorem behind arithmetic coding in Chapter 2, the method of generating a binary sequence in Chapter 3, the adaptive arithmetic code in Chapter 4, and end up with the conclusion.

**CHAPTER 1: BASIC CONCEPTS**

In order to distinguish a sequence of symbols from another sequence of symbols, we need to tag it with a unique identifier. A possible set of tags for representing sequences of symbols is the numbers in the unit interval [0, 1]. Because the number of numbers in the unit interval is infinite, it is able to assign a unique tag to each distinct sequence of symbols. In order to do this we need a function that will map sequences of symbols into the unit interval. Apparently, a possible function that maps random variables, and sequences of random variables, into the unit interval is the cumulative distribution function (cdf) of the random variable associated with the source. This is the function we will use in developing the arithmetic code.

Before we begin the development of the arithmetic code, we introduce an important lemma that motivates the concept of arithmetic coding.

**Lemma 1.1**  
*Let*  *be a random variable with continuous probability distribution function . Let (i.e.,is a function of*  *defined by its distribution function). Then*  *is uniformly distributed on .*

**Proof**Since , the range of is *.* Also, for ,  
which proves that has a uniform distribution in .

□

Now we establish some notations. From the definition, a random variable maps the outcomes, or sets of outcomes, of an experiment to values on the real number line. For example, in a coin-tossing experiment, a random variable can map a head to one and a tail to zero. Similarly, we can map the source symbols to numbers. In the following discussion, we will use the mapping

where is the alphabet for a discrete source and is a random variable. This mapping means that given a probability model for the source, we can obtain a probability density function (pdf) for the random variable  
and the cumulative density function (cdf) can be expressed as  
Hence for each symbol with a nonzero probability, we have a distinct value of . Since the value of is distinct for distinct symbol , we are able to use this fact in what follows to develop the arithmetic code.  
 If you have heard the Shannon-Fano code, which is a technique for constructing a prefix code based on a set of symbols and their probabilities, you may discover that the above development sounds familiar. In fact, the motivation of the arithmetic code is based on the Shannon-Fano code and the Lemma 1.1 described above [1].

**CHAPTER 2: CODING A SEQUENCE**

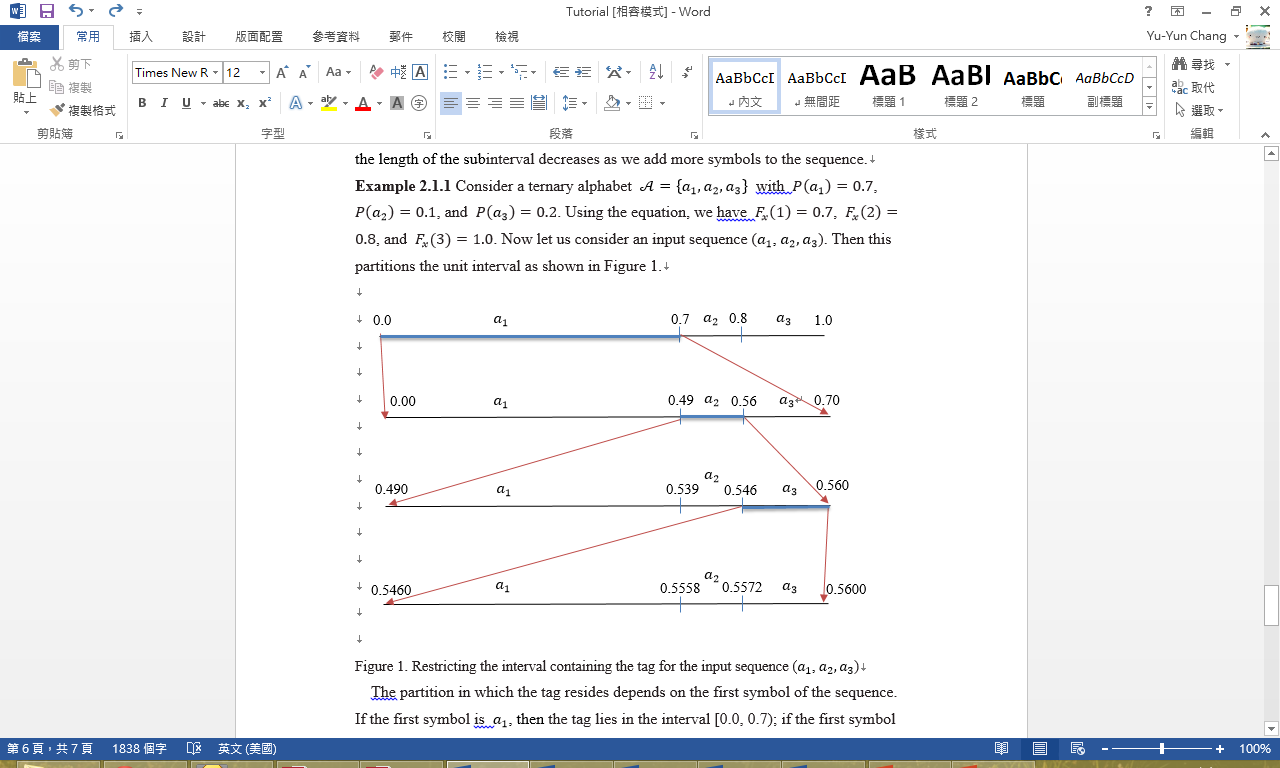
**2.1 Motivation behind the Arithmetic Coding**

In order to gain the intuition to Arithmetic coding, we first discuss the motivation behind the arithmetic code. The purpose of data compression or source coding is to find a binary code that will represent the sequence in a unique and efficient manner, where and can be viewed as the ith sample of the random variable . Note that and .  
 As described in the introduction, we can generate a tag that forms a unique representation for the sequence. This means that the binary representation of the tag forms a unique binary code for the sequence. However, since we have placed no restrictions on what values the tag can take in the unit interval, the binary representation of some of these values would be infinitely long. This will make the code inefficient although the code is unique. Therefore, in order to make the code efficient, the binary representation has to be truncated.  
 In order to explain more clearly, let us consider an infinite sequence of random variables drawn from the set corresponding to the alphabet . For any infinite sequence from this alphabet, we can place in front of the sequence and consider it as a real number (base ) between 0 and 1. Let be the real-valued random variable . Then has the following cumulative density function:

Now let .If the distribution on infinite sequences has no atoms, then, by Lemma 1.1, has a uniform distribution on , and therefore the bits in the binary expansion of are Bernoulli. In other words, they are independent and uniformly distributed on . Hence these bits are incompressible and form a compressed representation of the sequence , which can be used as a code representing the sequence. We demonstrate how to compute the cumulative density function in the following example.  
**Example 2.1**  
Let be Bernoulli(). Then the sequence **x** = 110101 maps into  
whereNote that each term is easily computed from the previous terms.

□

**2.2 Generating a Unique Tag**

The procedure for generating the tag works by reducing the size of the interval in which the tag resides as more and more symbols of the sequence are added. First, we divide the unit interval into subintervals of the form We associate this subinterval with the symbol . The appearance of the first symbol in the sequence restricts the interval containing the tag to one of these subintervals. Suppose that the first symbol was *,* andthen the interval containing the tag value will be the subinterval*.* This subinterval is now partitioned in exactly the same proportions as the original interval. That is, the jth interval corresponding to the symbol is given by. So if the second symbol in the sequence is *,* then the interval containing the tag value becomes . Each succeeding symbol causes the tag to be restricted to a subinterval that is further partitioned in the same proportions. Hence the length of the subinterval decreases as we add more symbols to the sequence.  
**Example 2.1.1**Consider a ternary alphabet with , , and . Using the equation, we have , , and . Now let us consider an input sequence (,). Then this partitions the unit interval as shown in Figure 1.  
  
Figure 1. Restricting the interval containing the tag for the input sequence (,)  
 The partition in which the tag resides depends on the first symbol of the sequence. If the first symbol is , then the tag lies in the interval [0.0, 0.7); if the first symbol is , then the tag lies in the interval [0.7,0.8); if the first symbol is , then the tag lies in the interval [0.8, 1.0). Once the interval containing the tag has been determined, the rest of the unit interval is discarded, and this restricted interval is again divided in the same proportions as the original interval.

Now the input sequence is (,). The first symbol is , and the tag would be contained in the subinterval [0.0, 0.7). This subinterval is then subdivided in exactly the same proportions as the original interval, yielding the subintervals [0.00, 0.49), [0.49, 0.56), and [0.56, 0.70). The first partition [0.00, 0.49) corresponds to the symbol , the second partition [0.49, 0.56) corresponds to the symbol , and the third partition [0.56,0.70) corresponds to the symbol . The second symbol in the sequence is . The tag value is then restricted to lie in the interval [0.49, 0.56). We now partition this interval in the same proportion as the original interval in order to obtain the subinterval [0.49, 0.539) corresponding to the symbol , the subinterval [0.539,0.546) corresponding to the symbol and the subinterval [0.546,0.560) corresponding to the symbol . If the third symbol is , then the tag will be restricted to the interval [0.546,0.560), which can be subdivided further by following the procedure described above.  
 Note that the appearance of each new symbol restricts the tag to a subinterval that is disjoint from any other subinterval that may have been generated using this process. For the sequence (,), since the third symbol is , the tag is restricted to the subinterval [0.546, 0.560). If the third symbol is instead of , the tag would have resided in the subinterval [0.49, 0.539), which is disjoint from the subinterval [0.546,0.560).

□

As described in the above example, the interval in which the tag for a particular sequence resides is disjoint from all intervals in which the tag for any other sequence may reside. Hence, any number in this interval can be used as a tag since the intervals for distinct sequences are disjoint, and one possible choice is the midpoint of the interval. In the following discussion, we will use the midpoint of the interval as the tag.

Suppose we have a source that generates symbols from the alphabet . Using Equations (7), (8), and (9), we define  
For each , has a unique value, and thus this value can be used as a unique tag for . In the following example, we show how to find an appropriate tag.

**Example 2.1.2**Consider an experiment that we throw a fair die. The outcomes of a roll of the die can be mapped into the numbers . For a fair die,  
Hence, using (13), we obtain that the tag for is  
and the tag for is

□

This approach can be extended to longer sequences by imposing an order on the sequences. We need an ordering on the sequences because we will assign a tag to a particular sequence as  
where means that the order of is in front of , and the superscript denotes the length of the sequence.

An ordering that is common to use is lexicographic ordering. In lexicographic ordering, the ordering of letters in an alphabet induces an ordering on the words constructed from this alphabet. In fact, the use of lexicographic ordering is very common in our daily life. For example, the ordering of words in a dictionary is lexicographic ordering. We give an example to demonstrate how to use (14) to generate a tag.

**Example 2.1.3**

We throw a die twice and keep a record of the outcome of each roll. Using the ordering strategy described above, i.e., the lexicographic ordering, the outcomes in order would be 11, 12, 13, …, 66. The tags can be generated using (14).

For example, the tag for the sequence 15 is

□

Note that in order to generate the tag for 15, we do not need to generate other tags for other messages. Based on (14), the only thing we need to know is the probability of every sequence that precedes the sequence for which the tag is being generated. Moreover, if we want to compute a tag for a given sequence of symbols, we just need the probability of individual symbols, or the probability model. In other words, if we know the distribution of the sequences, we can apply (14) directly without inefficient computation. As described in the induction, this is one of the advantages of the arithmetic code.

**2.3 Another Approach for Generating a Tag**

As described in Section 2.2, the interval containing the tag value for a given sequence is disjoint from the intervals containing the tag values of all other sequences. This means that any value in this interval would be a unique identifier for , and a usual choice is the midpoint of the interval. Therefore, in order to satisfy that each sequence can be uniquely decodable, it is sufficient to compute the top end and the bottom end of the interval containing the tag and select any value in that interval. Moreover, the top end and the bottom end can be computed recursively.

**Example 2.1.4**Consider an experiment that we throw a fair die. The outcomes of a roll of the die can be mapped into the numbers {l,2,… ,6}. Now we want to find the top end and the bottom end of the interval containing the tag for the sequence 322.  
Assume that we observe 3, 2, 2 in a sequential manner, i.e., we see 3 in the beginning, then 2, and then 2. After each observation, we compute the top end and the bottom end of the interval containing the tag of the sequence observed to that point. In the following, we denote the top end by and the bottom end by , where denotes the length of the sequence.  
(1) First we observe 3. Then we have  
(2) Then we observe 2, and the observed sequence becomes Hence  
Using the technique developed in Example 2.1, we obtain  
We can rewrite the above equation as  
Similarly, we can obtain the relation  
by following the same procedure.  
(3) The third element of the observed sequence is 2, and thus the observed sequence is . The top end and the bottom end of the interval containing the tag for this sequence are  
Using the same approach as above, we find that  
or equivalently,   
(4) In general, we can show that for any sequence Hence again, in this approach, we do not need to compute any joint probabilities.  
If we use the midpoint of the interval for the tag, then

□

The above equation indicates that we have a coding scheme that is incremental; that is, the code for an extension to a sequence can be calculated simply from the code for the original sequence, as described in the introduction. The only information required by the tag generation procedure is the cumulative density function or the distribution of the source, which can be obtained directly from the probability model.

**CHAPTER 3: GENERATING A BINARY SEQUENCE**

**3.1 Efficiency of the Arithmetic Code**

According to (22), is a number in the interval [0, 1]. Therefore, a binary code for can be obtained by taking the binary representation of this number and truncating it to bits, where denotes the logarithm to base 2. The following example show how to generate a binary code.

**Example 3.1.1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Symbol |  |  |  | in binary |  | Codeword |
| 1 | 0.25 | 0.125 | 0.125 | 0.001 | 3 | 001 |
| 2 | 0.5 | 0.75 | 0.5 | 0.10 | 2 | 10 |
| 3 | 0.125 | 0.875 | 0.8125 | 0.1101 | 4 | 1101 |
| 4 | 0.125 | 1.0 | 0.9375 | 0.1111 | 4 | 1111 |

The quantity is obtained using Equation (13), and the binary representation of is truncated to bits to obtain the binary code.

□

**Example 3.1.2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Symbol |  |  |  | in binary |  | Codeword |
| 1 | 0.25 | 0.25 | 0.125 | 0.001 | 3 | 001 |
| 2 | 0.25 | 0.50 | 0.375 | 0.011 | 3 | 011 |
| 3 | 0.2 | 0.70 | 0.60 |  | 4 | 1001 |
| 4 | 0.15 | 0.85 | 0.775 |  | 4 | 1100 |
| 5 | 0.15 | 1.0 | 0.925 |  | 4 | 1110 |

□

It can be proved that a code obtained in this fashion is a uniquely decodable code, but we do not go through the detail here. If you are interested, you can consult the references for this tutorial.

Now we show the efficiency of the arithmetic code. Since we use bits to represent a sequence, the average length of an arithmetic code for a sequence of length m is given by  
Since the average length is always greater than the entropy, the bounds on are

Define the average length per symbol , sometimes called the rate of the arithmetic code, is  
If the source is i.i.d., then  
and thus  
Therefore, as a sequence of length increases, we can guarantee a rate as close to the entropy as we desire.

**3.2 Algorithm Implementation**

In Section 2.3, we developed a recursive algorithm for the boundaries of the interval containing the tag for the sequence being encoded as

Before we implement this algorithm, there is a problem we must resolve. We use numbers in the interval [0, 1) as a tag, and there are an infinite number of numbers in this interval. However, in practice the number of numbers that can be uniquely represented on a machine is limited by the maximum number of bits that we can use for representing a number. As gets larger, these values come closer. This means that in order to represent all the subintervals uniquely we need to increase precision as the length of the sequence increases. In a system with finite precision, the two values are bound to converge, and we will lose all information about the sequence.  
 Therefore, in order to avoid this situation, we need to rescale the interval. However, we still want to preserve the information that is being transmitted. We would also like to perform the encoding incrementally, that is, to transmit portions of the code as the sequence is being observed, rather than wait until the entire sequence has been observed before transmitting the first bit.  
 As the interval becomes narrower, we have three possibilities:

1. The interval is entirely confined to the lower half of the unit interval [0,0.5).

2. The interval is entirely confined to the upper half of the unit interval [0.5, 1.0).

3. The interval straddles the midpoint of the unit interval.

For the first two cases, once the interval is confined to either the upper or lower half of the unit interval, it is forever confined to that half of the unit interval. Hence, once the interval gets confined to either the upper or lower half of the unit interval, the most significant bit of the tag is fully determined. Hence, without waiting to see what the rest of the sequence is, we can indicate to the decoder whether the tag is confined to the upper or lower half of the unit interval by sending a 1 for the upper half and a 0 for the lower half. The bit that we send is also the first bit of the tag. Once the encoder and decoder know which half of the unit interval contains the tag, we can ignore the half of the unit interval that does not contain the tag and focus on the half of the unit interval that contains the tag.

If only finite precision or bits that we can use, we can do this by mapping the half interval that contains the tag to the full [0, 1) interval. The mappings required are

Once we perform either of these mappings, we lose all information about the most significant bit. However, this is not important since we have already sent that bit to the decoder. We can now continue with this process, and then generate another bit of the tag. This process of generating the bits of the tag without waiting to see the entire sequence is called incremental encoding, which we have discussed a little in the introduction.  
**Example 3.2.1** *(Tag generation with scaling)*Now we wish to encode the sequence 1 3 2 1. The probability model for the source is , , and . In the beginning,

(a) For the first symbol 1, we have  
The interval [0,0.8) is not restricted to either the upper or lower half of the unit interval, so we can proceed further.  
(b) For the second symbol 3, we have  
The interval [0.656,0.8) is contained entirely in the upper half of the unit interval, so we send the binary code 1 and rescale as

(c) For the third symbol 2, we have  
The interval for the tag is [0.5424, 0.54816), which is contained entirely in the upper half of the unit interval. Thus we transmit a 1 and go through another rescaling  
This interval is contained entirely in the lower half of the unit interval, so we send a 0 and use the mapping to rescale

Since the interval containing the tag remains in the lower half of the unit interval, we send a 0 and rescale  
Since the interval containing the tag remains in the lower half of the unit interval, we send a 0 and rescale again

Now the interval containing the tag is contained entirely in the upper half of the unit interval. We transmit a 1 and rescale using the mapping

In fact, at each stage, we transmits the most significant bit that is the same in both the top end and the bottom end of the interval. If the most significant bits in the top end and the bottom end are the same, then the value of this bit will be identical to the most significant bit of the tag. Therefore, by sending the most significant bits of the upper and lower endpoint of the tag whenever they are identical, we are actually sending the binary representation of the tag.

(d) For the last symbol 1, we have

(e) Now if we want to stop encoding, the only thing we need to do is inform the receiver of final status of the tag value. We can do so by simply sending the binary representation of any value in the final tag interval. In this example, it is convenient to use the value of 0.5. Since the binary representation of 0.5 is 0.10…, we would transmit a 1 followed by as many 0s as required by the word length of the implementation used.

□

The binary sequence generated during the encoding process in the previous example is 1100011. Moreover, we could simply treat this as the binary expansion of the tag. In other words, a binary number 0.1100011 corresponds to the decimal number 0.7734375. Note that this number lies within the final tag interval, and thus we could use this to decode the sequence.

Now we discuss how to decode. Once we start to decode, all we have to do is to mimic the encoder. Hence, in order to guarantee unambiguous decoding, the number of bits received should point to an interval smaller than the smallest tag interval. Based on the smallest tag interval, we can determine how many bits we need before we start the decoding procedure. We demonstrate how to decode by an example.

**Example 3.2.2**

As in the encoder, we start with . The sequence of received bits is

110001100…0. The first 6 bits correspond to a tag value of 0.765625, which means that the first symbol of the sequence is 1. Hence, we have

(a)

Note that the interval [0, 0.8) is not confined to either the upper or lower half of the unit interval, so we proceed.

(b)

The tag 0.765625 lies in the top 18% of the interval [0, 0.8). Therefore, the second symbol of the sequence is 3. Hence we have

(c)

The interval [0.656,0.8) is contained entirely in the upper half of the unit interval. At the encoder, we sent the bit 1 and rescaled. At the decoder, we will shift l out of the receive buffer. We also update the tag interval, which results in

while shifting a bit to give us a tag of 0.546875. When we compare this value to the tag interval, we can see that this value lies in the 80-82% range of the tag interval, so we decode the third symbol of the sequence as 2. We can then update the equations for the tag interval as

As the tag interval is now contained entirely in the upper half of the unit interval, we rescale using to obtain  
We also shift out a bit from the tag and shift in the next bit. Now the tag is 000110. The interval is contained entirely in the lower half of the unit interval. Therefore, we apply and shift another bit. The lower and upper limits of the tag interval become  
and the tag becomes 001100. The interval is still contained entirely in the lower half of the unit interval, so we shift out another 0 to get a tag of 011000 and go through another rescaling

Because the interval containing the tag remains in the lower half of the unit interval, we shift out another 0 from the tag to get 110000 and rescale again

Now the interval containing the tag is contained entirely in the upper half of the unit interval. Therefore, we shift out a 1 from the tag and rescale using the mapping

Now we compare the tag value to the tag interval to decode our final element. The tag is 100000, which corresponds to 0.5. This value lies in the first 80% of the interval, so we decode this element as 1.

□

**CHAPTER 4: ADAPTIVE ARTITHMETIC CODING**

In this chapter, we discuss adaptive arithmetic coding but do not go through the detail. So far we have realized how to construct arithmetic codes when the distribution of the source is available. However, we are often ignorant about the distribution of the source. Hence, we always want to find a coding scheme such that even though we do not know the true distribution, we still can approach the entropy as the length of the sequence goes to infinity. A possible solution is to use a technique called the universal source coding. Fortunately, it is a relatively simple task to modify the algorithms discussed above so that the coder learns the distribution as the procedure of coding proceeds.

A possible implementation is to start with a count of 1 for each letter in the alphabet. We need a count of at least 1 for each symbol. Note that we assume that we know nothing about the distribution of the source. If we do know something about the distribution of the source, we can let the initial counts based on our knowledge. After we start to encode, the count for each letter encountered is increased after that letter has been encoded. The cumulative count table is updated accordingly. It is very important that the update procedure must be taken after encoding, or otherwise the decoder will not use the same cumulative count table. At the decoder, the count and cumulative count tables are updated after each letter is decoded.

**CONCLUSION**

In this tutorial, we introduced some concepts and the motivation behind arithmetic coding. We showed that the arithmetic code provide an average description length per symbol is close to the entropy as the length of the sequence approach infinity. We compared the arithmetic code to the Huffman code, and discuss how to encode a source with the arithmetic code.

If you are interested in the detail about arithmetic coding, we recommend you to consult the books or papers listed in the references, which may explain more clearly about the detail.

**REFERENCES**

[1] Thomas M. Cover, Joy A. Thomas, “Elements of Information Theory”, 2nd ed., *Wiley-Interscience*, 2006.

[2] Khalid Sayood,”Introduction to Data Compression”, 3rd ed., *Morgan Kaufmann,* 2005.

[3] Rissanen, J.J, Langdon, G.G., Jr., "Arithmetic coding", *IBM Journal of Research and Development*, 23 (2): 149–162, March, 1978.

[4] Langdon, G.G., Jr, “An Introduction to Arithmetic Coding”, *IBM Journal of Research and Development,* Vol. 28, Issue 2, March 1984.