Arithmetic Coding for Joint Source-Channel Coding

A dissertation submitted to the University of Malta
in partial fulfilment of the award of
Master of Science in the Faculty of Engineering

2008

Trevor Spiteri

Department of Computer and Communications Engineering

Unless otherwise acknowledged, the content of this dissertation is the original work of the author. None of the work in this dissertation has been submitted by the author in support of an application for another degree or qualification at this or any other university or institute of learning.

The views in this document are those of the author and do not in any way represent those of the university.

Abstract

In this project, a joint source-channel coding technique involving arithmetic coding is implemented. The maximum *a posteriori* (MAP) estimation approach developed by M. Grangetto et al. is implemented and analysed. In this approach, a forbidden symbol is introduced into the arithmetic coder to improve the error-correction performance. According to Grangetto, the error-correction performance obtained is better than a separated source and channel coding approach based on arithmetic codes without forbidden gaps and rate-compatible punctured convolutional (RCPC) codes.

The placement of the forbidden symbol was investigated and modified to decrease the delay from the introduction of an error to the detection of the error. The arithmetic decoder was also modified for quicker detection of introduced errors. Another improvement was obtained by changing the way the prior probability component of the MAP metric is calculated.

The improvements in the system are simulated for both hard and soft decoding. An improvement of up to 0.4 dB for soft decoding and 0.6 dB for hard decoding over the scheme by Grangetto can be seen.

Acknowledgements

I wish to thank Dr. Ing. Victor Buttigieg for his helpful suggestions, and for the valuable feedback he provided, even when I gave him little time to do so.

This work has also benefited from the work of numerous authors of free software which I have used during both the development of the project and the writing of this dissertation. Thanks go to the authors of the GNU/Linux operating system, GNU Emacs, GCC, LaTeX 2_ε, Inkscape, gnuplot, and the various other tools I used.

My final word of thanks is to my family for their patience throughout the project, especially during the later stages, and to Kevin for giving feedback on the draft.

Contents

List of Figures

List of Tables

Glossary of Terms and Abbreviations

ε	The forbidden gap factor
E_b/N_0	Signal-to-noise ratio per bit
$H(U)$	Entropy of the source U
L	The number of symbols in a sequence of source symbols
$[low, high)$	The interval $low \leq x < high$
M	The maximum number of nodes stored in stack during stack decoding
N	The number of bits in a transmitted bit sequence
$P(t \mid y)$	Probability that bit t was transmitted given that signal y is received
$P(y \mid t)$	Probability that signal y is received given that bit t is transmitted
$\mathbf{t}$	Transmitted bit sequence
t_n	Bit number n in the transmitted bit sequence
$\mathbf{u}$	Sequence of source symbols
u_l	Symbol number l in the sequence of source symbols
$\mathbf{y}$	Vector containing received signals
y_n	Received signal corresponding to transmitted bit number n

AWGN	Additive white Gaussian noise
BCJR algorithm	Bahl-Cocke-Jelinek-Raviv algorithm
BPSK	Binary phase-shift keying
MAP	Maximum *a posteriori*
PER	Packet error ratio
RCPC code	Rate-compatible punctured convolutional code

Chapter 1

Introduction

Digital communications [1] refers to the transmission of digital information from a source to one or more destinations. Figure 1.1 shows the basic elements of a digital communication system.

The source encoder converts the source information into digital data, in as few bits as possible. The source encoder compresses the data to remove unwanted redundancy in order to make efficient use of the transmission channel.

The channel encoder introduces redundancy in a controlled manner into the bit sequence. This redundancy can be used later on by the channel decoder to overcome the effects of noise and interference in the transmission of the data. Thus, the redundancy increases the reliability of the system.

The digital modulator and demodulator serve as an interface to the communication channel. The modulator converts the binary data received from the channel encoder into signals which are transmitted over the channel.

The communication channel is the physical medium over which the signals are transmitted. There are many kinds of physical channels, including wire lines, optical fibre cables, and wireless channels. On the channel, the transmitted signal may be

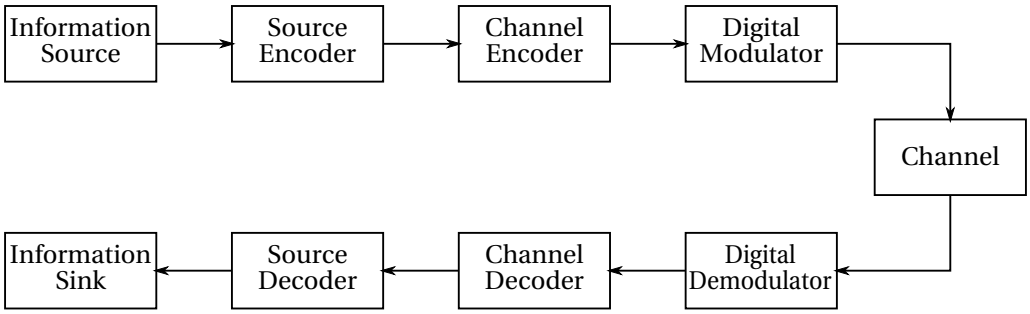

Figure 1.1: Elements of a digital communication system.

corrupted by noise and interference.

At the receiving end, the demodulator converts the signal received from the channel into numbers, which will be passed to the channel decoder.

The channel decoder uses these numbers to reconstruct the original data encoded by the channel encoder. To do its job, the decoder has to know the scheme used by the encoder. The effectiveness of the channel code is measured by the number of errors that occur in the decoded sequence.

The source decoder then converts the digital data decoded by the channel decoder into the form required by the target.

Arithmetic coding [2] is a source coding method for lossless data compression. It is a variable-length coding scheme, which means that the length of the bit sequence produced by the arithmetic encoder depends on which particular source symbols are passed to the encoder. Arithmetic coding performs better than Huffman codes [3], another kind of variable-length coding scheme, in almost every aspect. It represents data more compactly and adapts better to adaptive models.

Arithmetic codes were introduced by Rissanen [2] in 1976 as an alternative to Huffman codes. In the 1980's, practical implementation techniques were presented, and the popularity of arithmetic codes began to grow. Their adoption was somehow hampered by patent protection of practical implementations. Since then, key patents have expired.

Thus, the use of arithmetic codes is increasing once more, especially in multimedia applications, where the high data rates make the high compression performance of arithmetic codes very desirable. Multimedia compression usually employs several stages. In the first stages, lossy compression schemes are used. These schemes are dependent on the medium and remove nonessential features from the source. After this, the data is quantized, and finally the quantized data is passed to an entropy encoder. Entropy encoding is a lossless compression scheme which is independent of the characteristics of the medium. New multimedia compression schemes are increasingly using arithmetic coding as this final stage of the source coding system.

Multimedia data is transmitted over different channels. Transmission of multimedia over wireless channels is increasing. The wireless medium is susceptible to noise and interference, and needs powerful channel coding techniques to be useful in digital communication systems. If the bit sequence generated by the arithmetic encoder is not protected by a suitable channel code, the arithmetic decoder will not be able to reconstruct the original data. Hence, suitable channel coding techniques are essential for arithmetic codes to be transmitted over channels such as the wireless channel.

Data transmission requires both source coding and channel coding. Source coding is required for efficient use of the channel, and channel coding is required for

reliable transmission of data. Joint source-channel coding techniques are emerging as a good choice to transmit digital data over wireless channels.

Shannon's source-channel separation theorem suggests that reliable data transmission can be accomplished by separate source and channel coding schemes, where the source encoder does not need to take the channel characteristics into account, and the channel encoder does not need to take the source characteristics into account. Vembu et al. [4] point out shortcomings of the separation theorem when dealing with non-stationary probabilistic channels. The bandwidth limitations of the wireless channels, and the stringent demands of multimedia transmission systems are emphasizing the practical shortcomings of the separation theorem. In practical cases, the source encoder is not able to remove all the redundancy from the source. Joint source-channel coding techniques can exploit this redundancy to improve the reliability of the transmitted data.

Joint schemes can also provide implementation advantages. For example, Boyd et al. [5] proposes a joint source-channel coding technique using arithmetic codes, and mentions several advantages, including (a) saving on software, hardware, or computation time by having a single engine that performs both source and channel coding, (b) the ability to control the amount of redundancy easily to accommodate prevailing channel conditions, and (c) the ability to perform error checking continuously as each bit is processed.

1.1 Objectives

Joint source-channel codes using arithmetic codes have been receiving more attention, so a review of the current research was in order.

The aim of this project was to implement a joint source-channel coding scheme that uses arithmetic codes, to check that the implemented scheme performs as shown in published results, and to possibly improve the scheme.

1.2 Organization

In Chapter 2, arithmetic coding and common implementation techniques are reviewed. The forbidden symbol technique used for error detection is reviewed in this chapter as well.

In Chapter 3, joint source-channel coding is discussed. The chapter deals with the decoding problem and MAP decoding, and decoding metrics for both hard and soft decoding are reviewed. The sequential search technique used is also described.

In Chapter 4, some improvements of error correction of arithmetic codes are presented. The placement of the forbidden symbol in the forbidden symbol technique is discussed. A modification to the decoder that allows for earlier detection of errors by looking ahead is presented in this chapter as well. Finally, an improvement in the continuous calculation of the MAP decoding metric for MAP decoding is presented.

The implementation of the joint source-channel coding system is presented in Chapter 5. The results are then presented in Chapter 6.

Finally, Chapter 7 draws some conclusions and mentions some directions for future work.

An overview of the project source code is available in Appendix A. An attached CD contains the source code.

Chapter 2

Arithmetic Coding

Arithmetic coding is a method for compressing a message **u** consisting of a sequence of L symbols $u_1, u_2, \cdots, u_L$ with different probability of occurrence. This method requires that we have a good source model. The model gives us the distribution of probabilities for the next input symbol. The encoder and decoder must have access to the same source model.

The model can be a static model, that is, the distribution of probabilities does not change from symbol to symbol. Alternatively, the model can be adaptive, where the distribution of probabilities for each symbol can be different. In this case, the encoder's model will change with each encoded symbol. The decoder's model will be updated similarly with each decoded symbol.

The compression performance of arithmetic coding depends on the accuracy of the source model. More complex models can provide better accuracy, which leads to greater compression. Given a perfect source model, arithmetic coding is optimal. The better the model is, the more optimal the arithmetic code is.

2.1 Representing a message as an interval

In arithmetic coding, a message is represented by an interval of real numbers between 0 and 1. For each source symbol, the encoder needs to know the current interval and the probability of each possible source symbol. The current interval is split into a number of parts, one for each symbol. The length of each part of the interval is proportional to the probability of the symbol it represents. Thus, encoding a likely symbol will reduce the interval by a small factor, and encoding an unlikely symbol will reduce the interval by a large factor.

At the start of the encoding process, the interval is the half-open interval [0, 1), that is, $0 \leq x < 1$. The encoding process that reduces this interval to the final interval is illustrated in Example 2.1 below.

Table 2.1: Symbol interval ranges for Example 2.1.

Symbol	Probability	Range
a	0.3	[0, 0.3)
b	0.5	[0.3, 0.8)
c	0.2	[0.8, 1)

Table 2.2: Steps for finding the interval for Example 2.1.

Step	Symbol	Interval Before	Interval After
1	b	[0, 1)	[0.3, 0.8)
2	a	[0.3, 0.8)	[0.3, 0.45)
3	c	[0.3, 0.45)	[0.42, 0.45)
4	b	[0.42, 0.45)	[0.429, 0.444)
5	b	[0.429, 0.444)	[0.4335, 0.441)
6	a	[0.4335, 0.441)	[0.4335, 0.43575)

Example 2.1. Suppose we have a symbol source with an alphabet consisting of three symbols, a, b, and c. We are required to find the interval that represents the input sequence $bacbba$. The source model is a static model. The probabilities and interval ranges can be seen in Table 2.1.

Since we have six input symbols, the process requires six steps. We start with the interval [0, 1). To encode the first symbol in the sequence, the encoder splits the interval into three sub-intervals. Symbol a gets [0, 0.3), symbol b gets [0.3, 0.8), and symbol c gets [0.8, 1). The first symbol to encode is b, so the sub-interval chosen is [0.3, 0.8).

To encode the second symbol, the interval [0.3, 0.8) is split into three sub-intervals. Since this interval is not [0, 1), we have to scale the sub-intervals to fit into our interval. If the current interval is [low, $high$), each endpoint x will be scaled as

$$x_{\text{scaled}} = low + (high - low)\,x.$$

In our case, $low = 0.3$ and $high = 0.8$. Scaling the endpoints of the three sub-intervals, symbol a gets the sub-interval [0.3, 0.45), symbol b gets [0.45, 0.7), and symbol c gets [0.7, 0.8). The second symbol in our example is a, so the interval chosen is [0.3, 0.45).

This process has to be repeated for each symbol. Table 2.2 shows each step in the process. Figure 2.1 shows a graphical representation of the whole process. After each symbol, the interval in the figure is scaled up, so that we can see the smaller intervals. The final interval is [0.4335, 0.43575). The width of this interval is

$$\begin{aligned} \textit{Interval width} &= 0.43575 - 0.4335 \\ &= 0.00225. \end{aligned}$$

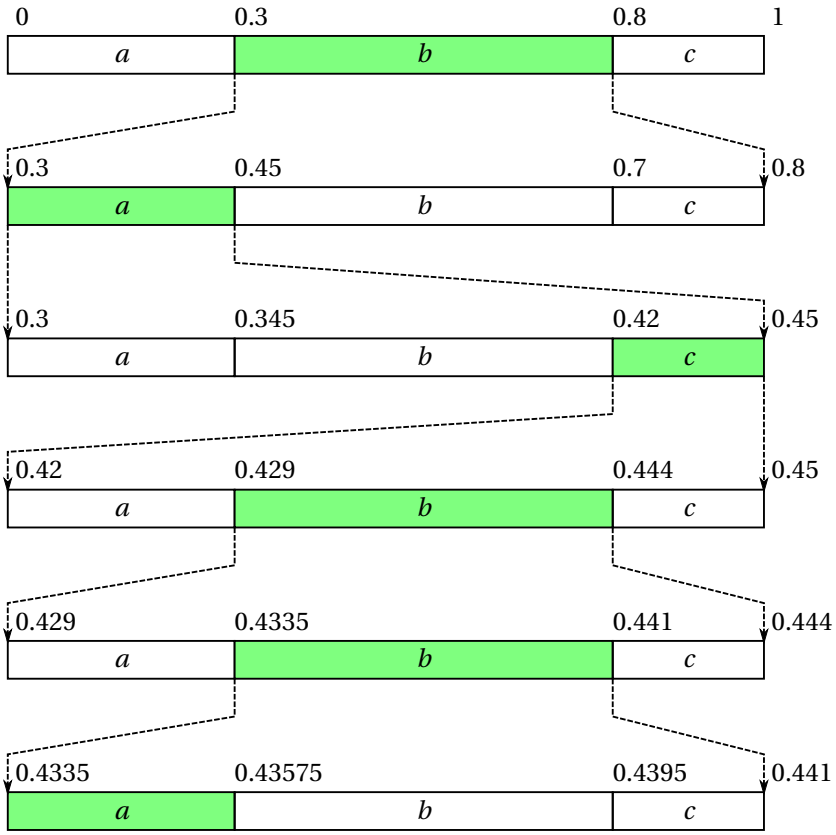

Figure 2.1: Finding the interval for Example 2.1.

Alternatively, the interval width can be obtained by multiplying the probability of each encoded symbol.

$$\begin{aligned} \textit{Interval width} &= P(b) \cdot P(a) \cdot P(c) \cdot P(b) \cdot P(b) \cdot P(a) \\ &= 0.5 \times 0.3 \times 0.2 \times 0.5 \times 0.5 \times 0.3 \\ &= 0.00225 \end{aligned}$$

2.2 Incremental encoding

The encoder's job is to convert a sequence of symbols into an interval, and to output a binary sequence which specifies the interval.

As the interval gets smaller, more precision is needed to specify it. Consequently, as more symbols are encoded, more bits are required to specify the interval. It also follows that encoding a likely symbol will result in less output bits than encoding an unlikely symbol, because encoding a likely symbol reduces the interval less.

As more symbols are encoded, the interval gets smaller and smaller, that is, the difference between the lower and upper endpoints becomes very small. Owing to the limited precision used for computations, a very small interval will case the process to fail. This makes it clear that we cannot find the interval corresponding to a whole sequence of symbols in a very long message.

To solve this problem, we use incremental encoding as shown by Witten et al. [6]. If the upper endpoint is less than or equal to 1/2, we can output a `0` and double the endpoints. Doubling the endpoints with the transformation $x \to 2x$ scales the range [0, 1/2) to the range [0, 1). For example, if the interval is [0.1, 0.4), we can output a `0` and change the interval to [0.2, 0.8).

Similarly, if the lower endpoint is greater than or equal to 1/2, we can output a `1` and scale the range [1/2, 1) to the range [0, 1) with the transformation $x \to 2x - 1$. For example, if the interval is [0.5, 0.85), we can output a `1` and change the interval to [0, 0.7).

This solves only a part of the precision problem. If the endpoints are very close and straddle 1/2, for example, if the interval is [0.49, 0.51), we can neither output a `0` nor a `1`. We solve this problem using another technique presented in [6].

Suppose the lower endpoint *low* and the upper endpoint *high* of the interval [*low*, *high*) lie in the range

$$1/4 \le \textit{low} < 1/2 < \textit{high} \le 3/4.$$

Then, the next two bits will be different, that is, they will be either `01` or `10`. If the first bit turns out to be `0`, that is, *high* descends below 1/2, then the range [0, 1/2) is

Table 2.3: Steps for Example 2.2.

Step	Action	Interval	*follow*
1	Encode *b*	[0.3, 0.8)	0
2	Encode *a*	[0.3, 0.45)	0
3	Output `0`	[0.6, 0.9)	0
4	Output `1`	[0.2, 0.8)	0
5	Encode *c*	[0.68, 0.8)	0
6	Output `1`	[0.36, 0.6)	0
7	Increment *follow*	[0.22, 0.7)	1
8	Encode *b*	[0.364, 0.604)	1
9	Increment *follow*	[0.228, 0.708)	2
10	Encode *b*	[0.372, 0.612)	2
11	Increment *follow*	[0.244, 0.724)	3
12	Encode *a*	[0.244, 0.388)	3
13	Output `0111`	[0.488, 0.776)	0

expanded to the range [0, 1). But before the expansion, $low \geq 1/4$, so that after the expansion, $low \geq 1/2$. This means that the second bit will be `1`. Similarly, if the first bit turns out to be `1`, the second bit will be `0`. Thus, we can remember this and expand the range [1/4, 3/4) to the range [0, 1) with the transformation $x \rightarrow 2x - 1/2$. If after the expansion, we find ourselves in the same situation, but before outputting the two opposite bits, that is,

$$1/4 \leq low < 1/2 < high \leq 3/4$$

once more, we simply perform another expansion and keep a count of the number of expansions performed. This count is called the *follow* count. When we finally find a bit to output, a number equal to *follow* of bits opposite to the found bit are outputted. For example, if $follow = 3$ and we can output a `0`, we follow this bit by three `1`s.

Example 2.2. For the input sequence $bacbba$ of Example 2.1, we need to output all the bits that we can.

Table 2.3 shows the whole process for the six symbols. In Step 1, we start with the interval [0, 1) and encode the symbol *b*. The interval becomes [0.3, 0.8). We cannot output any bits yet.

In Step 2, we encode the second symbol, *a*, and the interval becomes [0.3, 0.45). We can see that $high \leq 1/2$, so in Step 3 we can output a `0` and expand the interval to [0.6, 0.9). Now $low \geq 1/2$, so in Step 4 we can output a `1` and expand the interval to [0.2, 0.8).

Now let us move to Step 7. Here, $1/4 \leq low < 1/2 < high \leq 3/4$, so we increment the follow count so that $follow = 1$ and expand the interval to [0.22, 0.7). In Step 9, once again we have $1/4 \leq low < 1/2 < high \leq 3/4$. We increment the follow count again so

that *follow* = 2 and expand the interval to [0.228, 0.708).

The follow count is reset to 0 after outputting a `0` or `1`. This happens in Step 13. Here, since $high \leq 1/2$, we can output a `0`. But since $follow \neq 0$, we cannot output just a `0`; we output a `0` followed by three `1`s and reset *follow* to 0.

After this process, we have outputted seven bits, `0110111`. The width of the remaining interval is

$$\begin{aligned} \textit{Interval width} &= 0.776 - 0.488 \\ &= 0.288. \end{aligned}$$

This is larger than the width of the final interval in Example 2.1 by a factor of $2^7 = 128$.

In Example 2.2 above, the interval was prevented from getting too narrow using incremental encoding. This enables us to use arithmetic coding when we have limited precision in our computation. During the calculation of the interval, seven bits were output. However, the encoding process is not ready yet; we still need to terminate the output bit sequence.

2.3 Termination

When all symbols are encoded, we need to terminate the output bit sequence. If the input source has a fixed number of symbols, no extra symbol is required. However, if the number of symbols is not fixed, an end-of-sequence (EOS) symbol is required. This can be achieved by allocating a tiny part of the interval for an EOS symbol.

After the EOS symbol is encoded, we need to make sure that the interval specified by the transmitted bit sequence falls completely within the interval of the input symbols. The decoder does not need to know the upper and lower endpoints of the interval exactly; it is enough to pass a sub-interval which lies between the two endpoints. Two additional bits are enough to ensure this.

Incremental encoding ensures that during encoding, the current interval either includes the range [1/4, 1/2), or includes the range [1/2, 3/4). In the former case, outputting the bits `0` and `1` will be enough for the decoder to be able to decode all the symbols; in the latter case, the bits `1` and `0` can be used. Note that if the follow count is not zero, when we output the first of these two termination bits we must follow it by a number of opposite bits.

In Example 2.2, after encoding six symbols, the interval was [0.488, 0.776). This interval does not include the range [1/4, 1/2), but it does include the range [1/2, 3/4).

So if we need to terminate this sequence (assuming that the number of symbols is fixed at six), all we need to do is to output `10`. Thus, the whole sequence of Example 2.2 can be encoded using the bit sequence `011011110`.

2.4 Achievable compression

Consider a source U with an alphabet consisting of K symbols $U_1, U_2, \cdots, U_K$. The information content of each symbol U_k measured in bits is

$$I(U_k) = -\log_2 P(U_k)$$

where $P(U_k)$ is the probability that $U = U_k$.

The entropy of the source U is the expected value of $I(U)$, that is, the entropy is

$$\begin{aligned} H(U) &= E(I(U)) \\ &= -\sum_{k=1}^{K} P(U_k) \log_2 P(U_k). \end{aligned}$$

We can think of the entropy as a measure of uncertainty for the source; the maximum entropy is when the symbols in the source are equally probable.

The minimum number of bits per symbol required to encode a source is equal to the entropy of the source [2]. If the source model available is accurate, arithmetic coding can represent a source sequence $u_1, u_2, \cdots, u_L$ optimally, in an average of $L \cdot H(U)$ bits. The maximum achievable compression using arithmetic codes is

$$\text{Maximum achievable compression} = \frac{\text{Source entropy}}{\text{Average number of bits per source symbol}}.$$

This is illustrated in Example 2.3 below.

Example 2.3. Suppose we have a source with an alphabet consisting of four symbols, a, b, c and d. The probabilities of the symbols are $P(a) = 0.2$, $P(b) = 0.7$, $P(c) = 0.05$ and $P(d) = 0.05$. We are required to find how much arithmetic coding can compress this source.

Since the source has four symbols, the number of bits required per symbol for the uncompressed source is 2. We represent a as `00`, b as `01`, c as `10`, and d as `11`.

The entropy of this source is

$$\begin{aligned} H(U) &= -\sum_{k=1}^{4} P(U_k) \log_2 P(U_k) \\ &= -0.2 \log_2 0.2 - 0.7 \log_2 0.7 - 0.05 \log_2 0.05 - 0.05 \log_2 0.05 \\ &= 1.26. \end{aligned}$$

If we encode the source using arithmetic coding, each symbol will need an average of 1.26 bits.

The compression ratio achieved in this case is

$$\begin{aligned} \text{Compression} &= \frac{1.26}{2} \\ &= 0.63. \end{aligned}$$

2.5 Decoding

The decoding process is similar to the encoding process. We start with the half-open interval [0, 1). This interval is split into a number of sub-intervals, one for each possible source symbol, in exactly the same way as in the encoder.

Then, we start looking at the bit sequence being decoded. We shall call the interval specified by the bit sequence the input interval. The input interval starts as [0, 1) too. For each bit we decode, we can reduce the input interval by half. If the bit is `0`, we keep the lower half; if the bit is `1`, we keep the higher half.

If the input interval lies completely in one of the sub-intervals, the sub-interval which encloses the input interval is chosen. The symbol corresponding to the chosen sub-interval can be decoded immediately.

When the symbol is decoded, the chosen sub-interval becomes the new decoding interval. The decoding interval can be expanded as shown in § 2.2. The expansions should be identical to the expansions used during encoding so as not to introduce any error while rounding. Any expansion applied to the decoding interval should also be applied to the input interval, which always lies within the decoding interval. There is no need to keep a follow count while decoding.

Example 2.4. For the source model of Example 2.1 shown in Table 2.1, we need to decode six symbols from the bit sequence `011011110`.

Table 2.4 shows the whole decoding process.

In Step 1, we initialize the decoder. In Steps 2–4, we use the first three bits, which

Table 2.4: Steps for Example 2.2.

Step	Action	Input interval		a sub-interval		b	c
1	Initialize decoder	[0,	1)	[0,	0.3)	0.8)	1)
2	Use 1st bit, `0`	[0,	0.5)	[0,	0.3)	0.8)	1)
3	Use 2nd bit, `1`	[0.25,	0.5)	[0,	0.3)	0.8)	1)
4	Use 3rd bit, `1`	[0.375,	0.5)	[0,	0.3)	0.8)	1)
5	Decode symbol b	[0.375,	0.5)	[0.3,	0.45)	0.7)	0.8)
6	Use 4th bit, `0`	[0.375,	0.4375)	[0.3,	0.45)	0.7)	0.8)
7	Decode symbol a	[0.375,	0.4375)	[0.3,	0.345)	0.42)	0.45)
8	Expand, $x \to 2x$	[0.75,	0.875)	[0.6,	0.69)	0.84)	0.9)
9	Expand, $x \to 2x-1$	[0.5,	0.75)	[0.2,	0.38)	0.68)	0.8)
10	Use 5th bit, `1`	[0.625,	0.75)	[0.2,	0.38)	0.68)	0.8)
11	Use 6th bit, `1`	[0.6875,	0.75)	[0.2,	0.38)	0.68)	0.8)
12	Decode symbol c	[0.6875,	0.75)	[0.68,	0.716)	0.776)	0.8)
13	Expand, $x \to 2x-1$	[0.375,	0.5)	[0.36,	0.432)	0.552)	0.6)
13	Expand, $x \to 2x-1/2$	[0.25,	0.5)	[0.22,	0.364)	0.604)	0.7)
14	Use 7th bit, `1`	[0.375,	0.5)	[0.22,	0.364)	0.604)	0.7)
15	Decode symbol b	[0.375,	0.5)	[0.364,	0.436)	0.556)	0.604)
16	Expand, $x \to 2x-1/2$	[0.25,	0.5)	[0.228,	0.372)	0.612)	0.708)
17	Use 8th bit, `1`	[0.375,	0.5)	[0.228,	0.372)	0.612)	0.708)
18	Decode symbol b	[0.375,	0.5)	[0.372,	0.444)	0.564)	0.612)
19	Expand, $x \to 2x-1/2$	[0.25,	0.5)	[0.244,	0.388)	0.628)	0.724)
20	Use 9th bit, `0`	[0.25,	0.375)	[0.244,	0.388)	0.628)	0.724)
21	Decode symbol a						

narrows the interval. Now, the input interval lies within the sub-interval for symbol b; in Step 5 we decode the first symbol in the sequence, b.

Similarly, in Steps 6–7 we decode the second symbol in the sequence, a. After we decode this symbol, the three sub-intervals lie in the interval [0.3, 0.45), which can be expanded. In Steps 8–9, we perform the required expansion, taking care to expand the input interval as well.

This process goes on until we decode all the six symbols, $bacbba$. These are the same symbols used in Example 2.2, demonstrating that decoding a bit sequence will give us the original source sequence.

2.6 Integrating error detection

Arithmetic coding can compress data optimally when the source can be modelled accurately. However, arithmetic codes are extremely vulnerable to any errors that occur. Huffman codes [3] tend to be self-synchronizing [7], that is, errors tend not to propagate very far. When an error occurs in transmission, several codewords are misinterpreted, but before too long, the decoder is back in synchronization with the encoder.

Arithmetic coding, on the other hand, has no ability to withstand errors. Arithmetic coding does not use a codeword for each symbol like Huffman coding; the arithmetic encoder is a continuous encoder that updates the interval for every symbol. Hence, there are no points at which the arithmetic encoder can regain synchronization. If a bit is in error in the bit sequence, from then on, instead of the correct interval we get an incorrect interval. This is illustrated in Example 2.5 below.

Example 2.5. Suppose we want to decode the same bit sequence shown in Example 2.4, but our bit sequence is corrupted by a bit error in the second bit. So the bit sequence we have is `001011110`.

Table 2.5 shows the whole decoding process for this sequence.

The correct symbol sequence should have been $bacbba$. However, because of the bit error in the second bit, we decoded $abbbb$. Synchronization with the encoder is lost, and the arithmetic decoder will go on decoding incorrect symbols.

2.6.1 Reducing the interval on doubling

Boyd et al. [5] propose the introduction of some redundancy. This is done by forbidding a range from the interval. To do this, [5] suggests that the interval be reduced by a factor each time the interval is doubled. As we have seen in § 2.2 above, each interval

Table 2.5: Steps for Example 2.5.

Step	Action	Input interval	a sub-interval	b	c
1	Initialize decoder	[0, 1)	[0, 0.3)	0.8)	1)
2	Use 1st bit, `0`	[0, 0.5)	[0, 0.3)	0.8)	1)
3	Use 2nd bit, `0`	[0, 0.25)	[0, 0.3)	0.8)	1)
4	Decode symbol a	[0, 0.25)	[0, 0.09)	0.24)	0.3)
5	Expand, $x \to 2x$	[0, 0.5)	[0, 0.18)	0.48)	0.6)
6	Use 3rd bit, `1`	[0.25, 0.5)	[0, 0.18)	0.48)	0.6)
7	Use 4th bit, `0`	[0.25, 0.375)	[0, 0.18)	0.48)	0.6)
8	Decode symbol b	[0.25, 0.375)	[0.18, 0.27)	0.42)	0.48)
9	Expand, $x \to 2x$	[0.5, 0.75)	[0.36, 0.54)	0.84)	0.96)
10	Use 5th bit, `1`	[0.625, 0.75)	[0.36, 0.54)	0.84)	0.96)
11	Decode symbol b	[0.625, 0.75)	[0.54, 0.63)	0.78)	0.84)
12	Expand, $x \to 2x-1$	[0.25, 0.5)	[0.08, 0.26)	0.56)	0.68)
13	Use 6th bit, `1`	[0.375, 0.5)	[0.08, 0.26)	0.56)	0.68)
14	Decode symbol b	[0.375, 0.5)	[0.26, 0.35)	0.5)	0.56)
15	Expand, $x \to 2x-1/2$	[0.25, 0.5)	[0.02, 0.2)	0.5)	0.62)
16	Decode symbol b	[0.25, 0.5)	[0.2, 0.29)	0.44)	0.5)
17	Expand, $x \to 2x$	[0.5, 1)	[0.4, 0.58)	0.88)	1)
18	Use 7th bit, `1`	[0.75, 1)	[0.4, 0.58)	0.88)	1)
19	Use 8th bit, `1`	[0.875, 1)	[0.4, 0.58)	0.88)	1)
20	Use 9th bit, `0`	[0.875, 0.9375)	[0.4, 0.58)	0.88)	1)

doubling denotes the processing of an output bit.

So in effect, for every output bit, some redundancy is introduced into the arithmetic code. This scheme uses a reduction factor R, which is chosen beforehand. Whenever the interval is expanded, the lower endpoint *low* is retained, but the upper endpoint *high* is changed such that the interval width $high - low$ is reduced by the reduction factor R. The reduced higher endpoint is $high_{\text{reduced}} = low + R(high - low)$. The new interval is $[low, high_{\text{reduced}})$. The other part of the interval, $[high_{\text{reduced}}, high)$, is forbidden; if a bit sequence leads the decoder into that part of the interval, an error is detected. This will be illustrated using an example.

Example 2.6. During encoding, the current interval becomes [0.68, 0.8). We need to expand this interval. For each doubling of the interval, we need to reduce the interval by a reduction factor $R = 0.99$.

The starting interval is $[low, high) = [0.68, 0.8)$. Since $low \geq 1/2$, we double the interval with the transformation $x \to 2x - 1$. The interval becomes [0.36, 0.6). The interval width is $0.6 - 0.36 = 0.24$. We need to reduce this width by the factor R, so the new width is $0.99 \times 0.24 = 0.2376$. We obtain this by setting $high = 0.5976$. The reduced interval becomes [0.36, 0.5976).

Now, $1/4 \leq low < 1/2 < high \leq 3/4$, so we need to double the interval using the transformation $x \rightarrow 2x - 0.5$. The interval becomes [0.22, 0.6952). Again, the interval width has to be reduced to $0.99 \times (0.6952 - 0.22) = 0.4704$. We obtain this by setting $high = 0.6904$, with the final interval becoming [0.22, 0.6904).

This scheme is a joint source-channel coding method, as it combines source coding used for compression with channel coding used to protect from errors.

When this redundancy is introduced, we will be able to detect errors in the decoder. The decoder will reduce its interval in exactly the same way as the encoder. Eventually, if an error is present in the bit sequence passed to the decoder, the interval specified by the bit sequence will fall into a forbidden part of the interval. This will not happen immediately, but after some delay. The delay from the bit in error till the detection of the error is shown to be about $1/(1-R)$ in [5].

2.6.2 Introducing a forbidden symbol

Instead of rescaling the interval for every output bit, Sayir [8] suggests introducing forbidden gaps in the interval for each source symbol. After each symbol is encoded, the source probabilities are rescaled by a factor γ. On average, for every symbol encoded, $-\log_2 \gamma$ bits of redundancy are added. So for a source U with entropy $H(U)$, the code rate R can be expressed as

$$R = \frac{H(U)}{H(U) - \log_2 \gamma}.$$

Instead of working with the rescaling factor γ, we can work with the gap factor $\varepsilon = 1 - \gamma$.

Example 2.7. Suppose we have to find the interval for the input sequence *bacbba* of Example 2.1, but this time we want to introduce some redundancy. We need to introduce a gap after each symbol, with $\varepsilon = 0.1$.

To do this, we can modify the table of interval ranges. The modified ranges can be seen in Table 2.6.

We start with the interval [0, 1). To encode the first symbol in the sequence, *b*, the encoder narrows the interval to [0.27, 0.72). To encode the second symbol, *a*, the encoder narrows the interval to [0.27, 0.3915).

Table 2.7 shows the encoding steps. Figure 2.2 shows a graphical representation of the process. The final interval is [0.3660, 0.3672). The width of the interval is 0.0012. In Example 2.1, the width was 0.00225, which is larger. The width of the interval in

Table 2.6: Symbol interval ranges for Example 2.7.

Symbol	Probability	Range	Scaled Probability	Scaled Range
a	0.3	[0, 0.3)	0.27	[0, 0.27)
b	0.5	[0.3, 0.8)	0.45	[0.27, 0.72)
c	0.2	[0.8, 1)	0.18	[0.72, 0.9)
Forbidden	0		0.1	[0.9, 1)

Table 2.7: Steps for finding the interval for Example 2.7.

Step	Symbol	Interval Before	Interval After
1	b	[0, 1)	[0.27, 0.72)
2	a	[0.27, 0.72)	[0.27, 0.3915)
3	c	[0.27, 0.3915)	[0.3575, 0.3794)
4	b	[0.3575, 0.3794)	[0.3634, 0.3732)
5	b	[0.3634, 0.3732)	[0.3660, 0.3705)
6	a	[0.3660, 0.3705)	[0.3660, 0.3672)

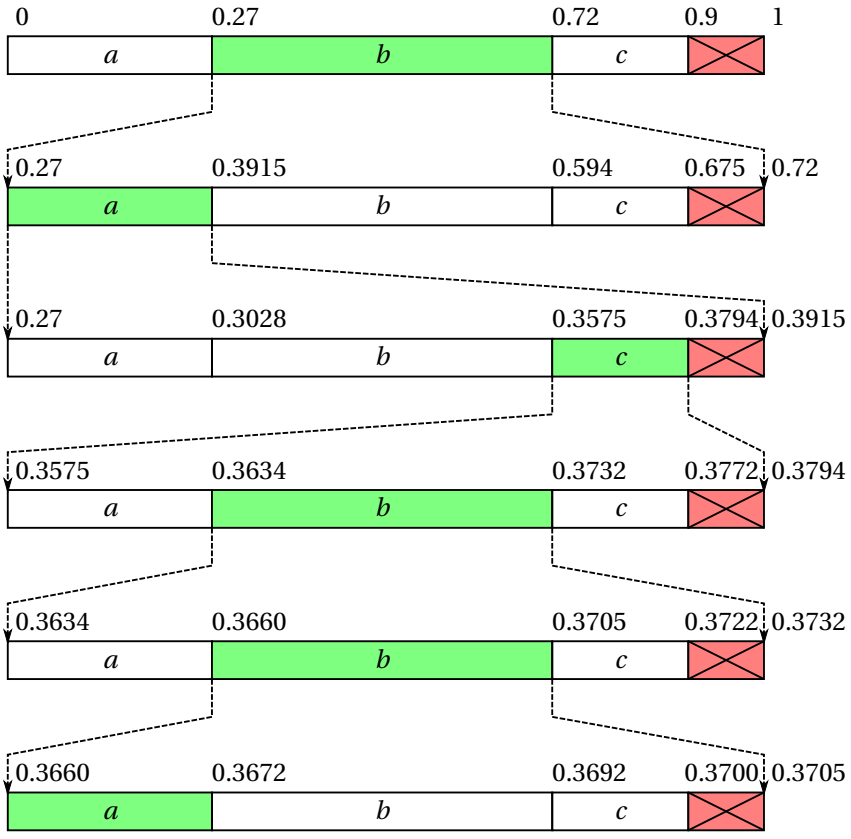

Figure 2.2: Finding the interval for Example 2.7.

this case is smaller by a factor of 0.9^6 owing to the forbidden gap. We need $-6\log_2 0.9$ more bits to encode this interval.

Similar to the scheme in § 2.6.1, when a bit sequence with an error is passed to a decoder the decoder may not detect an error immediately. However, eventually, the interval specified by the bit sequence with errors will fall into one of the forbidden gaps. When this happens, the decoder detects that an error has occurred. It is important to note that there is some delay before error detection.

2.7 Termination and error detection

The forbidden symbol used will detect errors only after some delay. As a consequence, an error near the end of the output bit sequence may remain undetected.

To allay this problem, a termination strategy is used by the arithmetic coder. After all symbols (including an EOS symbol if required) are encoded, the arithmetic encoder will encode a termination symbol. The termination symbol has some small probability ω, and on average will add $-log_2\omega$ bits to the encoded bit sequence.

If the input sequence contains L symbols, the termination symbol will be encoded after symbol u_L. Care has to be taken where to place this termination symbol. If we place it at the beginning or at the end of the interval, it may interfere with the decoding of symbol u_L.

Consider Figure 2.3. In part (a), u_L is b and the termination symbol is placed at the beginning of the interval for symbol b. During decoding, the decoding interval may include a part of the termination symbol and a part of the symbol $u_L = a$. Hence, the decoding of symbol $u_L = b$ will be delayed unnecessarily because the decoder cannot discard the possibility that u_L is a instead of b.

To solve this problem, the termination symbol can be placed in the middle of the interval as shown in Figure 2.3 part (b). Since the interval for the termination symbol is more removed from the interval for $u_L = a$, the decoding of symbol $u_L = b$ is not delayed.

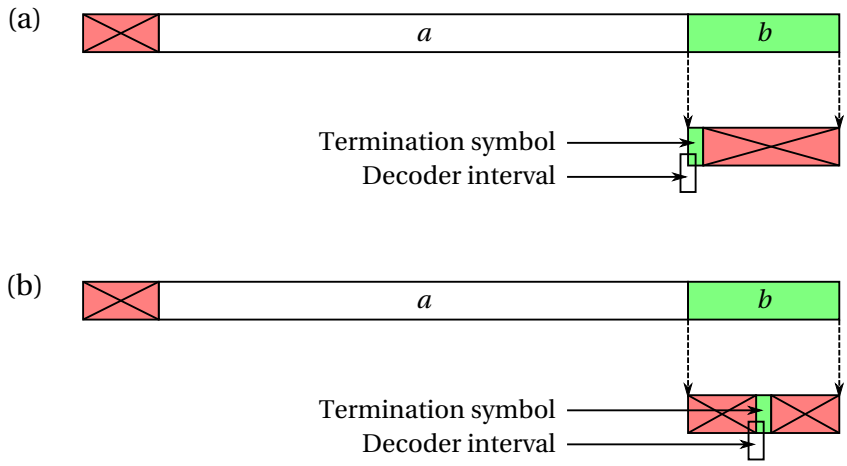

Figure 2.3: Placement of the end-of-sequence symbol (a) at the beginning of the interval and (b) in the middle of the interval.

Chapter 3

Error Correction of Arithmetic Codes

3.1 The decoding problem

In § 2.6, we have already seen a joint source-channel method for detecting errors in arithmetic codes. There is also work on correcting errors using joint source-channel coding.

To perform error correction, we must first encode the symbols with an encoder that introduces redundancy. Suppose we have a message $\mathbf{u}$ consisting of a sequence of L symbols, $u_1, u_2, \cdots, u_L$. We encode this into a bit sequence $\mathbf{t}$, which has N bits, $t_1, t_2, \cdots, t_N$. The bit sequence $\mathbf{t}$ is then transmitted over a noisy channel, and the received signal is $\mathbf{y}$.

Figure 3.1 is a block diagram of the encoding and decoding process. The task of the decoder is to infer the message $\hat{\mathbf{u}}$ given the received signal $\mathbf{y}$. If the inferred message $\hat{\mathbf{u}}$ is not identical to the source message $\mathbf{u}$, a decoding error has occurred.

3.2 MAP decoding

Maximum *a posteriori* (MAP) decoding [9] is the identification of the most probable message $\mathbf{u}$ given the received signal $\mathbf{y}$.

Figure 3.1: Block diagram of the encoding and decoding process.

By Bayes' theorem, the *a posteriori* probability of a message **u** given the signal **y** is

$$P(\mathbf{u}\,|\,\mathbf{y}) = \frac{P(\mathbf{y}\,|\,\mathbf{u})P(\mathbf{u})}{P(\mathbf{y})}. \tag{3.1}$$

Since **u** has L elements and the signal **y** has N elements, it can be convenient to work in terms of the bit sequence **t** instead of the message **u**. Since there is a one-to-one relationship between **u** and **t**, $P(\mathbf{t}) = P(\mathbf{u})$. Thus, we can rewrite (3.1) as

$$P(\mathbf{t}\,|\,\mathbf{y}) = \frac{P(\mathbf{y}\,|\,\mathbf{t})P(\mathbf{t})}{P(\mathbf{y})}. \tag{3.2}$$

The right-hand side of this equation has three parts.

1. The first factor in the numerator, $P(\mathbf{y}\,|\,\mathbf{t})$, is the *likelihood* of the bit sequence (which is equal to $P(\mathbf{y}\,|\,\mathbf{u})$). For a memoryless channel, the likelihood may be separated into a product of the likelihood of each bit, that is,

$$P(\mathbf{y}\,|\,\mathbf{t}) = \prod_{n=1}^{N} P(y_n\,|\,t_n). \tag{3.3}$$

   If we transmit $+x$ for $t_n = 1$ and $-x$ for $t_n = 0$ over a Gaussian channel with additive white noise of standard deviation σ, the probability density of the received signal y_n for both values of t_n is

$$P(y_n\,|\,t_n = 1) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y_n - x)^2}{2\sigma^2}\right) \tag{3.4}$$

$$P(y_n\,|\,t_n = 0) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y_n + x)^2}{2\sigma^2}\right). \tag{3.5}$$

2. The second factor in the numerator, $P(\mathbf{t})$ is the *prior* probability of the bit sequence **t**. In our case, this probability is equal to $P(\mathbf{u})$, so that

$$P(\mathbf{t}) = \prod_{l=1}^{L} P(u_l). \tag{3.6}$$

3. The denominator is the *normalizing constant*. The normalizing constant is the sum of $P(\mathbf{y}\,|\,\mathbf{t})P(\mathbf{t})$ for all possible bit sequences **t**.

$$P(\mathbf{y}) = \sum_{\mathbf{t}} P(\mathbf{y}\,|\,\mathbf{t})P(\mathbf{t}) \tag{3.7}$$

   The normalizing constant has a value such that the sum of $P(\mathbf{t}\,|\,\mathbf{y})$ for all possible

t becomes 1.

$$\begin{aligned}\sum_{\mathbf{t}} P(\mathbf{t}|\mathbf{y}) &= \sum_{\mathbf{t}} \frac{P(\mathbf{y}|\mathbf{t})P(\mathbf{t})}{P(\mathbf{y})} \\ &= \frac{\sum_{\mathbf{t}} P(\mathbf{y}|\mathbf{t})P(\mathbf{t})}{P(\mathbf{y})} \\ &= \frac{\sum_{\mathbf{t}} P(\mathbf{y}|\mathbf{t})P(\mathbf{t})}{\sum_{\mathbf{t}} P(\mathbf{y}|\mathbf{t})P(\mathbf{t})} \\ &= 1\end{aligned}$$

3.3 Decoding arithmetic codes using MAP decoding

We have already seen that MAP decoding is the identification of the message **u** with the highest probability $P(\mathbf{u}|\mathbf{y})$ given the received signal **y**. So the problem of decoding arithmetic codes using MAP decoding is a problem of searching for this best **u** from all the possible sequences **u**.

To search for the required **u**, we build a decoding tree. The tree will consist of a number of nodes (or states) and a number of edges connecting them. The state may be either the bit state or the symbol state. If we are using the bit state, each edge will represent one bit, and each node will have two child nodes, one corresponding to a `0`, and the other corresponding to a `1`. When traversing this tree, going from one state to the next (from one node to its child) happens every time we decode one bit.

If we are using the symbol state, the edges will represent symbols instead of bits, and the number of child nodes depends on the number of possible symbols. This time, going from one state to the next happens every time we decode one symbol.

For arithmetic codes, the size of the decoding tree increases exponentially with the number of symbols in the input sequence. So we have to use techniques to limit our search on some section of the tree; it is not feasible to compute $P(\mathbf{u}|\mathbf{y})$ for all possible sequences **u**.

Guionnet and Guillemot [10] present a scheme that uses synchronization markers in the arithmetic codes for error correction capabilities. Then they use estimation algorithms to decode the received signals. They use two kinds of markers, bit markers and symbol markers.

The idea in using bit markers is to insert a number of dummy bit patterns in the bit sequence after encoding some known number of symbols. Since the number of bits necessary to encode a number of symbols is not fixed, these bit patterns will occur at random places in the output bit sequence. The decoder then expects to find these bit patterns when decoding. If these patterns are not found, the path is pruned from the decoding tree. This helps limit the complexity.

Alternatively, symbol markers are used. This time, the idea is to insert a number of dummy symbols in the input sequence. The dummy symbols are introduces after encoding some known number of symbols. As for the bit markers, if the decoder does not find these dummy symbols when decoding, the path is pruned from the decoding tree.

Grangetto et al. [11] present another MAP estimation approach for error correction of arithmetic codes. Instead of bit markers or symbol markers, this approach uses the forbidden gap technique mentioned in § 2.6.2. The decoding tree uses the bit state, rather than the symbol state. Whenever an error is detected in a path of the decoding tree, that path is pruned. The number of bits N is sent as side information. If a path in the tree has N nodes but is not yet fully decoded, the detector prunes the path.

A comparison was made in [11] between this joint source-channel scheme and a separated scheme. In the separated scheme, an arithmetic code with $\varepsilon = 0$ is protected by a rate-compatible punctured convolutional (RCPC) code. The RCPC code used was of the family with memory $\nu = 6$ and non-punctured rate 1/3, proposed by Hagenauer [12]. The comparison indicated an improvement from the separated scheme.

In another paper, Grangetto et al. [13] present an iterative decoding technique that uses an adapted BCJR algorithm [14] for error correction of arithmetic codes.

In this dissertation, the ideas in [11] are implemented and some modifications are introduced. Recall that in MAP decoding, the problem is to find the transmitted sequence $\mathbf{t}$ which has the maximum probability $P(\mathbf{t}|\mathbf{y})$, and that this probability can be written as

$$P(\mathbf{t}|\mathbf{y}) = \frac{P(\mathbf{y}|\mathbf{t})P(\mathbf{t})}{P(\mathbf{y})}. \tag{3.2}$$

As we have already seen in § 3.2, the right-hand side of this equation has three parts, the likelihood $P(\mathbf{y}|\mathbf{t})$, the prior probability $P(\mathbf{t})$, and the normalizing constant $P(\mathbf{y})$.

The *a posteriori* probability $P(\mathbf{t}|\mathbf{y})$ is the decoding metric used, that is, the decoding algorithm tries to maximize this value. In the case of memoryless channels, we can use an additive metric m by taking logs of the decoding metric.

$$\begin{aligned} m &= \log P(\mathbf{t}|\mathbf{y}) \\ m &= \log P(\mathbf{y}|\mathbf{t}) + \log P(\mathbf{t}) - \log P(\mathbf{y}). \end{aligned} \tag{3.8}$$

The normalizing constant $P(\mathbf{y})$ is difficult to evaluate; (3.7) indicates that this requires knowledge of all possible bit sequences $\mathbf{t}$, which is not feasible. In [11], an approximation by Park and Miller [15] is used to go around this problem. We have N bits in

our bit sequence. If we assume that all 2^N bit sequences are possible, then

$$P(\mathbf{y}) \approx \prod_{n=1}^{N} \frac{P(y_n \mid t_n = 1) + P(y_n \mid t_n = 0)}{2}. \tag{3.9}$$

With this approximation, [11] claims that good results are obtained.

In § 3.3.1 and § 3.3.2 below, we will review the hard decoding and soft decoding metrics for MAP decoding presented by [11].

3.3.1 Hard decoding

Suppose we have an additive white Gaussian noise (AWGN) channel using binary phase-shift keying (BPSK) modulation with a signal to noise ratio E_b/N_0. For hard decoding the signal y_n can be either 0 or 1. The channel transition probability is

$$P(y_n \mid t_n) = \begin{cases} 1-p & \text{if } y_n = t_n \\ p & \text{if } y_n \neq t_n \end{cases} \tag{3.10}$$

where p is the probability that a bit is decoded in error, $p = \frac{1}{2}\operatorname{erfc}\sqrt{E_b/N_0}$.

We can split the additive decoding metric m into N parts,

$$m = \sum_{n=1}^{N} m_n \tag{3.11}$$

where m_n is the part of m for the nth bit. This is convenient as it enables us to update the metric m for each bit y_n we try to decode. That is, after each bit, we can update the metric m.

In the case of hard decoding, we can combine (3.8) and (3.11) to obtain

$$m_n = \log P(y_n \mid t_n) + \log P(t_n) - \log P(y_n). \tag{3.12}$$

Let us deal with the last component, $\log P(y_n)$. Using (3.9),

$$\log P(y_n) = \log[P(y_n \mid t_n = 1) + P(y_n \mid t_n = 0)] - \log 2.$$

Since $y_n = 1$ or $y_n = 0$, substituting (3.10) into this equation gives us

$$\begin{aligned} \log P(y_n) &= \log[1 - p + p] - \log 2 \\ &= -\log 2. \end{aligned}$$

So we can rewrite (3.12) as

$$m_n = \log P(y_n \mid t_n) + \log P(t_n) + \log 2. \tag{3.13}$$

Note that the second term on the right-hand side of (3.13), the prior probability, is not very straightforward to evaluate. We know that $P(\mathbf{t}) = P(\mathbf{u})$, because the transmitted bit sequence $\mathbf{t}$ has a one-to-one relationship with the input symbol sequence $\mathbf{u}$. When decoding a sequence $\mathbf{t}$, for each bit, the decoder will either decode no symbols, or it will decode one or more symbols. Suppose that after bit t_n, the decoder decodes the symbols $\mathbf{u}_n$. $\mathbf{u}_n$ is a vector containing I symbols $u_{n,1}, u_{n,2}, \cdots, u_{n,I}y$. If no symbols are decoded after bit t_n, $I = 0$ and $\mathbf{u}_n$ is an empty vector. In any case,

$$\log P(\mathbf{u}_n) = \sum_{i=1}^{I} \log P(u_{n,i}). \tag{3.14}$$

We can rewrite (3.13) as

$$m_n = \log P(y_n \mid t_n) + \log P(\mathbf{u}_n) + \log 2. \tag{3.15}$$

3.3.2 Soft decoding

For soft decoding, the metric can be found in a similar way. As in the case of hard decoding in § 3.3.1, suppose we have an AWGN channel using BPSK modulation with a signal to noise ratio E_b/N_0. The additive metric for soft decoding is

$$m_n = \log P(y_n \mid t_n) + \log P(t_n) - \log P(y_n).$$

For soft decoding, the signal y_n will not be constrained to only two values, 0 and 1. The first component of the metric is $\log P(y_n \mid t_n)$. Recall that the probability distribution for y_n is

$$P(y_n \mid t_n = 1) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y_n - x)^2}{2\sigma^2}\right) \tag{3.4}$$

$$P(y_n \mid t_n = 0) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y_n + x)^2}{2\sigma^2}\right). \tag{3.5}$$

For BPSK modulation and an AWGN channel, $x = \sqrt{E_b}$ and $\sigma = \sqrt{N_0/2}$.

The last component of the metric is $-\log P(y_n)$. As we have done for the hard decoding metric, we can approximate this component.

$$P(y_n) = \frac{P(y_n \mid t_n = 1) + P(y_n \mid t_n = 0)}{2}$$

It is more convenient to compute the value of $P(y_n \mid t_n)/P(y_n)$ directly instead of computing the values of $P(y_n \mid t_n)$ and $P(y_n)$ individually and dividing. It is trivial to rewrite (3.4) and (3.5) as

$$P(y_n \mid t_n = 1) = cr \tag{3.16}$$

$$P(y_n \mid t_n = 0) = \frac{c}{r} \tag{3.17}$$

where

$$c = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{y_n^2 + x^2}{2\sigma^2}\right) \quad \text{and}$$
$$r = \exp\left(\frac{2xy_n}{2\sigma^2}\right) = exp\left(2\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right).$$

Using these substitutions,

$$\begin{aligned}\frac{P(y_n \mid t_n = 1)}{P(y_n)} &= \frac{cr}{(cr + c/r)/2} \\ &= \frac{2r^2}{r^2+1} \\ &= \frac{2\exp\left(4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right)}{\exp\left(4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right)+1}\end{aligned}$$

$$\log\left[\frac{P(y_n \mid t_n = 1)}{P(y_n)}\right] = \log 2 + \left(4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right) - \log\left[\exp\left(4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right)+1\right]. \tag{3.18}$$

Similarly,

$$\begin{aligned}\frac{P(y_n \mid t_n = 0)}{P(y_n)} &= \frac{c/r}{(cr + c/r)/2} \\ &= \frac{2}{r^2+1} \\ &= \frac{2}{\exp\left(4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right)+1}\end{aligned}$$

$$\log\left[\frac{P(y_n \mid t_n = 0)}{P(y_n)}\right] = \log 2 - \log\left[\exp\left(4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right)+1\right]. \tag{3.19}$$

The metric m_n for the soft decoding case can thus be written as

$$m_n = \begin{cases} \log P(\mathbf{u}_n) + \log 2 + \left(4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right) - \log\left[\exp\left(4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right)+1\right] & \text{if } t_n = 1 \\ \log P(\mathbf{u}_n) + \log 2 - \log\left[\exp\left(4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right)+1\right] & \text{if } t_n = 0 \end{cases}. \tag{3.20}$$

3.4 Sequential decoding with the stack algorithm

Until now, we have seen the MAP decoding metrics used for hard decoding (3.15) and for soft decoding (3.20). Direct evaluation of the MAP metric over all possible bit sequences is not feasible. The size of the decoding tree would grow exponentially with the number of symbols L. To prevent this problem, sequential search techniques are used.

The decoder proposed in [11] uses a search algorithm along the branches of a binary tree. The sequential algorithm used is the stack algorithm [16]. The tree paths are kept in a list ordered by their metric; the path with the best metric is kept at the top of the list.

In each iteration, the best path is removed from the list and replaced by two paths; one assuming $t_n = 0$, and the other assuming $t_n = 1$. These two new paths then have their corresponding metrics updated, and are placed in their place in the ordered list.

The ordered list has a predefined maximum size M. When there are more then M paths, the paths with the worst metric are removed from the list.

Although the algorithm is called a stack algorithm, because the concept of a stack is useful for describing the algorithm, [16] suggests that storing the paths in a physical stack is not optimal, and that it is preferable to store the paths in random access storage. A physical stack would require a sequential comparison of the metric to insert a path, and relocation of large amounts of data in the required stack position. As an alternative method, [16] proposes splitting the stack into a number of buckets, each containing metrics that are close in value.

The following example will illustrate the stack algorithm.

Example 3.1. Let us assume that we have a decoding tree, where we go from one state to the next for every input bit. Every time we expand a node into two children, the metric for each child is calculated. Let us assume that the maximum size of the stack containing the nodes is $M = 8$.

Figure 3.2 shows a decoding tree as it evolves during the decoding process. Table 3.1 shows each step in the MAP decoding process.

In the beginning, we only have one node, node 0, with metric $m_0 = 0$. In Step 1, we expand this node into two child nodes, node 1 with metric $m_1 = 2.1$ and node 2 with metric $m_2 = 1.6$. Node 1 corresponds to the bit sequence `0` and node 2 corresponds to `1`. Now, the stack contains the nodes 1 (2.1) and 2 (1.6), in that order.

In Step 2, we choose the best node, node 1, and expand it into two child nodes, node 3, which corresponds to `00`, with metric $m_3 = 1.9$, and node 4, which corresponds to `01`, with metric $m_4 = 3.8$. Node 1 is removed from the stack and nodes 3

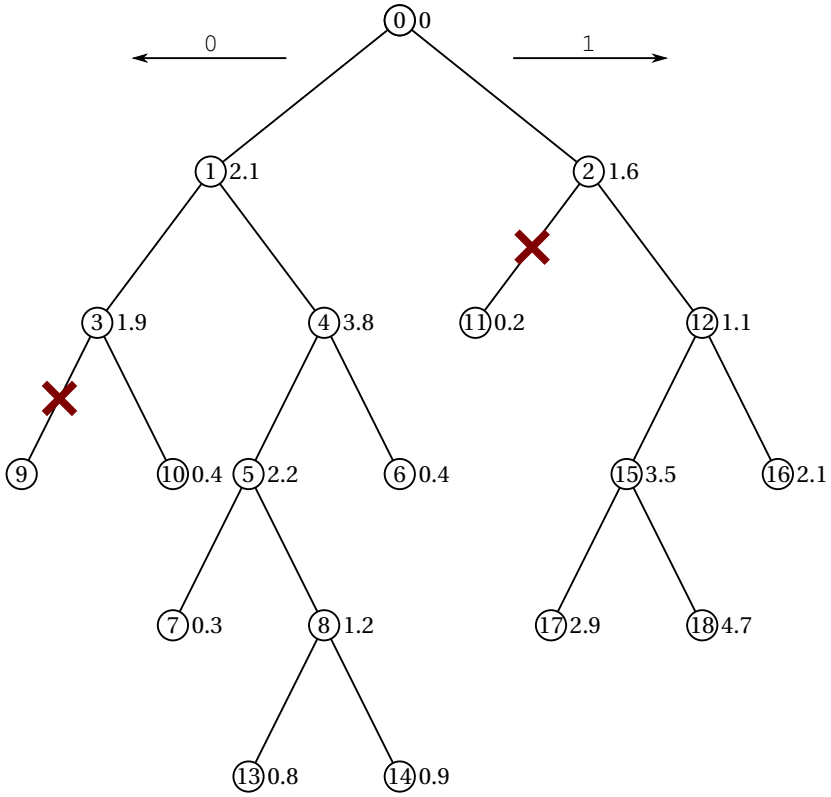

Figure 3.2: Binary decoding tree for Example 3.1.

Table 3.1: MAP decoding steps for Example 3.1.

Step	Action	Ordered List After Action
1	Expand node 0	1, 2
2	Expand node 1	4, 3, 2
3	Expand node 4	5, 3, 2, 6
4	Expand node 5	3, 2, 8, 6, 7
5	Expand node 3, node 9 has error	2, 8, 6, 10, 7
6	Expand node 2	8, 12, 6, 10, 7, 11
7	Expand node 8	12, 14, 13, 6, 10, 7, 11
8	Expand node 12	15, 16, 14, 13, 6, 10, 7, 11
9	Expand node 15, remove worst node	18, 15, 17, 16, 14, 13, 6, 10, 7

and 4 are inserted in their place so that the stack remains ordered. The stack contains the nodes 4 (3.8), 3 (1.9), and 2 (1.6), in that order.

Steps 3–4 are similar to the first two steps. Then, in Step 5, we expand node 3 into two nodes, node 9 and node 10. The bit sequence for node 9, `000`, gives an error when passed to the decoder, because the input interval falls within a forbidden gap. So node 9 is pruned from the tree. Node 10, `001` has a metric $m_{10} = 0.4$, and it is inserted in its place on the stack. Note that Step 5 does not increase the number of nodes in the tree; before Step 5 there were five nodes and after Step 5 there are five nodes.

Steps 6–8 continue to expand the tree until it contains eight nodes. Recall that the maximum size of the stack is $M = 8$. In Step 9, node 15, `110`, is expanded into two child nodes: node 17, `1100`, with metric $m_{17} = 2.9$, and node 18, `1101`, with metric $m_{18} = 4.7$. None of these are pruned, as the decoder detects no errors. If we keep all the nodes, the size of the stack will exceed the maximum size, so the worst node in the stack, node 11, with corresponding bit sequence `10`, is pruned.

In Example 3.1 above, the path starting with the two bits `10` was pruned. Note that we cannot be certain that the sequence $\mathbf{t}$ which will have the highest $P(\mathbf{t}|\mathbf{y})$ does not start with these two bits. Since we have to remove some nodes from the stack, we run the risk of losing the bit sequence we are searching for; and node 11 in the example above might have been in the path of the best bit sequence.

The maximum number of nodes in the stack, M, determines the complexity of the decoder. This parameter can be scaled according to processing and memory constraints. If the decoder has a large memory and a powerful processor, a large M can be chosen. If the processing power or memory are limited, a smaller M is chosen. Reducing M will reduce the complexity at the cost of error-correction performance. This is a trade-off between performance and complexity. Another option to reduce complexity is to increase the forbidden gap factor ε. If ε is large, errors are detected earlier and error paths may be pruned from the tree. Thus, M can be smaller and achieve the same PER if ε is increased.

Chapter 4

Improved Error Control for Arithmetic Codes

In the previous chapters, we have reviewed arithmetic coding, integrated error detection in arithmetic coding, and error correction techniques using MAP decoding. The MAP decoding scheme presented in [11] was implemented, and some modifications were then introduced and tested.

To enable comparison, the data encoded in this dissertation is of the same form as that used in [11]. In [11], a 256×256 image is represented as a sequence of 256 9-bit codewords, which is equal to 2304 bits. The probability of a `0` is 0.8666 and the probability of a `1` is 0.1334. This packet can be considered as a packet of 2304 symbols, with the source alphabet consisting of two symbols, one with a probability of 0.8666 and the other with a probability of 0.1334.

4.1 Placing the forbidden gap

In § 2.6, we have seen that a forbidden gap can be used to detect errors in arithmetic codes. In [8], the gap is placed at the end of the interval, and in Example 2.7, the gap was placed at the end of the interval too. Placing the forbidden gap at a different location may reduce the error-detection delay.

To investigate this possibility, an arithmetic encoder was used on a binary source with symbols a and b, with probabilities $P(a)$ and $P(b)$ respectively. The implementation details of the experiment will be discussed further on in § 5.3.

The results of the experiment indicated that if $P(a) > P(b)$, the forbidden gap should be placed before the symbols as shown in Figure 4.1 part (a). If $P(a)$ and $P(b)$ are modified such that $P(a) < P(b)$, the forbidden gap should be placed after the symbols as shown in Figure 4.1 part (b).

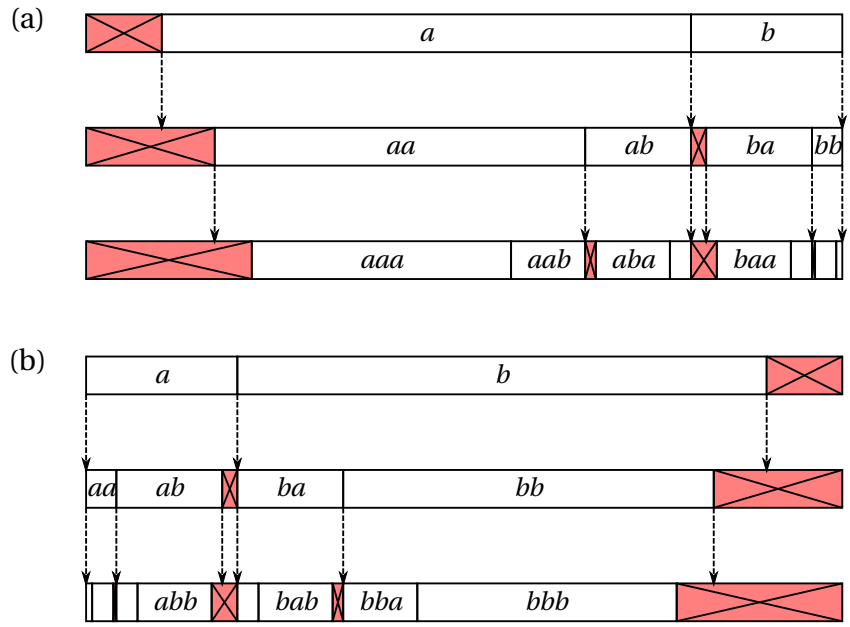

Figure 4.1: Forbidden gaps after three stages when placed to reduce error-detection delay for (a) $P(a) > P(b)$ and (b) $P(a) < P(b)$.

If the probabilities of the two symbols are equal, this advantage is lost. If $P(a) = P(b)$, no advantage can be seen in placing the forbidden gap as shown.

Suppose the model is adaptive, and that sometimes $P(a) > P(b)$ and sometimes $P(a) < P(b)$. This placement scheme needs some modification to work with such a model.

When encoding a symbol, the first thing to do is to find the most probable symbol. The sub-interval for this symbol is then swapped with the first sub-interval. Then, the forbidden gap can be placed at the beginning of the interval. This ensures that the sub-interval for the most probable symbol will be at the beginning of the interval. The example below will illustrate this technique.

Example 4.1. Suppose we have to find the interval for the input sequence *bacbba* of Example 2.7, with the same gap factor $\varepsilon = 0.1$. This time, we want to place the forbidden gap as described in this section.

Since the probability of the second symbol, b, is the largest, our first step during encoding is to swap the location of symbols a and b. We also place the forbidden gap at the beginning of the interval. The modified interval ranges can be seen in Table 4.1.

Again, we start with the interval [0, 1). Encoding the first symbol, b, we get the interval [0.1, 0.55). Encoding the second symbol, a, we get the interval [0.3475, 0.469).

Table 4.2 shows the six steps. Figure 4.2 shows a graphical representation of the process. The final interval is [0.4527, 0.4539). The width of the interval is 0.0012, which is exactly the same as the width of the final interval in Example 2.7. This is

Table 4.1: Symbol interval ranges for Example 4.1.

Symbol	Probability	Range	Scaled Probability	Scaled Range
Forbidden	0		0.1	[0, 0.1)
b	0.5	[0, 0.5)	0.45	[0.1, 0.55)
a	0.3	[0.5, 0.8)	0.27	[0.55, 0.82)
c	0.2	[0.8, 1)	0.18	[0.82, 1)

Table 4.2: Steps for finding the interval for Example 4.1.

Step	Symbol	Interval Before	Interval After
1	b	[0, 1)	[0.1, 0.55)
2	a	[0.1, 0.55)	[0.3475, 0.469)
3	c	[0.3475, 0.469)	[0.4471, 0.469)
4	b	[0.4471, 0.469)	[0.4493, 0.4592)
5	b	[0.4493, 0.4592)	[0.4503, 0.4547)
6	a	[0.4503, 0.4547)	[0.4527, 0.4539)

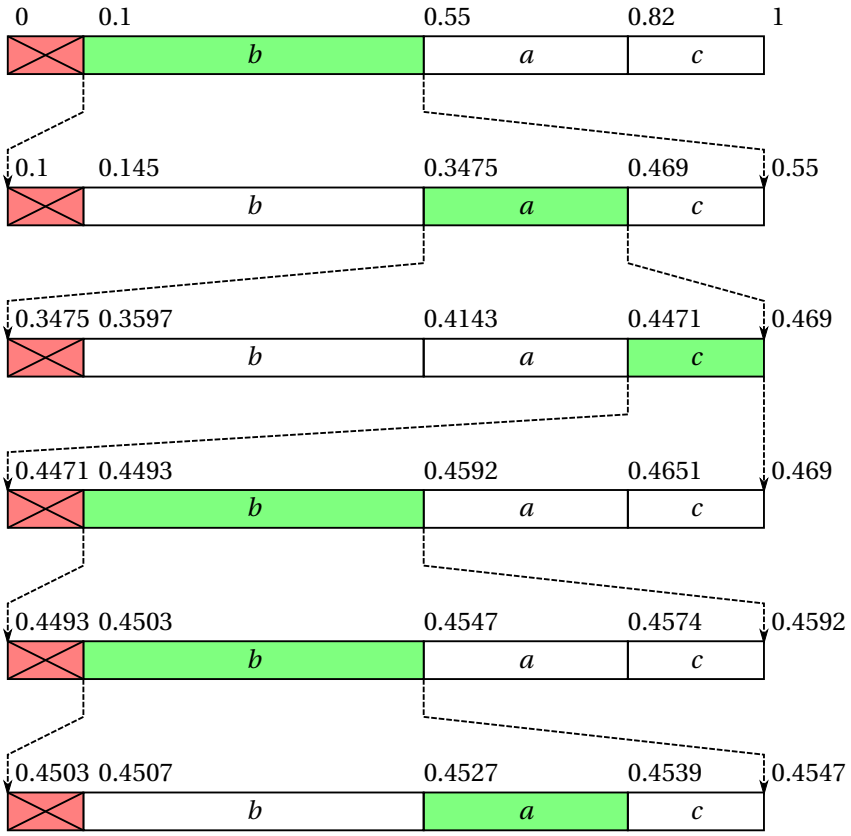

Figure 4.2: Finding the interval for Example 4.1.

because we only shifted the symbols around. So any gain from this technique has no space penalty; the only penalty is a slight increase in complexity.

4.2 Looking ahead during decoding

In § 2.5, we have seen that during decoding, we keep track of two different intervals. One interval is the decoding interval, which changes when a symbol is decoded; the other interval is the input interval, which depends on the bit sequence being decoded. The decoding interval is split into a number of sub-intervals, one for each possible symbol.

For a symbol to be decoded, the input interval has to lie within one of the sub-intervals of the decoding interval. When we introduced the forbidden gap, we began to detect errors when the input interval lies completely within the forbidden gap.

When decoding arithmetic codes with a forbidden gap, sometimes we can decode a symbol when the input interval is not completely within the sub-interval corresponding to the symbol. If the input interval is divided between one sub-interval and the forbidden gap, there is only one symbol that can be decoded, the symbol corresponding to that one sub-interval.

In this case, we can decode the symbol immediately and for the moment assume that the bit sequence is not in error. We still need to keep track of the input interval.

When we look ahead in this way, we may be able to detect errors earlier as will be illustrated in Example 4.2. Table 4.3 shows how the look-ahead decoder handles an input bit.

Example 4.2. Let us assume that we have a source alphabet with two symbols a and b having probabilities $P(a) = 0.6$ and $P(b) = 0.4$, and that we are using the forbidden gap technique with $\varepsilon = 0.2$. We need to start decoding the bit sequence `0011`..., first without using look-ahead, then using look-ahead. The interval ranges can be seen in Table 4.4.

Let us begin by not using look-ahead. We start with the input interval [0, 1) and the decoding interval [0, 1). We split the decoding interval using Table 4.4 The initial decoding interval is split into the forbidden sub-interval [0, 0.2), the sub-interval for a [0.2, 0.68), and the sub-interval for b [0.68, 1).

The first bit is a `0`, so our input interval becomes [0, 0.5), which does not lie within any one sub-interval. With the second bit, a `0`, our input interval becomes [0, 0.25), which does not lie within any sub-interval. With the third bit, a `1`, our input interval becomes [0.125, 0.25), which again does not lie within any sub-interval. With the

Table 4.3: Look-ahead decoding steps for each bit.

1: Initially, the input interval is $[l_i, h_i)$,
the decoding interval is split into S sub-intervals $[l_1, h_1)$, $[l_2, h_2)$, $\cdots$, $[l_S, h_S)$, and t_n is the input bit.
2: **if** $t_n = 0$ **then**
3: $h_i \Leftarrow (l_i + h_i)/2$
4: **else**
5: $l_i \Leftarrow (l_i + h_i)/2$
6: **end if**
7: Search for symbols s with their corresponding sub-interval $[l_s, h_s)$ overlapping the input interval $[l_i, h_i)$, that is, $l_s < h_i$ and $h_s > l_i$.
8: **if** no matching symbols are found **then**
9: Flag an error.
10: **else if** only one matching symbol is found **then**
11: Add found symbol s to the list of decoded symbols.
12: Scale the region $[l_s, h_s)$ if necessary, scaling $[l_i, h_i)$ in the same way.
13: Split the scaled $[l_s, h_s)$ into S new sub-intervals $[l'_1, h'_1)$, $[l'_2, h'_2)$, $\cdots$, $[l'_S, h'_S)$.
14: $l_1 \Leftarrow l'_1$, $h_1 \Leftarrow h'_1$, $l_2 \Leftarrow l'_2$, $h_2 \Leftarrow h'_2$, $\cdots$, $l_S \Leftarrow l'_S$, $h_S \Leftarrow h'_S$
15: Go to 7.
16: **else** {more than one matching symbol is found}
17: Go to 19.
18: **end if**
19: End.

Table 4.4: Symbol interval ranges for Example 4.2.

Symbol	Probability	Range	Scaled Probability	Scaled Range
Forbidden	0		0.2	[0, 0.2)
a	0.6	[0, 0.6)	0.48	[0.2, 0.68)
b	0.4	[0.6, 1)	0.32	[0.68, 1)

fourth bit, a `1`, our input interval becomes [0.1875, 0.25), which still does not lie within any one sub-interval. Thus, we are not yet able to do anything with the first four bits.

Let us now use look-ahead. We start with the same input interval [0, 1) and decoding interval [0, 1). The initial decoding interval is split into the forbidden sub-interval [0, 0.2), the sub-interval for a [0.2, 0.68), and the sub-interval for b [0.68, 1).

The first bit is `0`, so our input interval becomes [0, 0.5), which does not lie within any one sub-interval. However, it is split between the forbidden gap and only one symbol sub-interval, a; the input interval has no overlap with the sub-interval for symbol b. Thus, we can decode symbol a. When we decode symbol a, the decoding interval becomes [0.2, 0.68), which is split into the forbidden sub-interval [0.2, 0.296), the sub-interval for a [0.296, 0.5264), and the sub-interval for b [0.5264, 0.68).

The input interval is still [0, 0.5), which again is split between the forbidden gap and only one symbol, a. We decode another a, and the decoding interval becomes [0.296, 0.5264). Since $1/4 \leq 0.296 < 1/2 < 0.5264 \leq 3/4$, we can expand using the transformation $x \rightarrow 2x - 0.5$. The input interval becomes $[-0.5, 0.5)$. Note that the lower endpoint of the input interval is negative; we shall deal with this below. The decoding interval becomes [0.092, 0.5528) which is again split into the forbidden sub-interval [0.092, 0.18416), the sub-interval for a [0.18416, 0.405344), and the sub-interval for b [0.405344, 0.5528). Now the input interval has parts in both the sub-intervals for a and b, so we cannot decode further.

The second bit is `0`, so the input interval becomes $[-0.5, 0)$. Now the input interval does not have any overlap with either of the two source symbols a and b, so an error is detected.

Using no look-ahead, four bits did not decode a single symbol; using look-ahead, only two bits were necessary to detect an error.

When using look-ahead, the input interval does not need to lie completely within the decoding interval, so care needs to be taken when we expand the intervals. Since the input interval can be larger than the decoding interval, it is possible for the input interval to extend out of the [0, 1) interval. This is what happened in the example above. So in the implementation, we must make sure to cater for the possibility that $low_{input} < 0$ or that $high_{input} > 1$. This will be seen later on in § 5.2.

4.3 Updating the prior probability continuously

In § 3.3.1, we have seen how we can calculate the prior probability $P(\mathbf{u})$. Suppose that after bit t_n, the decoder decodes the symbols $\mathbf{u}_n$, where $\mathbf{u}_n$ is a vector containing I

symbols, $I \geq 0$. Recall that after each decoded bit t_n, the additive MAP metric (3.15) or (3.20) can be increased by

$$\log P(\mathbf{u}_n) = \sum_{i=1}^{I} \log P(u_{n,i}) \tag{3.14}$$

to deal with the prior probability.

There is another way to calculate the prior probability. In arithmetic coding, the width of the interval is directly related to the probability of the source symbols. For the moment let us ignore the forbidden gaps; we can say that if we encode an input sequence $u_1, u_2, \cdots, u_L$,

$$\text{Interval width} = P(\mathbf{u}) = \prod_{l=1}^{L} P(u_l).$$

Also, every time the encoder outputs a bit, the interval width is being halved. If we ignore termination, we can say that

$$\text{Interval width} = 2^{-N}$$

where N is the number of bits required to encode the source sequence.

Thus,

$$P(\mathbf{u}) = 2^{-N}.$$

Taking logs, this equation becomes

$$\log P(\mathbf{u}) = -N \log 2.$$

So all we have to do to the additive MAP metric to cater for the prior probability is to subtract $\log 2$ for each decoded bit.

We have ignored the effect of forbidden gaps on the interval width. Compensating for forbidden gaps is not very difficult. Every time there is a forbidden gap, that is, for each symbol encoded or decoded, the interval is reduced by a factor of $(1-\varepsilon)$. To compensate for this, for each decoded symbol we subtract $\log(1-\varepsilon)$ from the metric. Note that $(1-\varepsilon) < 1$, so we are subtracting a negative number, and the metric is increasing, not decreasing.

We have also ignored the effect of termination in our calculation. Compensating for termination is not difficult either. After we decode the last symbol, u_L, the decoder is keeping track of two intervals; the input interval specified by the input bit sequence, and the decoding interval.

If the input interval and the decoding interval have the same width, we do not

need to compensate for termination. However, usually the input interval is narrower than the decoding interval,

$$\text{Input interval} = f \cdot (\text{Decoding interval}), \qquad f < 1.$$

This would mean that our 2^{-N} is too small, and in effect, our $\log P(\mathbf{u})$ is too small. To compensate for this we subtract $\log f$ from the metric. Remember that $f < 1$, so we are subtracting a negative number and the metric is increasing.

After we decode the last symbol u_L and compensate for termination, the MAP metric includes the prior probability of the whole source sequence, $P(\mathbf{u})$. After this, any changes to the metric will have nothing to do with the prior probability, and will only be related to the likelihood and the normalization factor.

4.4 Using a different decoding tree

In § 3.3, we mentioned that we may use two kinds of decoding tree; one in which each edge represents a bit and the other in which each edge represents a symbol. The tree used in [11] is the first kind, where each edge represents a bit.

An attempt to use the other kind of tree was made, where each edge represents a symbol rather than a bit. Using this tree, there would be no need to prune incorrectly decoded bit sequences, as these would not appear in the tree. This experiment did not give satisfactory results; using this scheme, more packet errors occurred than when using a tree with edges representing bits. So this modification was discarded.

The problem may be this scheme takes longer to account for the signal received, $\mathbf{y}$. When the edges represent bits, going from a node to its child always has a corresponding bit, thus we can update the likelihood component of the metric, $P(\mathbf{y}|\mathbf{t})$, immediately. When the edges represent symbols, going from a node to its child will take longer to handle this; the process of going down a path in this case is similar to the arithmetic encoding process. An arithmetic encoder may output bits with some delay, and this process will update the metric with some delay. So the effect of the likelihood $P(\mathbf{y}|\mathbf{t})$ on the decoding metric is delayed.

Chapter 5

Implementation

The arithmetic encoder and decoder and the simulation were developed in C++ [17]. The GCC compiler version 4.3.0, `http://gcc.gnu.org/gcc-4.3/`, was used on the Fedora 9 GNU/Linux operating system and an x86-32 architecture.

In the implementation, more emphasis was given to generality than to optimization. This is because the aim was more to analyse different configurations and methods than to optimize a particular setup. When an implementation for a particular setup is required, some functions can be optimized and some features can be removed, leading to faster encoding and decoding.

To help ensure correctness, assertions were used throughout the source code. Also, the Valgrind memory leak detector, available at `http://valgrind.org/`, was used to ensure the program does not leak memory. These techniques are very useful in ensuring the quality of the developed software.

5.1 Fixed-point arithmetic

The arithmetic encoder converts a sequence of symbols into a sequence of bits. To do this, the encoder needs to handle fractions. The arithmetic decoder later converts the sequence of bits back to a sequence of symbols. The decoder needs to handle fractions as well. It is important that the fractions used by the encoder and the decoder match exactly.

Floating-point numbers were avoided in the implementation of the arithmetic encoder and decoder. This is because floating-point numbers may be represented differently on different architectures. A small change in representation can cause loss of synchronization between the encoder and the decoder, which might be operating on different machines.

Fixed-point numbers are different, because they are essentially represented as integers. A C++ fixed-point class was to be used. After a short search, no available

classes with the required flexibility were found, so a simple fixed-point class was developed. The developed class uses an integer for internal representation, and can have any number of bits used for the integer part, and the remaining bits used for the fraction part.

In arithmetic coding, the fractional numbers that are stored are probabilities, and thus they cannot be less than 0 or greater than 1. The integer used was a 32-bit unsigned integer. It was not desirable to have all 32 bits set aside for the fraction; consider the case when we find the average of two fixed-point numbers.

```
x = (l + h) / 2;
```

The sum of the two numbers can exceed 1, so at least one bit should be left for the integer part. So the fixed-point type used was a 32-bit unsigned integer split into one integer bit and 31 fraction bits. This type can hold the numbers $[0, 2)$.

Care has to be taken when transforming these fixed-point numbers using a transformation such as $x \to 2x - 1$. This can be done in two ways:

1. `x = 2 * x - 1;`

2. `x = (x - 0.5) * 2;`

The first form has a problem; if we start with $x = 1$, we need to store the intermediate value $x = 2$, which cannot be represented in our fixed-point representation. The second form does not have this problem, if $x = 1$ we first subtract 0.5 getting an intermediate value $x = 0.5$, which we then multiply by 2 to get the final value $x = 1$.

When multiplying fixed-point numbers, one method is to cast the two multiplicands to 64-bit integers, multiply them to get one 64-bit integer, right-shift this integer by 31 bits, and cast the answer back to a 32-bit integer. However this can take a lot of time considering that multiplication is a very common operation and is used extensively in arithmetic coding.

This problem can be solved easily for the x86-32 architecture, where multiplying two 32-bit integers will give a 64-bit answer inside the processor. The problem is that C++ does not have access to this 64-bit integer. So the multiplication of two fixed-point numbers was implemented in assembly as one 32-bit multiplication and one right-shift operation. Note that portability was maintained when developing this optimization; the assembly code is compiled on condition that the architecture is the x86-32 architecture. If the architecture is different, the compiler will use portable code which, although slower, will have exactly the same effect.

The fractional part has 31 bits, so the resolution is 2^{-31}, which is 4.7×10^{-10}. The width of the scaled encoding/decoding interval at any time is at least $1/4$; this is guaranteed by the expansion process described in § 2.2. A factor of $1/4$ is equivalent to a loss of two bits of resolution. Thus, the decoding interval has a minimum resolution

of 2^{-29}, which is 1.9×10^{-9}. Any sub-interval will have a maximum error of 1.9×10^{-9}. The probability of each symbol is usually much larger than this value, so the resolution of the fixed-point numbers used is adequate for arithmetic coding.

Note that the fixed-point numbers make the source code more readable than if integers are used, and impose no runtime performance penalty whatsoever. The internal representation of a fixed-point number is that of a 32-bit integer. Thus, the object code produced is identical to object code produced using integer arithmetic, only the syntax is clearer.

5.2 Arithmetic coding

Arithmetic coding can have two kinds of models, a static model and an adaptive model. The implementation in this work allows for both. An abstract base class `model` was written with the following abstract virtual member functions.

- `n_symbols() const` returns the number of possible symbols.
- `probability() const` returns the probability of a specified source symbol.
- `symbol()` informs the model that a particular symbol was encoded/decoded, so that the model changes its state.
- `fork() const` creates a copy of the model.

For a static model, the derived class does not need to have any state, so `symbol()` has no effect, and `fork() const` just creates a new instance of the class. For an adaptive model, the derived class needs to store some state information. In this case, the `symbol()` function will update the state. The `fork() const` function creates a new instance of the class having the same state.

The encoder and decoder support placing the gap at different locations in the interval.

The decoder supports decoding both with and without the look-ahead feature presented in § 4.2. Recall that when the look ahead feature is enabled, the input interval does not need to lie within the decoding interval. So for the input interval, there is a possibility that $low_{input} < 0$ or that $high_{input} > 1$. To cater for this, low_{input} and $high_{input}$ are split into an integer and a fixed-point number. Note that we still know that $low_{input} < 1$ and that $high_{input} > 0$.

The integer part of low_{input} holds $-\lfloor low_{input} \rfloor$, which will always be ≥ 0 because $low_{input} < 1$. The fraction part will hold $low_{input} - \lfloor low_{input} \rfloor$, which will be in the range $[0, 1)$.

The integer part of $high_{input}$ holds $\lceil high_{input} - 1 \rceil$, which will always be ≥ 0 because $high_{input} > 0$. The fraction part will hold $high_{input} - \lceil high_{input} - 1 \rceil$, which will be in the range $(0, 1]$.

5.3 Placing the forbidden gap

In § 4.1, we mentioned an experiment to find the optimal location of the forbidden gap in the interval. The experiment was performed on a source with a binary alphabet containing two symbols, a and b.

The forbidden gap was placed in different parts of the interval. It was placed at the beginning of the interval, in the middle of the interval, or at the end of the interval. A simulation was performed to compare the detection delay for an error using each of these schemes. For the simulation, the first symbol was assigned the probability $P(a)$ in the range $0.5 \leq P(a) < 1$ in steps of 0.025.

Note that because of symmetry, the performance of the code with $P(a) = p$ and with the forbidden gap placed at the beginning of the interval is equivalent to the code with $P(a) = 1 - p$ and with the forbidden gap placed at the end of the interval. Also, the performance of the code with $P(a) = p$ and with the forbidden gap placed in the middle of the interval, that is, between the sub-intervals for the two symbols, is equivalent to the code with $P(a) = 1 - p$ and with the forbidden gap placed in the middle of the interval.

To check the detection delay, a source message was generated with the required source probability. Then, it was encoded using the required scheme. A random bit from the first 100 bits was flipped, introducing an error. The error was constrained in the first 100 bits to avoid termination from influencing the result of this experiment. The bit sequence with the error was then passed into the decoder, and the number of bits decoded until the error was detected was recorded. This was repeated for 100,000 times for each configuration.

Until now we have seen how to analyse the placement of the forbidden gap on a source with a static model. To check the performance in a source with an adaptive model, the source model was modified so that for odd symbols, that is, for u_1, u_3, $\cdots$, $P(a) = p$ and $P(b) = 1 - p$, and for even symbols, that is, for u_2, u_4, $\cdots$, $P(a) = 1 - p$ and $P(b) = p$.

In this new setup, placing the forbidden gap at the beginning of the interval is equivalent to placing the forbidden gap at the end of the interval. That is because on alternating symbols, we get either $P(a) > P(b)$ or $P(a) < P(b)$. So a and b are not really different. If we use the scheme presented in § 4.1, we swap the symbol sub-intervals around so that the sub-interval for the most probable symbol, be it a or b, is at the

beginning of the interval; then we place the forbidden gap at the beginning of the interval. This is a different from just placing the forbidden gap at the beginning and leaving the sub-intervals untouched.

The adaptive model was used in three different setups; the interval was placed at the beginning of the interval, at the middle of the interval, and at the beginning with the sub-intervals swapped as mentioned above.

Each of these setups was tested both with and without look-ahead as presented in § 4.2. The results will be presented later in § 6.1. The smallest delay for the static model with $P(a) > P(b)$ was obtained by placing the interval at the beginning of the interval. The smallest delay for the adaptive model was obtained by placing the forbidden gap at the beginning, followed by the sub-interval for the most probable symbol. Hence, in the simulation of the MAP decoder, this scheme was used throughout. Look-ahead reduced the delay for each case except when placing the forbidden gap in the middle of the interval.

5.4 The MAP decoder metric

To implement the MAP decoder, the main task was to implement the decoding tree. The implementation was split into a class for the node and a class for the tree itself. The node class contains pointers to its parent and to its children, the decoding metric, and a pointer to an arithmetic decoder.

The MAP decoder class keeps an ordered list of the nodes. In this project, the ordered list was implemented using the `set` template class from the C++ standard library.

The metric for both hard and soft decoding is updated after each bit t_n by adding two components; the likelihood divided by the normalizing factor $P(y_n \mid t_n)/P(y_n)$ and the prior probability $P(\mathbf{u}_n)$.

At the beginning of the decoding process, we have the vector $\mathbf{y}$ which contains N values. Each bit t_n can be either `0` or `1`. Before we start searching the decoding tree, we create two vectors, one containing $P(\mathbf{y} \mid \mathbf{t} = 1)$ and the other containing $P(\mathbf{y} \mid \mathbf{t} = 0)$. This way, we do not have to calculate the same numbers for different paths.

A look at the calculation of this part of the metric for soft decoding is in order. Recall the two equations

$$\log\left[\frac{P(y_n \mid t_n = 1)}{P(y_n)}\right] = \log 2 + \left(4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right) - \log\left[\exp\left(4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right) + 1\right] \tag{3.18}$$

$$\log\left[\frac{P(y_n \mid t_n = 0)}{P(y_n)}\right] = \log 2 - \log\left[\exp\left(4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right) + 1\right]. \tag{3.19}$$

We can also write these in an equivalent alternative form

$$\log\left[\frac{P(y_n \mid t_n = 1)}{P(y_n)}\right] = \log 2 - \log\left[\exp\left(-4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right) + 1\right] \tag{5.1}$$

$$\log\left[\frac{P(y_n \mid t_n = 0)}{P(y_n)}\right] = \log 2 - \left(4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right) - \log\left[\exp\left(-4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}\right) + 1\right]. \tag{5.2}$$

To calculate these two values, we first calculate the value

$$r = 4\frac{E_b}{N_0}\frac{y_n}{\sqrt{E_b}}. \tag{5.3}$$

Then we rewrite (3.18) and (3.19) as

$$\log\left[\frac{P(y_n \mid t_n = 1)}{P(y_n)}\right] = \log 2 + r - \log(e^r + 1) \tag{5.4}$$

$$\log\left[\frac{P(y_n \mid t_n = 0)}{P(y_n)}\right] = \log 2 - \log(e^r + 1). \tag{5.5}$$

We also rewrite (5.1) and (5.2) as

$$\log\left[\frac{P(y_n \mid t_n = 1)}{P(y_n)}\right] = \log 2 - \log(e^{-r} + 1) \tag{5.6}$$

$$\log\left[\frac{P(y_n \mid t_n = 0)}{P(y_n)}\right] = \log 2 - r - \log(e^{-r} + 1). \tag{5.7}$$

Equations (5.4) and (5.6) are mathematically equivalent. However, during computation one may have an advantage over the other. If $r < 0$, e^{-r} may become very large, but e^r will not. So for $r \leq 0$, we use equations (5.4) and (5.5). Also, if $r < -40$, e^r will be very small, so the term $\log(e^r + 1)$ becomes zero. For $r > 0$, we use equations (5.6) and (5.7), and if $r > 40$, e^{-r} will be very small and the term $\log(e^{-r} + 1)$ becomes zero.

The second part of the metric, the prior probability $P(\mathbf{u}_n)$, is calculated as described in § 4.3.

For every iteration of the decoding process, the node with the highest metric is removed from the ordered list and replaced by two child nodes.

5.5 The simulation

The scheme presented in [11] was implemented. As already mentioned previously, the input symbols are grouped in packets of $256 \times 9 = 2304$ symbols. The source alphabet consists of two symbols, a with probability $P(a) = 0.8666$ and b with probability $P(b) = 0.1334$. Each packet is terminated using a termination symbol with $\omega = 10^{-5}$.

The entropy $H(U)$ of the code is

$$\begin{aligned} H(U) &= -0.8666 \log_2 0.8666 - 0.1334 \log_2 0.1334 \\ &= 0.5667. \end{aligned}$$

On average, 0.5667 bits will be needed to encode each source symbol. So on average, the length N of the encoded bit sequence is

$$\begin{aligned} N &= 2304 H - \log_2 \omega \\ &= 1322.27. \end{aligned}$$

In the implementation, the length N of each encoded packet and the probability $P(a)$ are passed as side information. The decoder can use the length N to prune paths that are fully decoded using less than N bits, or that are not fully decoded at bit N.

In practice, the packet length and the model probability would have to be passed over the channel as well. They would be protected against errors using a more powerful code, as errors in receiving the side information will compromise the whole packet. There is still some probability that the side information is corrupted. Synchronization schemes would then need to be set up to ensure that if the side information of one packet is corrupted and the packet is lost, decoding of the other packets can still be performed. These schemes are beyond the scope of this work.

In [11], the MAP decoding algorithm using the stack sequential search with a maximum of $M = 4096$ nodes is simulated for different values of ε. The source alphabet contains two symbols a and b with probabilities $P(a) = 0.8666$ and $P(b) = 0.1334$. The values of ε used are $\varepsilon = 0.05$, which corresponds to a code rate of $8/9$, $\varepsilon = 0.097$, which corresponds to a code rate of $4/5$, and $\varepsilon = 0.185$, which corresponds to a code rate of $2/3$. The results obtained can be used to plot the packet error ratio (PER) against the signal-to-noise ratio E_b/N_0.

The simulation for the above was done using both soft and hard decoding. Also, two values of the maximum size of the stack M were used, 256 and 4096. The value of M corresponds to a trade-off between complexity and performance as discussed in § 3.4. When $M = 256$, the complexity is reduced and the error-correcting process is faster, but the PER is higher.

In this project this simulation was repeated without look-ahead and with look-ahead, and with the original method of calculating the prior probability $P(\mathbf{u}_n)$, and with the continuous calculation of the prior probability. The results will be presented in § 6.2.

The results obtained were similar to those in [11], and the modifications improved

on the published results.

To check that the results are not limited to this model, a first-order Markov model with three source symbols was also simulated. The probability of each symbol depends on the last symbol. The model used has the following probabilities:

$$\begin{array}{lll} P(a\,|\,a) = 0.7 & P(b\,|\,a) = 0.2 & P(c\,|\,a) = 0.1 \\ P(a\,|\,b) = 0.1 & P(b\,|\,b) = 0.4 & P(c\,|\,a) = 0.5 \\ P(a\,|\,c) = 0.6 & P(b\,|\,c) = 0.2 & P(c\,|\,c) = 0.2 \end{array}$$

The probability of each symbol is $P(a) = 0.5278$, $P(b) = 0.25$, and $P(c) = 0.2222$. The entropy of U conditional on each previous symbol can be calculated to be $H(U\,|\,a) = 1.1568$, $H(U\,|\,b) = 1.3610$, and $H(U\,|\,c) = 1.3710$.

The entropy $H(U)$ of this code is

$$\begin{aligned} H(U) &= -0.5278 \times 1.1568 - 0.25 \times 1.3610 - 0.2222 \times 1.3710 \\ &= 1.2554. \end{aligned}$$

For a code rate of 8/9, we need a forbidden gap factor ε such that

$$\begin{aligned} \frac{H}{H - \log_2(1-\varepsilon)} &= \frac{8}{9} \\ \log_2(1-\varepsilon) &= -\frac{H}{8} \\ \varepsilon &= 0.12 \end{aligned}$$

So this code has the same code rate as the binary code with $P(a) = 0.8666$ and $P(b) = 0.1334$ protected by a forbidden gap with $\varepsilon = 0.05$. For the length of the bit packet to be equal, we need N to be 1322.

$$\begin{aligned} N &= LH - \log_2 \omega \\ L &= \frac{N + \log_2 \omega}{H} \\ L &= 889 \end{aligned}$$

This code was then tested without look-ahead and with look-ahead, and with the original method of calculating the prior probability $P(\mathbf{u}_n)$, and with the continuous calculation of the prior probability.

The results will be presented in § 6.2 and compared to the results for the other scheme.

5.5.1 Random numbers

For the simulation, a random number generator was written using ideas in [18], including a method to generate Gaussian random numbers. Let x and y be two independent random numbers distributed uniformly in $[-1, 1]$. Let $r = \sqrt{x^2 + y^2}$, and ensure that r lies in $(0, 1)$. If $r = 0$ or $r \geq 1$, get replacements for x and y and repeat. Then let $f = \sqrt{\frac{-2\log_e r}{r}}$. To get two independent Gaussian random numbers with mean 0 and standard deviation 1, we calculate $x \cdot f$ and $y \cdot f$. The Gaussian random numbers were used in getting a signal vector $\mathbf{y}$.

5.5.2 Confidence interval of results

For the simulation, the number of iterations for each point had to be decided. Using too few iterations gives inaccurate results. Using too many iterations will take too long.

Since the outcome of each iteration is either a correct packet or a bad packet, finding the confidence interval for the PER is a binomial proportion confidence interval problem.

The Wilson interval [19] is a good approximation of the confidence for such systems. The confidence is given by the formula

$$\frac{\hat{p} + \frac{z^2_{1-\alpha/2}}{2n} \pm z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n} + \frac{z^2_{1-\alpha/2}}{4n^2}}}{1 + \frac{z^2_{1-\alpha/2}}{n}}$$

where $\hat{p}$ is the PER estimated from the simulation, n is the number of iterations performed, and $z_{1-\alpha/2}$ is the $1-\alpha/2$ percentile of a standard normal distribution.

This interval extends by

$$\frac{z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n} + \frac{z^2_{1-\alpha/2}}{4n^2}}}{1 + \frac{z^2_{1-\alpha/2}}{n}}$$

on each side of the centre.

In the simulation, we want to be 80 percent sure that our PER is within ± 0.01 from the centre of the confidence interval. The confidence is 0.8, and $\alpha = 1 - \text{confidence} = 0.2$, so $1 - \alpha/2 = 0.9$. We find the percentile $z_{1-\alpha/2}$ using the function `ltqnorm()` (see Appendix A).

If we have m packet errors from n runs, and the PER is $\hat{p} = m/n$, the expression for the extension of the confidence interval on each side of the centre can be simplified

to

$$\frac{z_{1-\alpha/2}\sqrt{\frac{m(n-m)}{n}+\frac{z_{1-\alpha/2}^2}{4}}}{n+z_{1-\alpha/2}^2}. \tag{5.8}$$

During the simulation, we monitor this score, and when this score is less than 0.01, we are satisfied with the number of runs.

This is not enough when we need to draw a logarithmic plot. For example, if the PER is about 10^{-4}, an error of ± 0.01 is very bad. In this case, we would need the error to be proportional to the PER m/n, say, fm/n where f is a factor such as 0.1. Since n is much larger than m, and much larger than $z_{1-\alpha/2}$, we can simplify as follows:

$$\frac{fm}{n} = \frac{z_{1-\alpha/2}\sqrt{\frac{m(n-m)}{n}+\frac{z_{1-\alpha/2}^2}{4}}}{n+z_{1-\alpha/2}^2}$$

$$fm = z_{1-\alpha/2}\sqrt{m+\frac{z_{1-\alpha/2}^2}{4}}$$

Solving for m we can easily obtain

$$m = \frac{2z_{1-\alpha/2}^2}{f^2}\left(1+\sqrt{1+f^2}\right).$$

This results indicates that the error on a logarithmic plot can be controlled by ensuring a minimum number of errors. In the implementation, a minimum of 50 errors was required. The simulation goes on until there are 50 packet errors. However, to avoid very large number of iterations when the PER is very small, a maximum of 10^6 iterations was set.

For large PER values, the number of iterations is determined using (5.8); the simulation stops when the value of the expression reaches 0.01. For small PER values, the simulation stops when the number of errors reaches 50. For very small PER values, when less than 50 errors are encountered in 10^6 iterations, the simulation stops after 10^6 iterations. This enables us to obtain smooth curves for the PER without taking too long where it is not necessary.

Chapter 6

Results

6.1 Placing the forbidden gap

In § 5.3 an experiment to find the optimal location of the forbidden gap was presented. The test was performed for a binary source model with alphabet containing symbols a and b in the following scenarios:

1. Fixed model, forbidden gap placed at the beginning of the interval.
2. Fixed model, forbidden gap placed at the beginning of the interval, look-ahead.
3. Fixed model, forbidden gap placed in the middle of the interval.
4. Fixed model, forbidden gap placed in the middle of the interval, look-ahead.
5. Fixed model, forbidden gap placed at the end of the interval.
6. Fixed model, forbidden gap placed at the end of the interval, look-ahead.
7. Adaptive model, forbidden gap placed at the beginning of the interval.
8. Adaptive model, forbidden gap placed at the beginning of the interval, look-ahead.
9. Adaptive model, forbidden gap placed in the middle of the interval.
10. Adaptive model, forbidden gap placed in the middle of the interval, look-ahead
11. Adaptive model, forbidden gap placed in the beginning of the interval, with the sub-interval for the most probable symbol moved towards the beginning of the interval as described in § 4.1.

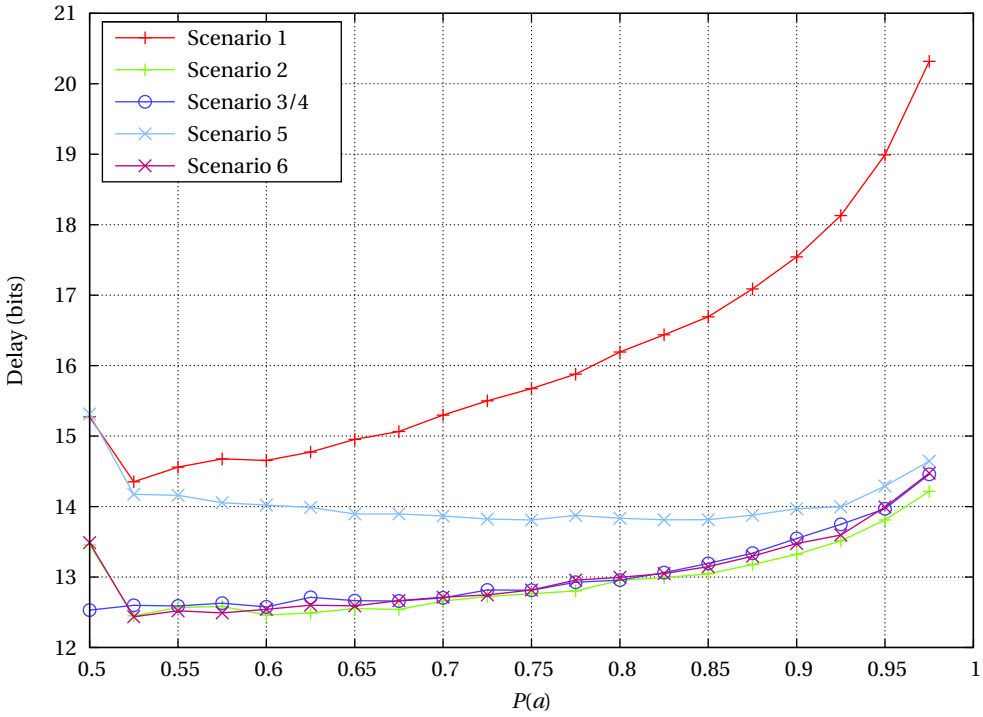

Figure 6.1: Error-detection delay for static source model with a code rate of 8/9.

12. Adaptive model, forbidden gap placed in the beginning of the interval, with the sub-interval for the most probable symbol moved towards the beginning of the interval as described in § 4.1, look-ahead.

The error-detection delay was measured for different values of $P(a)$, where $P(a)$ is the probability of the first symbol. Each test was performed using no look-ahead and using look-ahead.

Figure 6.1 shows the results for a static source model with a code rate of 8/9 (scenarios 1–6). Without look-ahead, placing the forbidden gap at the beginning of the interval suffers the largest delay, but with look-ahead, placing the forbidden gap at the beginning of the interval achieves the smallest delay.

When the probability of the first symbol in the interval is larger than the probability of the second symbol, and the forbidden gap is placed at the beginning of the interval, forbidden gaps tend to cluster together. Without look-ahead, this seems to make the delay larger. This may be because concentrating the gaps at fewer places makes it harder for a random interval to find one of the gaps. With look-ahead, the situation is reversed, because of the situation illustrated in Example 4.2.

Figure 6.2 shows the results for an adaptive source model with a code rate of 8/9 (scenarios 7–11). As discussed above, the performance of scenarios 11 and 12 is similar to the performance of scenarios 1 and 2. Without look-ahead, this scheme suffers the largest delay, and with look-ahead, it achieves the smallest delay.

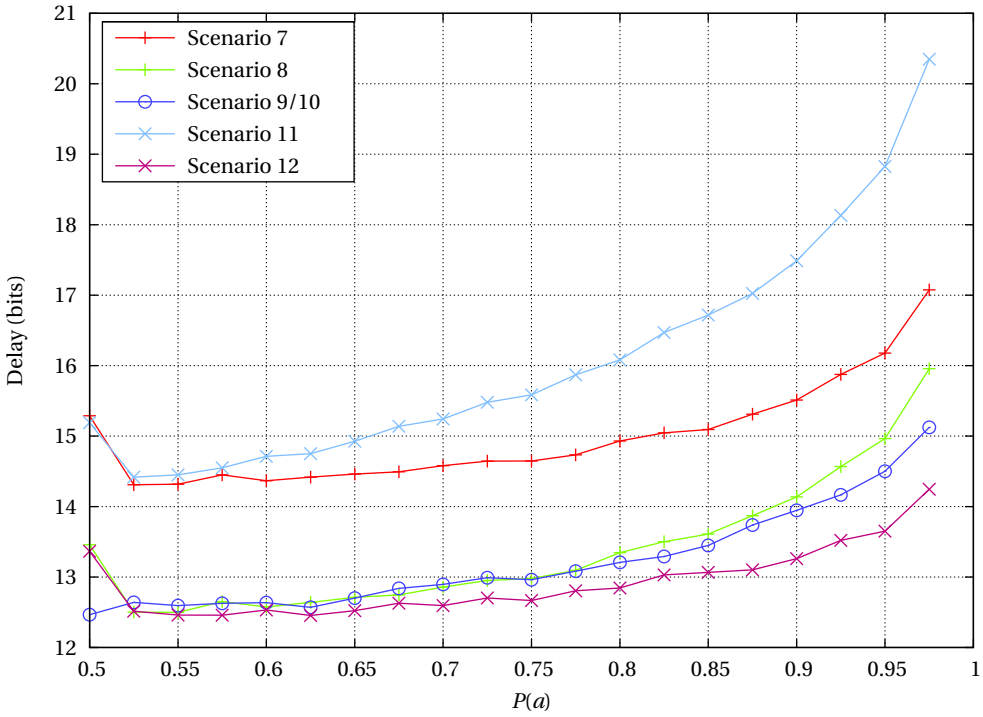

Figure 6.2: Error-detection delay for adaptive source model with a code rate of 8/9.

The results for scenarios 1, 2 and 11, 12 for $P(a) \geq 0.5$ were equivalent. This is because when $P(a) \geq 0.5$, they are identical schemes.

The results for scenarios 3 and 4 are identical. This means that look-ahead does not help when the forbidden gap is in the middle of the interval. This may be because with this scheme, forbidden gaps are never next to each other. In the graphs, only one of these scenarios is plotted. The same can be said for scenarios 9 and 10.

The simulation was performed for code rates of 4/5 and 2/3 as well. The results are similar to those for a code rate of 8/9. Figure 6.3 shows the results for a static source model with a code rate of 4/5 and Figure 6.4 shows the results for an adaptive source model with the same code rate.

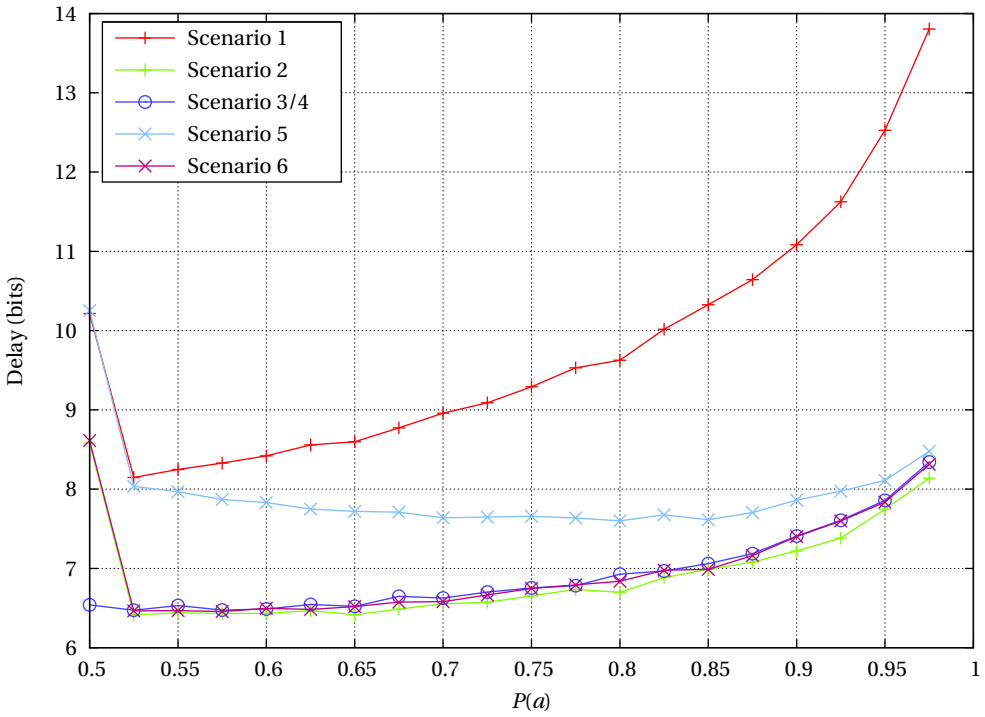

Figure 6.3: Error-detection delay for static source model with a code rate of 4/5.

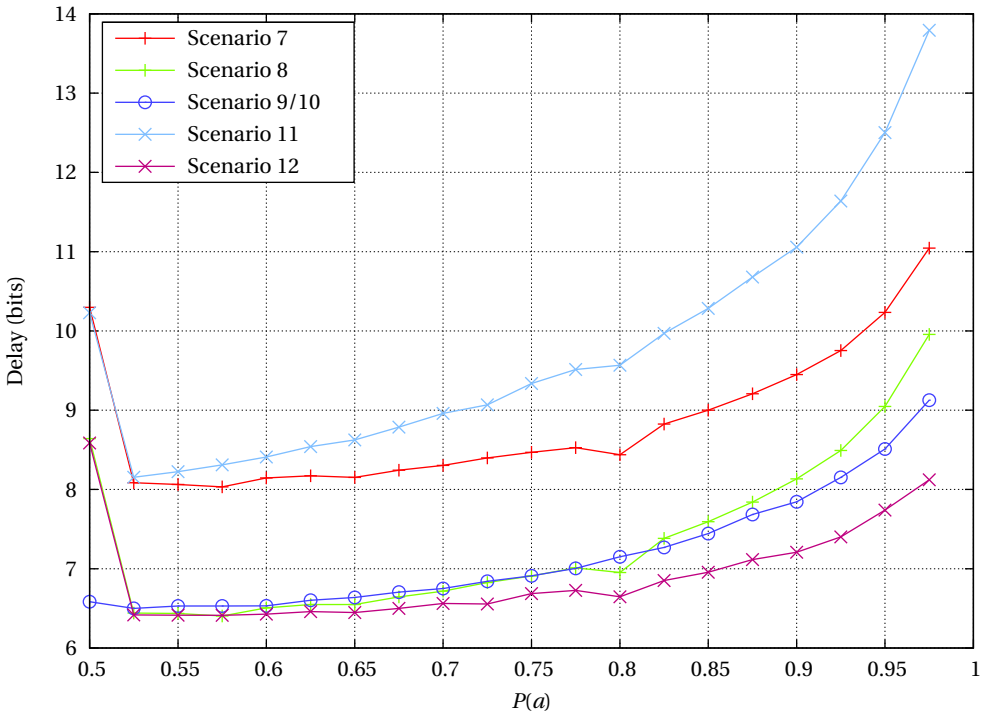

Figure 6.4: Error-detection delay for adaptive source model with a code rate of 4/5.

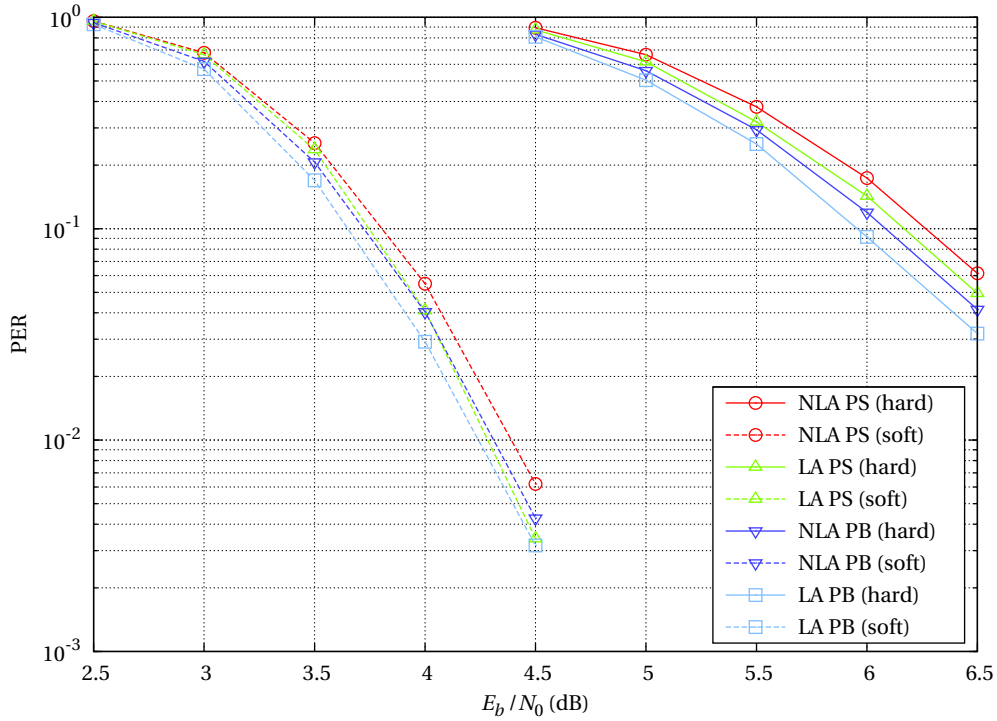

Figure 6.5: Performance for MAP decoder for static binary model with $M = 256$ and $\varepsilon = 0.05$; without look-ahead (NLA) and with look-ahead (LA); and with $P(\mathbf{t})$ adjusted every symbol (PS) or every bit (PB).

6.2 MAP decoding

The results for the simulation described in § 5.4 are presented in this section.

6.2.1 The static binary model

Figure 6.5 shows the PER for the scheme of the static binary model with $M = 256$ and $\varepsilon = 0.05$.

The performance is improved when the look-ahead technique is used. When this technique is used, the decoder may detect errors earlier, and it can detect correct symbols earlier as well. When errors are detected early, incorrect paths can be pruned earlier from the decoding tree, reducing the chance that the correct path is removed because of a stack overflow. Detecting symbols early will enable the MAP metric to be updated earlier, which can lead to better decoding. The performance is also improved with continuous updating of the prior probability $P(\mathbf{t})$, that is, when $P(\mathbf{t})$ is adjusted every time we decode a bit rather than every time we decode a symbol. The performance is improved most when the techniques are used together.

Figures 6.6 and 6.7 show the same results, but this time for $\varepsilon = 0.097$ and $\varepsilon = 0.185$ respectively. Notice that when ε is larger, the improvement becomes more noticeable.

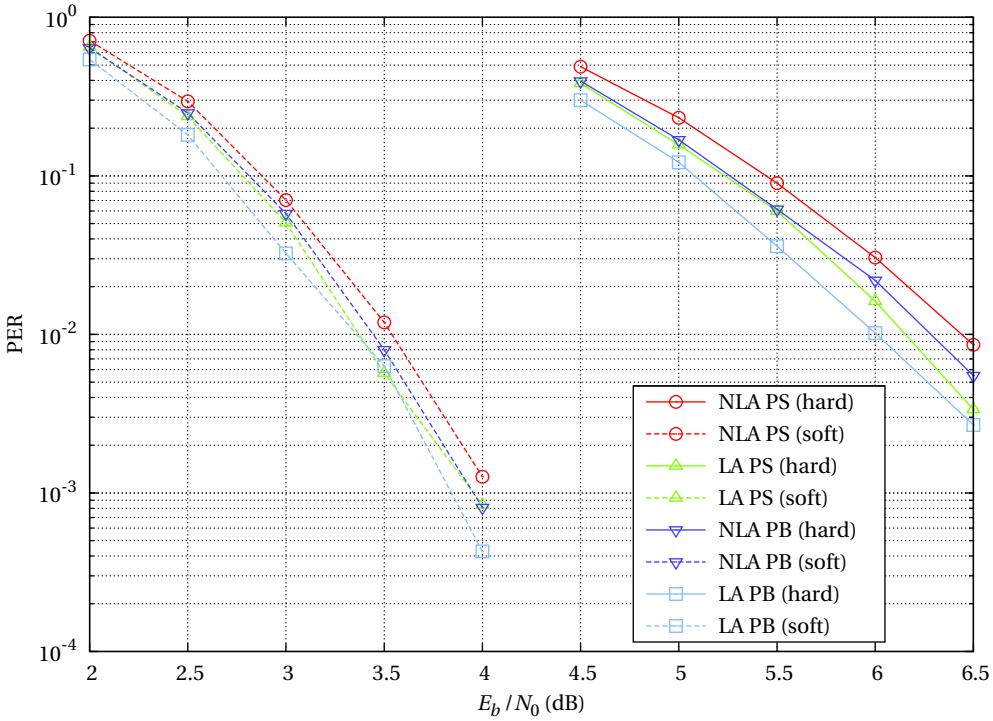

Figure 6.6: Performance for MAP decoder for static binary model with $M = 256$ and $\varepsilon = 0.097$; without look-ahead (NLA) and with look-ahead (LA); and with $P(\mathbf{t})$ adjusted every symbol (PS) or every bit (PB).

A small code rate, and hence a large value of ε, is usually used when the signal-to-noise ratio E_b/N_0 is low. The improvements are therefore particularly effective for poor channel conditions.

Finally, Figure 6.8 shows how the PER changes with ε when E_b/N_0 is fixed at 5.5 dB. The improved schemes can achieve the error-correction performance of the original scheme using less redundancy. For example, for a PER of 10^-2, the hard decoder for the original scheme needs $\varepsilon = 0.19$, which translates into a code rate of 0.65. For the same PER, the hard decoder for the improved scheme needs $\varepsilon = 0.13$, which translates into a code rate of 0.73.

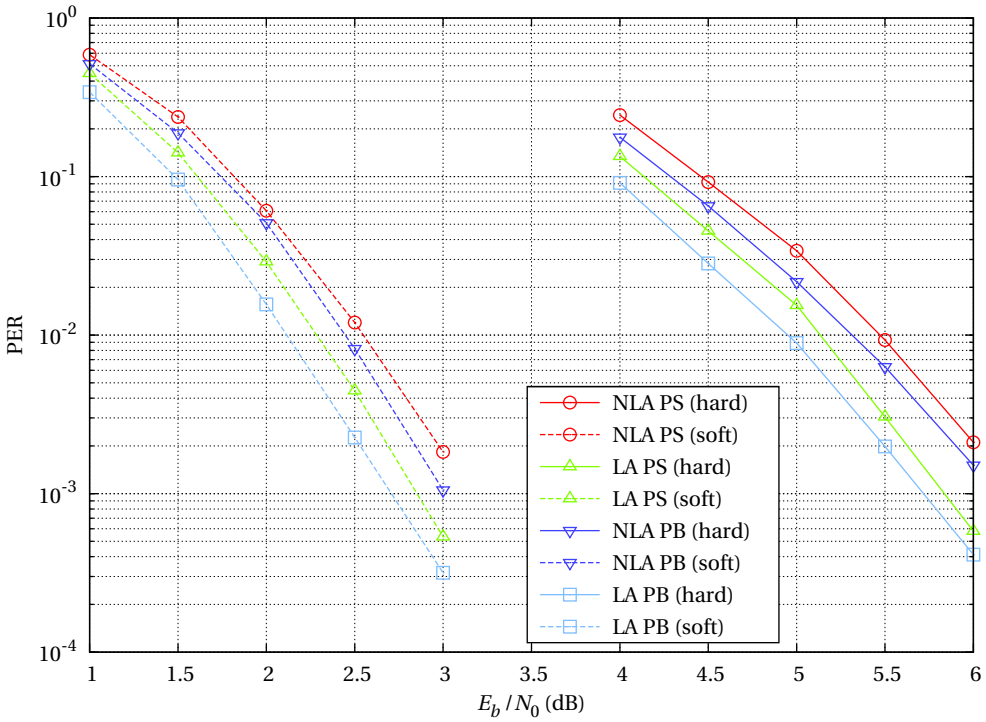

Figure 6.7: Performance for MAP decoder for static binary model with $M = 256$ and $\varepsilon = 0.185$; without look-ahead (NLA) and with look-ahead (LA); and with $P(\mathbf{t})$ adjusted every symbol (PS) or every bit (PB).

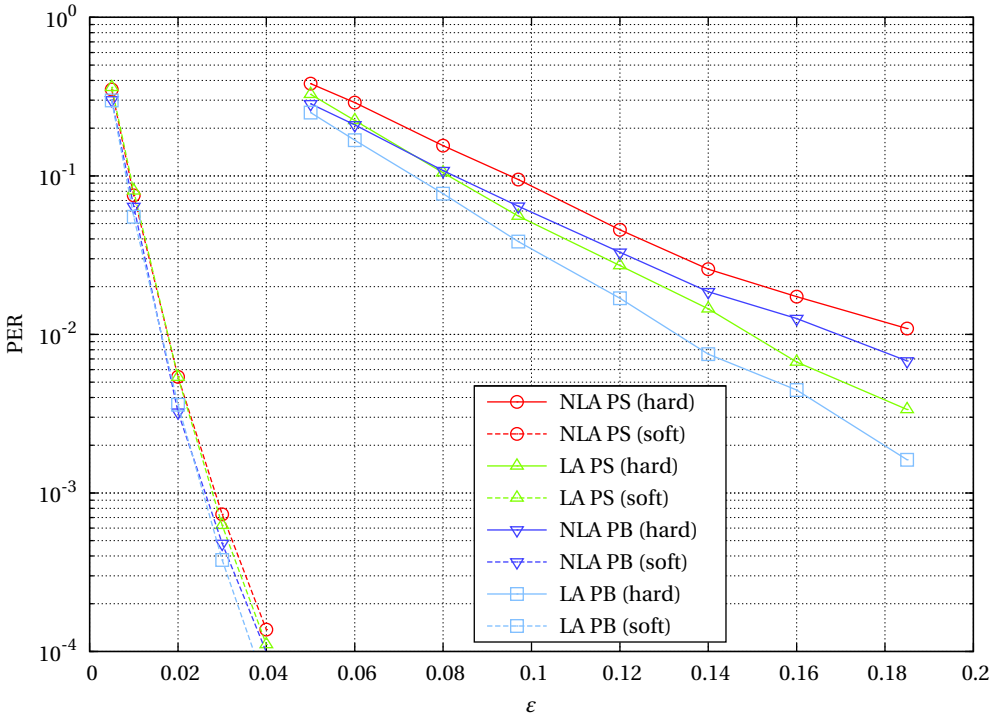

Figure 6.8: Performance for MAP decoder for static binary model with $M = 256$ and $E_b/N_0 = 5.5$ dB; without look-ahead (NLA) and with look-ahead (LA); and with $P(\mathbf{t})$ adjusted every symbol (PS) or every bit (PB).

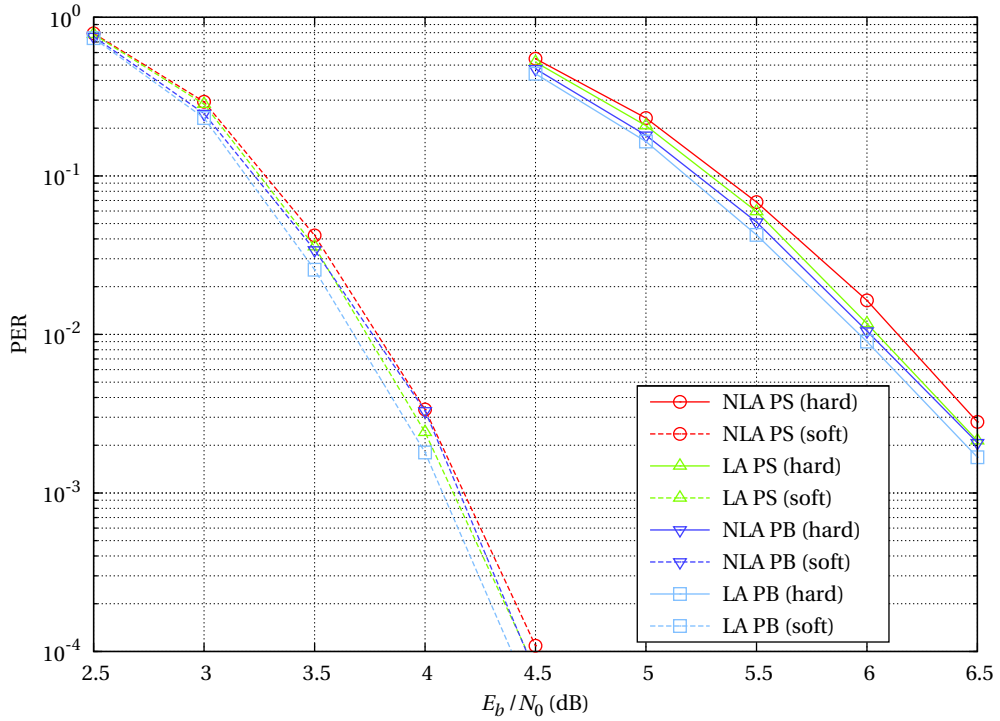

Figure 6.9: Performance for MAP decoder for static binary model with $M = 4096$ and $\varepsilon = 0.05$; without look-ahead (NLA) and with look-ahead (LA); and with $P(\mathbf{t})$ adjusted every symbol (PS) or every bit (PB).

Figures 6.9, 6.10, 6.11, and 6.12 are similar to the figures mentioned before, but with $M = 4096$. The complexity is higher, but the PER is lower. These curves can be compared to the results presented in [11]. In all four graphs, the plot for the scheme with no look-ahead and with $P(\mathbf{t})$ updated every symbol is comparable to the plots published in [11].

Notice the points at the bottom of the soft decoding plots in Figures 6.10 and 6.11; some plots seem to be misplaced. In § 5.5.2 we have seen that we require a minimum number of 50 errors to be reasonably confident of the location of a point in the logarithmic plot. However, the maximum number of iterations per point is 10^6. For a PER of around 10^{-5}, we get only about 10 errors, so we cannot be confident of the exact location of the points.

Once more, both look-ahead and adjusting the prior probability every time a bit is decoded improve on the original scheme, and combining both techniques gives the lowest PER.

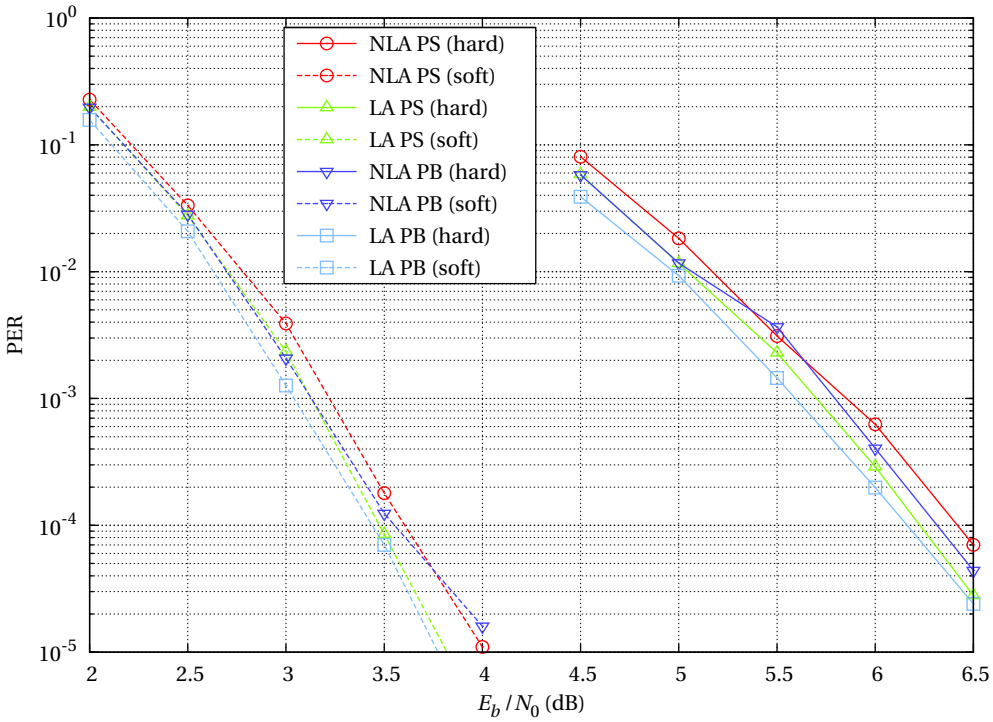

Figure 6.10: Performance for MAP decoder for static binary model with $M = 4096$ and $\varepsilon = 0.097$; without look-ahead (NLA) and with look-ahead (LA); and with $P(\mathbf{t})$ adjusted every symbol (PS) or every bit (PB).

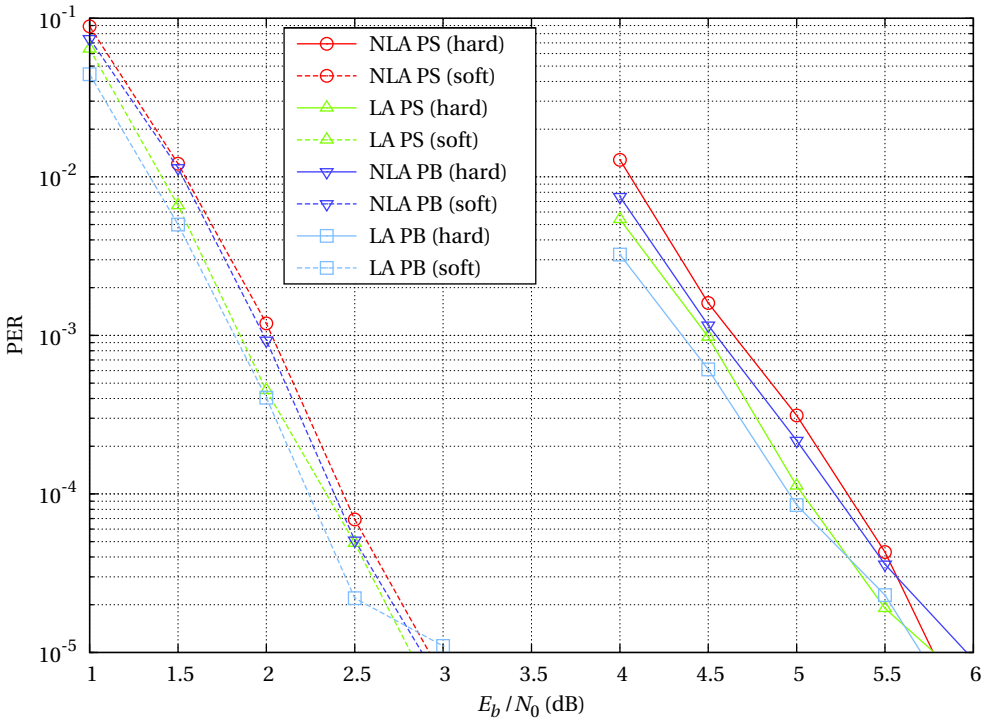

Figure 6.11: Performance for MAP decoder for static binary model with $M = 4096$ and $\varepsilon = 0.185$; without look-ahead (NLA) and with look-ahead (LA); and with $P(\mathbf{t})$ adjusted every symbol (PS) or every bit (PB).

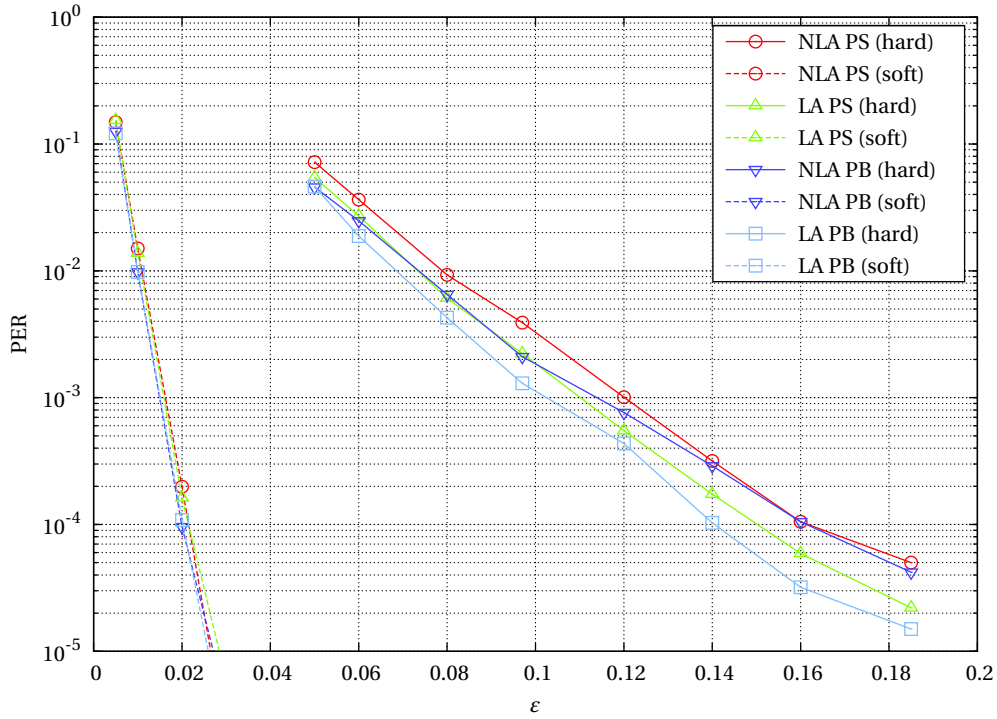

Figure 6.12: Performance for MAP decoder for static binary model with $M = 4096$ and $E_b/N_0 = 5.5$ dB; without look-ahead (NLA) and with look-ahead (LA); and with $P(\mathbf{t})$ adjusted every symbol (PS) or every bit (PB).

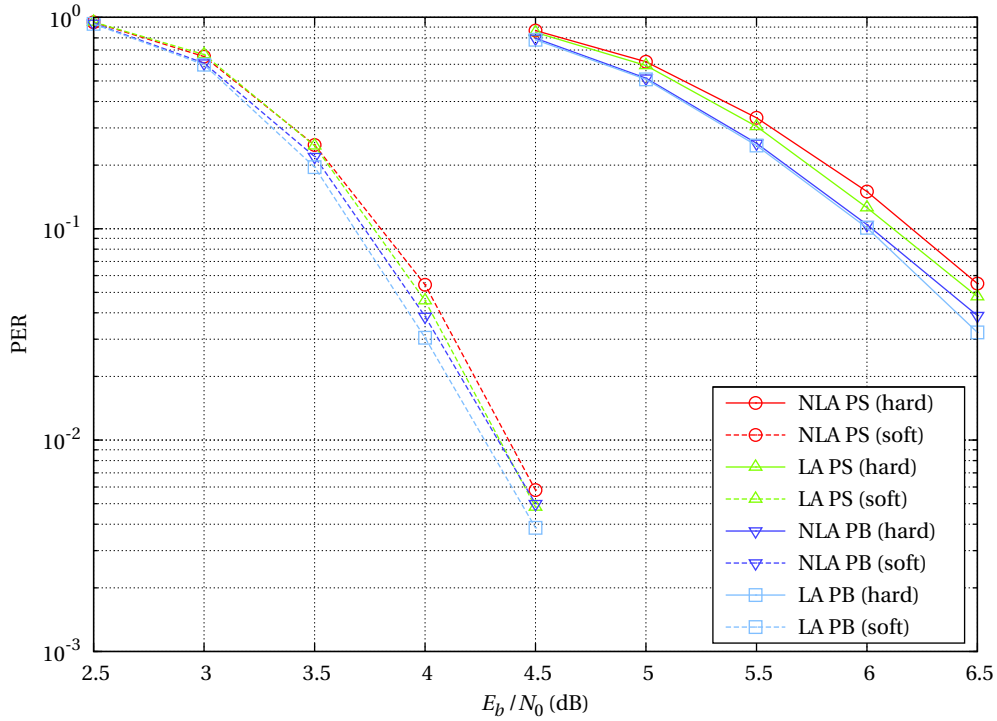

Figure 6.13: Performance for MAP decoder for adaptive ternary model with $M = 256$ and $\varepsilon = 0.1$; without look-ahead (NLA) and with look-ahead (LA); and with $P(\mathbf{t})$ adjusted every symbol (PS) or every bit (PB).

6.2.2 The adaptive ternary model

This section deals with the scheme having an adaptive model with three symbols in the source alphabet. Figure 6.13 shows the PER with $M = 256$ and $\varepsilon = 0.1$. The code rate for this code is 8/9, as for the static binary model with $\varepsilon = 0.05$, which was shown in Figure 6.5.

For hard decoding, the improvement in performance provided by the two mentioned techniques is very similar to that for the static binary model described before. However, for soft decoding, the PER of the original scheme and the scheme with look-ahead are very close, indicating that look-ahead does not improve the PER. When the prior probability $P(\mathbf{t})$ is adjusted every bit, the PER gets lower. When look-ahead is combined with this technique, it gives a further enhancement to the PER. This is because look-ahead has two effects on the decoder; it may detect errors earlier, and it may decode symbols earlier. Recall that the scheme with $P(\mathbf{t})$ adjusted for every bit corrects for the forbidden gap every time a symbol is decoded. This may be why look-ahead improves the performance of this scheme; the correction for the forbidden gap comes earlier, helping to improve the search in the decoding tree.

This indicates that the improvements are suitable for adaptive models as well as for static models, and also when there are more than two symbols.

Chapter 7

Conclusion

In this project, a joint source-channel arithmetic coding scheme proposed by [11] for decoding arithmetic codes with a forbidden symbol transmitted over an AWGN channel was analysed and implemented. The proposed scheme was a MAP decoder with a novel MAP metric. The decoder uses the stack sequential search algorithm, with maximum number of nodes M, to find the path with the best MAP metric. In [11], this scheme was shown to have a better error-correction performance than a separated scheme consisting of arithmetic coding with no forbidden gap, and a separate channel coding scheme based on RCPC codes.

The decoding scheme was implemented in C++ and simulated for an AWGN channel with both hard and soft decoding for BPSK modulation. The results obtained from the simulation conform to the results published in [11].

This joint source-channel arithmetic code has several advantages. The encoder has the same complexity as an arithmetic encoder with a forbidden symbol. The amount of redundancy in the encoded message can be controlled by tuning one parameter, the forbidden gap factor ε. This factor can be predetermined or varied adaptively according to channel conditions.

The decoder algorithm can be scaled according to complexity and memory constraints. If more memory and more processing power is available, the decoder can be improved by increasing the number of nodes in the stack, M.

In this dissertation, new techniques were introduced in order to improve the error-correction performance of the code. The decoder was improved with a look-ahead technique that enables the decoder to decode symbols earlier, and to detect errors earlier, in certain conditions. When used with the MAP decoder, this technique improves the PER at the cost of a small increase in complexity.

The calculation of one component of the MAP metric, the prior probability $P(\mathbf{t})$, was improved as well. In [11] the prior probability component of the metric is updated every time a symbol is decoded. A proposed improvement is to update the prior

probability every time a bit is passed into the decoder, which leads to faster updating of the metric and, consequently, to a better error-correction performance. This technique makes the MAP decoder faster as well; the decoding time of the MAP decoder is decreased because the correct path in the decoding tree is found earlier.

Using these techniques, a coding gain of up to 0.4 dB for soft decoding and 0.6 dB for hard decoding was observed for a code rate of 2/3 and $M = 256$.

When a channel has a low signal-to-noise ratio E_b/N_0, a high code rate is used. The improvements mentioned are more effective in such cases, and the performance gain in terms of signal-to-noise ratio is higher. Also, the improvements are more effective when $M = 256$ than when $M = 4096$, indicating that they offer an advantage when processing or memory capabilities are limited.

The system was simulated using a static source model having two symbols in its alphabet, and using an adaptive source model having three symbols in its alphabet. Similar results were obtained, indicating that the algorithms employed are adequate for both static models and adaptive models, and for two or more symbols in the source alphabet.

According to [11], the scheme proposed by [11] has been used profitably in image transmission. A number of multimedia applications are using arithmetic coding as a final lossless compression stage after a lossy compression stage. These applications then use a channel code to transmit multimedia data over wireless channels. The scheme presented in this dissertation could be used to combine the arithmetic compression stage and the channel coding stage. This would provide the benefits mentioned above.

7.1 Future work

The forbidden gap used in this project was constant throughout the process, use of adaptive gaps can be analysed. The gap can vary as the channel conditions vary, so the gap width can be adapted between packets. Also, the gap within one packet can vary from the beginning till the end of the packet and the effect analysed. Since an error at the start of the packet has more chance of being detected then an error at the end of the packet, adapting the gap within the same packet may improve performance.

When an error is detected in the implemented scheme, the whole packet is discarded. If the error occurs late in the packet, it may be possible to salvage data at the beginning of the packet, which may reduce the amount of retransmission necessary.

Joint source-channel coding for arithmetic codes was used in a sequential decoding algorithm in this work. Joint coding using arithmetic coding in iterative decoding techniques is an area of current research. Iterative joint source-channel coding using

arithmetic codes could be developed using metrics similar to the metrics presented in this dissertation.

Appendix A

Source Code Organization

The source code is split into several files. An overview of the file contents is given below.

misc.hpp, misc.cpp define some utility functions.

The `next_power_2_m1()` template function takes an integer and returns the minimum integer greater than or equal to the parameter which is of the form $2^n - 1$, where n is an integer.

The `next_power_2()` template function takes an integer and returns the minimum number greater than or equal to the parameter which is of the form 2^n, where n is an integer.

The `rnd` class produces random numbers. It supports floating point random numbers distributed linearly, integer random numbers distributed linearly, and floating point random numbers with the Gaussian distribution. The ideas were taken from [18].

The `ltqnorm()` function takes a parameter p and returns the lower tail quantile for the standard normal distribution. That is, it returns z satisfying $P(X < z) = p$, where X has a standard normal distribution. The algorithm used is presented in [20].

fix.hpp defines the `fix` template class, which is a generic implementation of fixed-point arithmetic. The class supports arithmetic operations `+`, `-`, `*` and `/`; bitwise and shift operations `~`, `&`, `^`, `|`, `<<` and `>>`; relational operations `==`, `!=`, `<`, `<=`, `>` and `>=`; and the logical operation `!`. It also allows explicit conversion to/from `double`; to avoid difficult-to-find bugs, no implicit conversions to/from `double` are provided by this class. The class also allows access to the internal integer representation.

coder.hpp, coder.cpp define some classes that are used for arithmetic encoding and decoding.

The `gap_place_t` enum is used to choose where to place the forbidden gap. It can take values `gap_begin` and `gap_end` for the gap to be at the beginning or at the end of the interval, `gap_mid` for the gap to be distributed between the symbols, and `gap_begin_mod` for the gap to be placed as the beginning of the interval and for the sub-intervals to be swapped such that the sub-interval for the most probable symbol is at the beginning, next to the forbidden gap (the scheme described in § 4.1).

The `coding_info` struct holds information used to define an arithmetic encoder, the forbidden gap factor ε, the termination symbol probability ω, the number of symbols in one packet L, and the placement of the forbidden gap.

The `decoding_info` struct inherits `coding_info` and adds information used for the arithmetic decoder and for the MAP decoder.

The `model` abstract base class defines the interface required for a class that acts as a source model.

The `encoder` class encodes a sequence of L symbols into a bit sequence. This class works incrementally, and output bits can be read before all L symbols are passed to the encoding instance.

The `decoder` class decodes a bit sequence into a sequence of symbols. This class work incrementally, and output symbols can be read before all of the bit sequence is passed to the decoding instance. Errors are detected, and the class supports decoding with and without look-ahead (see § 4.2).

map.hpp, map.cpp define some functions and classes that are used for MAP decoding.

The `soft_signal_probs` function takes E_b/N_0 and the signal vector $\mathbf{y}$ as inputs. It calculates $P(y_n \mid t_n = 0)/P(y_n)$ and $P(y_n \mid t_n = 1)/P(y_n)$ for each value element y_n of $\mathbf{y}$, assuming a soft channel.

The `hard_signal_probs` function takes E_b/N_0 and the signal vector $\mathbf{y}$ as inputs. It calculates $P(y_n \mid t_n = 0)/P(y_n)$ and $P(y_n \mid t_n = 1)/P(y_n)$ for each value element y_n of $\mathbf{y}$, assuming a hard channel.

The `maps_decoder` class performs MAP decoding using the stack sequential search technique, where each edge of the decoding tree represents one bit.

The `mapcs_decoder` class performs MAP decoding using the stack sequential search technique, where each edge of the decoding tree represents one symbol

instead of one bit.

simulation.hpp, simulation.cpp define some functions and classes used in simulating the MAP decoder.

The `generate_symbols` function generates a symbol sequence **u** where the probability of each symbol is taken from the source model.

The `generate_bits` function encodes a symbol sequence **u** into a bit sequence **t** given a source model.

The `generate_signals` function adds AWGN noise with the given signal to noise ratio E_b/N_0 to a bit sequence **t**, and returns the signal vector **y**.

The `decode_signals` function decodes a signal vector **y** using MAP decoding.

The `check_decode_signals` function is given a signal vector **y** and the actual source sequence **u** and **t**. It uses MAP decoding to decode the signal vector, checking continuously for an error in the decoding process. This is used to detect failures in the decoding early in simulations.

The `result` struct is used to return the result of a simulation.

The `simulator` class performs a simulation with a specific configuration and for a specific number of times.

detection_delay.cpp contains code to analyse the delay in detecting errors for arithmetic codes with integrated error detection.

map_simulate.cpp contains code to find the packet error rate of specified MAP decoders by simulation.

Makefile is used to build the programs from the C++ source.

References

[1] John G. Proakis, *Digital Communications*, 4th ed. McGraw-Hill, 2001, ch. 1, pp. 1–16.

[2] Jorma J. Rissanen, "Generalized Kraft inequality and arithmetic coding," *IBM Journal of Research and Development*, vol. 20, no. 3, pp. 198–203, May 1976.

[3] David A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[4] Sridhar Vembu, Sergio Verdù, and Yossef Steinberg, "The sourch-channel separation theorem revisited," *IEEE Transactions on Information Theory*, vol. 41, no. 1, pp. 44–54, Jan. 1995.

[5] Colin Boyd, John G. Cleary, Sean A. Irvine, Ingrid Rinsma-Melchert, and Ian H. Witten, "Integrating error detection into arithmetic coding," *IEEE Transactions on Communications*, vol. 45, no. 1, pp. 1–3, Jan. 1997.

[6] Ian H. Witten, Radford M. Neal, and John G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, Jun. 1987.

[7] Debra A. Lelewer and Daniel S. Hirschberg, "Data compression," *ACM Computing Surveys*, no. 3, pp. 261–296, Sep. 1987.

[8] Jossy Sayir, "Arithmetic coding for noisy channels," in *Proceedings of the 1999 IEEE Information Theory and Communications Workshop*, Jun. 1999, pp. 69–71.

[9] David J. C. MacKay, *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003, ch. 25, pp. 324–333.

[10] Thomas Guionnet and Christine Guillemot, "Soft decoding and synchronization of arithmetic codes: Application to image transmission over noisy channels," *IEEE Transactions on Image Processing*, vol. 12, no. 12, pp. 1599–1609, Dec. 2003.

[11] Marco Grangetto, Pamela Cosman, and Gabriella Olmo, "Joint source/channel coding and MAP decoding of arithmetic codes," *IEEE Transactions on Communications*, vol. 53, no. 6, pp. 1007–1016, Jun. 2005.

[12] Joachim Hagenauer, "Rate-compatible punctured convolutional codes (RCPC codes) and their applications," *IEEE Transactions on Communications*, vol. 36, no. 4, pp. 389–400, Apr. 1988.

[13] Marco Grangetto, Bartolo Scanavino, Gabriella Olmo, and Sergio Benedetto, "Iterative decoding of serially concatenated arithmetic and channel codes with jpeg 2000 applications," *IEEE Transactions on Image Processing*, vol. 16, no. 6, pp. 1557–1567, Jun. 2007.

[14] Lalit R. Bahl, John Cocke, Frederick Jelinek, and Josef Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Transactions on Information Theory*, vol. 20, no. 2, pp. 284–287, Mar. 1974.

[15] Moonseo Park and David J. Miller, "Joint source-channel decoding for variable-length encoded data by exact and approximate MAP sequence estimation," *IEEE Transactions on Communications*, vol. 48, no. 1, pp. 1–6, Jan. 2000.

[16] Frederick Jelinek, "Fast sequential decoding algorithm using a stack," *IBM Journal of Research and Development*, vol. 13, no. 6, pp. 675–685, Nov. 1969.

[17] Bjarne Stroustrup, *The C++ Programming Language*, 3rd ed. Addison Wesley, 1997.

[18] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, 1992, ch. 7, pp. 274–328.

[19] Edwin B. Wilson, "Probable inference, the law of succession, and statistical inference," *Journal of the American Statistical Association*, vol. 22, pp. 209–212, 1927.

[20] Peter J. Acklam. An algorithm for computing the inverse normal cumulative distribution function. [Online]. Available: http://home.online.no/~pjacklam/notes/invnorm/index.html