Communication System Lab Final Report

B09901069 張創渝
B09901099 江承恩
B09901156 張哲銘

1 Motivation

In our survey presentation, we have explored the fundamental limits of data compression as well as some practical compression schemes. We would like to further our understanding by implementing various data compression methods and identifying the pros and cons.

2 Preliminaries

We'll skip Huffman Coding here as it has already been well-discussed in lecture.

2.1 Arithmetic Coding (AC)

Arithmetic coding is a popular technique for lossless data compression, which was introduced by the mathematician Richard W. Hamming in 1950 and further developed by various researchers including IBM's Jorma Rissanen in the 1970s. Unlike Huffman coding, which assigns variable-length codes to individual symbols, arithmetic coding encodes entire sequences of symbols into a single floating-point number in the interval [0, 1].

The basic idea behind arithmetic coding is to represent a sequence as a sub-interval of the unit interval [0, 1], with the size of the sub-interval proportional to the probability of the sequence occurring. So we need probability density function for arithmetic coding. During encoding, the first input symbol is mapped to a sub-interval within the unit interval, and this sub-interval is divided further and further as each symbol is encoded, with the sub-interval for each subsequent symbol being nested within the sub-interval of the previous symbols. This nesting allows for the entire sequence to be represented by a single real number.

Arithmetic coding offers higher compression efficiency compared to Huffman coding, especially for sequences with non-uniform probability distributions. However, it is more computationally complex and requires floating-point arithmetic, which may be less efficient on some hardware platforms. In spite of the limitation, arithmetic coding is still widely used in various applications where high compression efficiency is paramount, such as in image and video compression algorithms like JPEG2000.

2.2 Asymmetric Numeral Systems(ANS)

Entropy coding is a lossless data compression method that aims to approach the lower bound announced by Shannon's source coding theorem: if the symbols are generated according to a discrete memoryless source (DMS), then the expected codeword length must be greater of equal to the entropy of the source. ANS is a relatively new entropy encoding method used in data compression [1], which is quite surprising long after Arithmetic Coding and Huffman Coding have already given a near-optimal compression result.

ANS aims to combine the efficiency of Huffman coding and the quality of Arithmetic coding. The main idea is pretty simple: Use a single number to encode the entire symbol sequence. Denote the symbol space as $\mathcal{A} = \{a_1, ..., a_n\}$. Suppose that we have already compressed the symbol sequence $s_1...s_i \in \mathcal{A}^i$ into a number X_i, then an encoding function takes in the pre-compressed number and the next symbol to generate a new compressed number

$$X_{i+1} := \mathcal{E}(X_i, s_{i+1}) \tag{1}$$

We would also expect that this operation is *reversible*, i.e., we can convert the result back to get the compressed symbol as well as the previous message:

$$(X_i, s_{i+1}) = \mathcal{D}(X_{i+1}) \tag{2}$$

Here, we can see that the key difference between ANS and AC lies in the order where the symbol is decoded: in ANS, the decoded symbol is 'stack-like' (last-in-first-out, LIFO), whereas in AC the decoded symbol is 'queue-like' (first-in-first-out, FIFO) [2]. With such advantages, ANS is used by various famous corporations, such as Facebook Zstandard compressor, Apple LZFSE compressor, and Google Draco 3D compressor.

2.3 Range-variant (rANS)

rANS can be viewed as a modification of base-n representation for a number. If a symbol sequence $S = s_1...s_n$ is sent, then the encoding function is given by

$$X_{i+1} = \left\lfloor \frac{X_i}{f_S(s_{i+1})} \right\rfloor \times N + F_S(s_{i+1}) + \mathrm{mod}(X_i, f_S(s_{i+1})) \tag{3}$$

where $f_S(a_i)$ is the frequency count of a_i in the symbol sequence S, $F_S(a_i) = \sum_{k=0}^{i-1} f_S(a_k)$ is the cumulative frequency counts, and $N = |S|$ is the number of symbols sent. Let $F_S^{-1}(y)$ be the generalized inverse for the cumulative frequency count function; that is, $F_S^{-1}(y) = a_i$ iff $F_S(a_i) \leq y < F_S(a_{i+1})$, then the decoding function can be given as

$$s_{i+1} = F_S^{-1}(\mathrm{mod}(X_{i+1}, N)) \tag{4}$$

$$X_i = \left\lfloor \frac{X_{i+1}}{N} \right\rfloor \times f_S(s_{i+1}) + \mathrm{mod}(X_{i+1}, N) - F_S(s_{i+1}) \tag{5}$$

Notice that the expected average codeword length is just the entropy of the source, which somehow shows the optimality of rANS:

$$\frac{X_i}{X_{i-1}} \sim \frac{N}{f_S(s_i)} \sim \frac{1}{p(s_i)} \tag{6}$$

$$\log_2(X_n) \sim \sum_{i=1}^{N} \frac{1}{p(s_i)} \sim nH(S) \tag{7}$$

2.3.1 Renormalization

Though this seems nice, in practice X_i would actually grow exponentially. For a reasonable input file length N in a sound file, it is impossible to represent the number 2^N within the computer arithmetic. To resolve the issue where X_i becomes inevitably large, a simple method is to restrict X_i into a specific range. This can be done by the *renormalization* process. By adapting the X_i into a specific range, rANS could continue to encode more symbols, known as the *streaming*-rANS. [1] has given the particular range for a given input sequence S.

2.4 Table-variant (tANS)

rANS is faster compared to arithmetic coding, but it still involves some computations, including multiplications. However, since the change in number X depends on the input symbol sequence in ANS, it is possible to precompute all possible input situations and store them in a table. This is the basic concept of the table variant, also known as tANS. In fact, tANS is likely the most practical variant, as it compacts the entire ANS behavior into lookup tables, thereby simplifying the coding procedure.

To set up the table, first we define 2^N states S_i for our finite state machine, where N is a user-defined parameter. Each state is then given a number X_i. Then we will spread the symbol to these states according to its probability distribution. The optimal symbol spread turns out to be a challenging task, here we refer to the way mentioned in [3]. Then, we consider the whole encoding (and decoding) process as transfer between states: imagine we are in state S_i and we met symbol s_i, we will transfer to a new state S_j which contains number X_j and output the bitstream representing this transfer. Surprisingly, such encoding workflow works just as formula (1), which is the idea of ANS, as long as we have a good definition on how encoding and decoding transfer the state [3].

Let's clarify with an example. Consider figure 1 with two possible symbols, a and b having probabilities $\frac{3}{4}$ and $\frac{1}{4}$ respectively. We create four states (imply that n = 2) with number 4, 5, 6, and 7 and spread three a and one b onto it. Then, let's assume that we're in the state where $X = 4$. From the FSM path, we may see that upon encountering a, we'll tranfer to state where $X = 6$ with no output; otherwise upon encountering b, we'll tranfer to state where $X = 5$ with two output bit. This makes great sense that b, being less probable, carries more information than a, which is more probable and carries less information. (ρ means the probability to get to that state when i.i.d input source, notice the sum for a is $\frac{3}{4}$.)

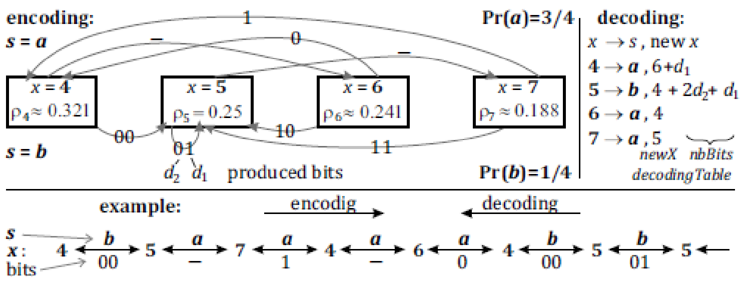


Figure 1: Example of tANS Finite State Machine

Decoding turns out to be the process to retrieve symbol from encoded bitstream. For decoding, we also precompute a decoding table t where $t[X]$ contains $t[X].newX$ and $t[X].nbBits$ that helps us with state transfer. The decoding process can be described by

$$X = t[X].newX + readBits(t[X].nbBits) \tag{8}$$

For example, if we're in state $X = 4$, this means that we arrive at this state by input a when doing encoding, so we output symbol a. Then, according to decoding table given top right corner, $t[4].newX = 6$ and $t[X].nbBits = 1$. This means the state to backtrace would be 6 plus 1 bit of encoded bitstream, making backtraced state either $X = 6$ or $X = 7$.

2.5 Lempel-Ziv-Welch (LZW)

LZW (Lempel-Ziv-Welch) coding is a popular compression algorithm used for lossless data compression. It was first developed by Abraham Lempel and Jacob Ziv in 1977, and later refined by Terry Welch in 1984. LZW works by replacing repeated occurrences of data with references to a dictionary that contains previously encountered patterns. This dictionary is built dynamically as the data is encoded.

Before encoding and decoding, we need an initial dictionary with all possible symbols. For encoding, encoder will export the code words according to the dictionary, and update the new sequence of symbols into the dictionary. For decoding, the decoder use the code words to redo the encoding process, export original sequence, and reconstruct the dictionary. Note that the decoding process doesn't need the appended dictionary from encoding process.

LZW offers efficient compression for data with repetitive patterns, such as text or certain types of images. It's widely used in various file formats like GIF (Graphics Interchange Format) and TIFF (Tagged Image File Format). However, it's worth noting that LZW may not be as effective for highly randomized or already compressed data.

3 Evaluation

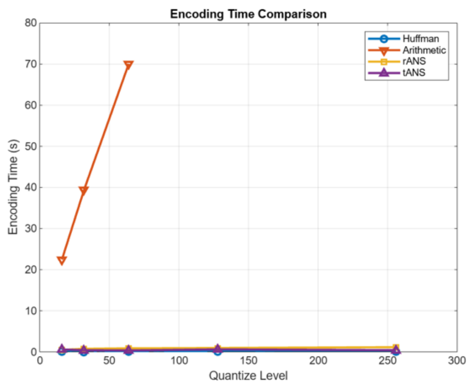
3.1 Experimental Setup

We used a device equipped with a 12th Gen Intel(R) Core(TM) i7-12700H 2.70 GHz processor and 8.0 GB of RAM. For implementation, we considered several programming languages, including Python, Cpp and Matlab. Ultimately, we chose Python for our code to be object-oriented and Python offers superior capabilities for vector operations.

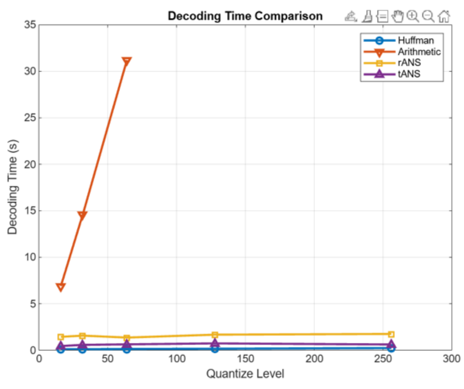
Despite our choice of Python, we also attempted implementation in Matlab, as it is the language we have focused on throughout the semester. However, we found no significant difference between our Python and Matlab implementations.

3.2 Results

3.2.1 Entropy Coding



(a) Encode Runtime



(b) Decode Runtime

Figure 2: Runtime Comparison

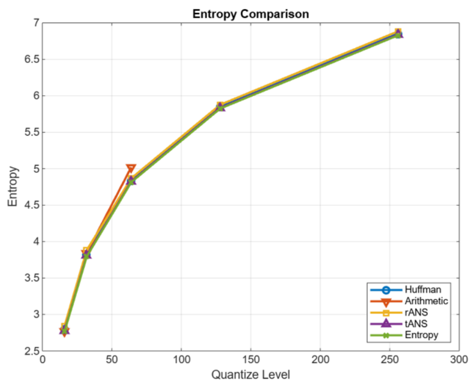


Figure 3: Compression Rate Comparison

For this experiment, we compare various entropy encoding schemes. Figure 2 illustrates the runtime comparison, while Figure 3 depicts the compression rate comparison. We utilized the sound data `"Handel.wav"` from a previous lab as input. The x-axis represents the different quantization levels applied to the sound data.

Furthermore, to demonstrate the compression capabilities of different encoding schemes, we introduced an additional input where we can manipulate the probability of the most probable symbol. Figure 4 illustrates the compression rate at various probabilities, with the x-axis ranging from 0.5 to 0.99.

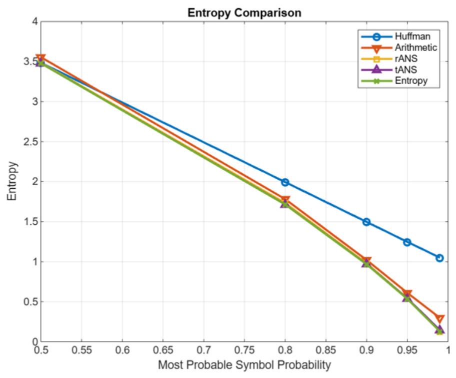


Figure 4: Compression Rate vs Probability

3.2.2 Relative Entropy Coding

In most real-world applications, we do not know the probability distribution of the symbols $p(s_i) := P_S(s_i)$, but we could use the received context to approximate the probability distribution as $q(s_i)$. This is also known as the *relative entropy coding* scheme, which compared to normal encoding, would lead to an error of magnitude bounded by the KL divergence of two distributions $O(D_{KL}(p(x) \parallel q(x)))$. Figure 5 illustrates the codeword length of rANS and tANS with x-axis being the number of leading symbol (context) used. Table 1 shows the KL Divergence versus numbers of symbols used.

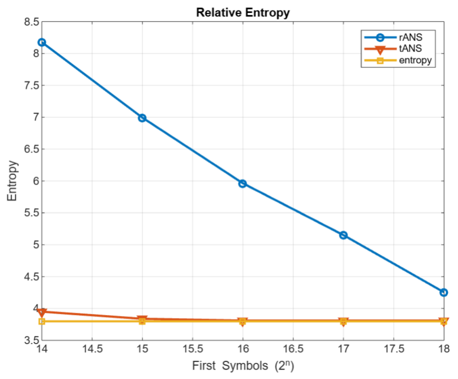


Figure 5: Relative Entropy rANS vs tANS

Leading Symbol	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}
KL divergence	0.1056	0.0249	0.0055	0.0056	0.0021

Table 1: KL Divergence

3.2.3 LZW

We also evaluate the LZW encoding algorithm with the same audio file and device. Note that there are "Entropy" defined like above, and "Entropy Rate" defined by a random process with a Markov chain.

Quantize Levels	16	32	64	128	256
Average Codeword Length	1.076	1.686	2.349	3.433	4.709
Entropy	2.771	3.798	4.816	5.826	6.831
Entropy Rate	0.798	1.303	2.036	2.932	3.891

Table 2: LZW Performance

3.3 Discussion

3.3.1 Entropy Coding

Before discussion, let's quickly go through the pros and cons of entropy coding schemes we mentioned in preliminary.

Encoding Scheme	Huffman	Arithmetic	rANS	tANS
Efficiency	Good	Bad	Good	Good (Better than rANS)
Quality	Bad	Good	Good	Good

Table 3: Entropy Encoding Comparison

Now, first examine Figure 2. At first glance, it is evident that arithmetic coding consumes the most runtime for both encoding and decoding. It takes so much time when the quantization level exceeds 128 that we only show results for lower levels. This aligns with our hypothesis that arithmetic coding requires the most time among these encoding schemes due to its complex arithmetic operations.

The remaining three encoding schemes consume nearly the same amount of time for encoding. However, there are differences in decoding times. Huffman coding consumes the least time, followed by tANS, with rANS taking the most time among the three. This observation aligns with the fact that Huffman coding is the fastest, while tANS is quicker than rANS due to its use of a look-up table.

In Figure 3, we can observe that the codeword lengths for all four encoding schemes approach the lower bound entropy as predicted by Shannon's source coding theorem. This indicates that all four schemes are effective in compressing the sound data.

Next, let's examine Figure 4, where we vary the probability of the most probable symbol. This experimental design highlights the limitations of Huffman coding. Huffman coding requires at least 1 bit to encode each symbol, whereas theoretical entropy can be less than 1 bit. This limitation is evident when considering the formula for information:

$$I(s_k) = \log_2(\frac{1}{p_k}) \tag{9}$$

If the probability is 0.5, the information content equals 1 bit, meaning each symbol requires 1 bit to encode. However, if the probability exceeds 0.5, the information content drops below 1 bit. This makes Huffman coding suboptimal in such cases. As shown in the figure, when the probability reaches 0.99, the codeword length for Huffman coding remains above 1 bit, while the actual entropy is much lower. In contrast, the other three schemes can closely approach the actual entropy, demonstrating the advantage of arithmetic coding and ANS.

We have encountered some difficulties during the implementation of arithmetic coding. The precision of the arithmetic operations of floating point numbers make the decoded sequence different from the original one. For this issue, we used *Decimal* module of python, which can provide arithmetic operations for floating point numbers with any precision. But we encountered another problem: the increasing of precision will make time efficiency drop rapidly. Finally we divides the sequence into windows. By doing this we can reach a good compression rate while keeping a reasonable execution time.

3.3.2 Relative Entropy coding: Towards universal source coding?

In our experimental result, we can see that as the KL divergence increases, the average codeword length increases. We have also found a new result that tANS shows greater stability: compared to rANS, when using a small window for distribution estimation, the average codeword length of rANS grows much faster compared to tANS. We suggest that this may due to the fact that rANS has much more arithmetic operations based on the probability distribution; whereas tANS would first distribute the symbols to the state, and this process might dilute the difference between actual distribution and the sampled distribution.

We also note that the predicted probability distribution could be more *adaptive*, i.e., continues to minimizes the KL divergence as more symbols are received. This may be a potential roadmap to *universal* source coding is a source coding scheme where $p(s_i)$ no need to be known in advance. This is currently combined with modern ML approaches using variational autoencoder (VAE), such as bit-back encoding. We also note that LZW is also a universal source coding scheme using the dynamic dictionary look-up.

3.3.3 LZW

From the evaluated result above, we can see that LZW has a lower average codeword length than the entropy. Since LZW doesn't use the probability distribution for encoding, the theoretical limit of it is not the entropy, but the entropy rate, which measures the information of a source generates symbols that statistically depend on the past (memory). For these sources with memory, as there are more redundancy within, thus it could be compressed further compared to other schemes.

4 Conclusion

In this project, we implemented several entropy encoding schemes and analyzed the trade-off between efficiency and quality. The experimental results matched our hypothesis. In conclusion, the use of Asymmetric Numeral Systems (ANS) not only speeds up the entire encoding process but also maintains high quality under extreme conditions. It is no surprise that ANS has become the favored compression method for many companies, including Google, Apple, Facebook, and Nvidia.

5 Appendix

5.1 Code Usage

Please refer to our `README.md` file for detailed instructions on how to use our code.

1. `main.py`: Usage of all encoding scheme.
2. `util.py`: Other utility functions, including the generation of sequence based on skewed distribution, and the calculation of the entropy rate.
3. `Huffman.py`: Implements the Huffman algorithm.
4. `Arithmetic.py`: Implements the arithmetic coding algorithm.
5. `LZW.py`: Implements the LZW algorithm.
6. `tANS.py`: Implements the tANS algorithm.
7. `rANS.py`: Implements the rANS algorithm.

5.2 Division of Work

B09901069 : Huffman, LZW, AC
B09901099 : tANS, main
B09901156 : rANS, util

References

[1] J. Duda, “Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding,” 2014.

[2] J. Townsend, “A tutorial on the range variant of asymmetric numeral systems,” 2020.

[3] J. Duda, “The use of asymmetric numeral systems as an accurate replacement for huffman coding,” 2015.