University of Glasgow | School of Engineering

2 Digital Communications 4: Forward Error Correction

2.1 Introduction

This laboratory project will introduce you to some issues that occur in digital communication channels. In particular we will study Forward Error Correction (FEC) as a technique to overcome the detrimental effects of noise on the communication channel. Again, we will circumvent some of the low-level coding by using the `komm` Python library `https://pypi.org/project/komm/` to provide the appropriate functionality. If you are using your own computer, make sure the Python libraries `scipy`, `numpy` and `matplotlib` are installed, in addition to a suitable library for reading grayscale images, such as `pillow`. It is recommended that you use a suitable IDE for your project, such as Spyder.

This laboratory will be scheduled over a 3 hour session, where you will be able to ask the lecturer for help. If you do not complete the project in this time, you should undertake to complete it in your own time. The laboratory project should be written up as a short report describing what you've done and the results you have taken along with any conclusions that you draw. Include your python code(s) in the appendices. Make sure your name and registration number is on the report, and, if you have worked as a pair, include the name of your partner. The report should be uploaded to the Moodle assignment by the stated deadline, either using Moodle's inbuilt editor, or as a single PDF.

2.2 Block Coding: BCH codes

Block codes work on fixed-size blocks (packets) of bits or symbols of predetermined size. Practical block codes can generally be hard-decoded in polynomial time to their block length. Bose-Chaudhuri-Hocquenghem (BCH) codes form a class of cyclic error-correcting codes that are constructed using polynomials over a finite field (also called Galois field). One of the key features of BCH codes is that during code design, there is a precise control over the number of symbol errors correctable by the code. In particular, it is possible to design binary BCH codes that can correct multiple bit errors. Another advantage of BCH codes is the ease with which they can be decoded, namely, via an algebraic method known as syndrome decoding. This simplifies the design of the decoder for these codes, using small low-power electronic hardware. BCH codes are used in applications such as satellite communications, compact disc players, DVDs, disk drives, solid-state drives and QR codes.

Table 2.2 lists indexes for a sample of some short primitive narrow-sense BCH binary codes which can correct up to t bit errors, and which will be used in this laboratory project. The parameters for the code require a code word length $n = 2^m - 1$ and $n - k \leq mt$. The setup to use the BCH code class in python is on providing suitable values for m and t:

```
import komm
code = komm.BCHCode(m,t)
n,k = code.length, code.dimension
```

m	n	k	t
3	7	4	1
4	15	5	3
5	31	6	7
6	63	24	7
6	63	16	11
6	63	10	13

Table 2.2: indexes for a sample of some short primitive narrow-sense BCH binary codes

2.2.1 (7,4) code alphabet

The (n =7,k =4) BCH code (top line of table 2.2) is equivalent to the Hamming (7,4) code. First, generate the alphabet of valid 7-bit codewords by examining `code.codeword_table`. Confirm that cyclic permutations of valid 7-bit codewords, and bitwise XOR of two valid 7-bit codewords result in other valid 7-bit codewords. The corresponding generator polynomial can be obtained from `code.generator_polynomial`

2.2.2 Forward error correction using BCH codes

A number of 8 bit depth grayscale images of various sizes have been provided for you to use in this laboratory project. You may also consider the use of the `numpy.random.randint()` command to create random binary streams for testing purposes. As the runtime depends on the size of the data, you should generally use the smallest data set first, although in terms of accuracy it may be advisable to use the larger datasets where the smallest images result in a small number of bit errors, say < 25, to improve your accuracy in determining the bit-error-rate.

In this laboratory project, use quadrature phase shift keying (QPSK) with unit average power per symbol. Begin with your python code from the previous laboratory for noisy channel simulation. Having set up the code as shown above, coding and decoding are done with:

```
coded_word = code.encode(message_word)
...
message_word = code.decode(coded_word)
```

where the `message_word` is a fixed length `k` bits and `coded_word` is a fixed length `n` bits. You will therefore need a program loop to code (and decode) your entire binary data.

Plot *ber* (logarithmic axis) vs *snr* (in dB) over a suitable range of signal-to-noise, for the BCH codes listed in table 2.2. Compare your results with your measurements for the bit error ratio with QPSK without error correction, as you investigated in the previous laboratory project.

2.3 Convolutional Coding

Convolutional codes work on bit or symbol streams of arbitrary length. They are most often soft decoded with the Viterbi algorithm, though other algorithms are sometimes used. Viterbi decoding allows asymptotically optimal decoding efficiency with increasing constraint length of the convolutional code, but at the expense of exponentially increasing complexity.

The `komm` library provides functions for encoding and decoding convolutional codes. There are a number of options. In particular the best convolutional code performance is normally with *soft*-decision decoding on the real values returned from the demodulator based on the euclidean distance from the symbol, thus providing a confidence level to the Viterbi decoder.

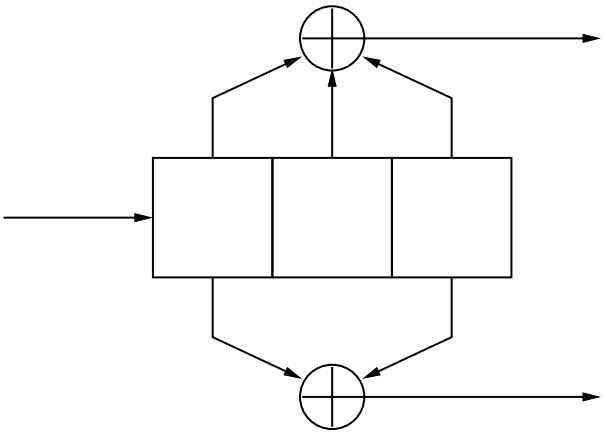

Figure 2.3: A simple rate $\frac{1}{2}$ convolutional encoder.

The constraint length L is the number of shift registers, including the input. Taking the simple rate $\frac{1}{2}$ convolutional encoder shown schematically in figure 2.3 as an example, this has a constraint length of 3. The code generator represents the connections to the modulo 2 adders. In this case the code generator is

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 7 \\ 5 \end{bmatrix}$$

where the right expression is the equivalent octal representation.

```
import komm
code = komm.ConvolutionalCode(feedforward_polynomials=[[0o7, 0o5]])
tblen = 18
encoder = komm.ConvolutionalStreamEncoder(code)
decoder = komm.ConvolutionalStreamDecoder(code,
       traceback_length=tblen, input_type="hard")
...
rx_enc = psk.demodulate(rx_data, decision_method="hard")
```

`tblen` is a positive integer scalar that specifies the traceback depth in the Viterbi algorithm. When used in stream (continuous) form, the decoder has a delay (latency) equal to `tblen` for a single input stream code. Therefore you should append `tblen` zeros to the input binary stream, and discard the first `tblen` bits of the output stream. Typical values for a traceback depth `tblen` are about five or six times the constraint length L. Remember, you will also need to ensure your encoded data stream corresponds to a whole number of modulation format symbols. With `"hard"` decoding, `rx_enc` should contain strict binary data. However with `"soft"` decoding, `rx_enc` should contain real values.

Again, take your python code from the previous laboratory for noisy channel simulation and adapt it to use convolutional coding. Plot *ber* (logarithmic axis) vs *snr* (in dB) over a suitable range of signal-to-noise for both `"hard"` and `"soft"` decoding. Compare your results with your measurements for the bit error ratio with QPSK without error correction from the previous laboratory project.

The Jet Propulsion Laboratory developed constraint length 7 convolutional codes for the Voyager missions. The rate $\frac{1}{2}$ code uses `feedforward_polynomials=[[0o155, 0o117]]` and the rate $\frac{1}{3}$ code uses `feedforward_polynomials=[[0o155, 0o117, 0o127]]`. Again, plot *ber* (logarithmic axis) vs *snr* (in dB) over a suitable range of signal-to-noise for each of these using `"soft"` decoding.

2.4 Concatenated Codes

Compare the various error-correction techniques (simple parity check and ARQs, BCH block codes, Convolutional Code with hard/soft decision) you examined for the configurations given in the lab descriptor. Which of these provides the best error correcting for the data provided in cases of values for snr =3 dB using QPSK? What is the code rate in each optimal case?

Many implementations of FEC can tolerate even lower signal-to-noise ratio by concatenating two different FEC code methods. Now adapt your code to use a convolutional code as an inner code, modulating as QPSK and passing the result through an AWGN channel, demodulate and decode using the soft-decision Viterbi method. Now use this as an effective transmission channel and apply a BCH code as the outer code. What is the lowest ber you can obtain with the codes listed here for the case where the signal power is equal to the noise, i.e. snr = 0 dB?

2.5 Documentation

Python 3 `https://docs.python.org/3/`

komm `https://komm.readthedocs.io/en/latest/`

numpy and scipy `https://docs.scipy.org/doc/`

matplotlib `https://matplotlib.org/contents.html`

pillow `https://pillow.readthedocs.io`

spyder `https://docs.spyder-ide.org/`

Getting the python libraries

If you are using your own computer, make sure the Python libraries `scipy`, `numpy`, `matplotlib` and `pillow` are installed. These libraries are installed by default with the Anaconda python distribution. It is recommended that you use a suitable IDE for your project, such as Spyder. The `komm` Python library is available from PyPI repository and, if required, can be installed using `pip`. From a python-activated command line (such as Anaconda prompt), use the following command to install in your user App config

```
pip install komm --user
```