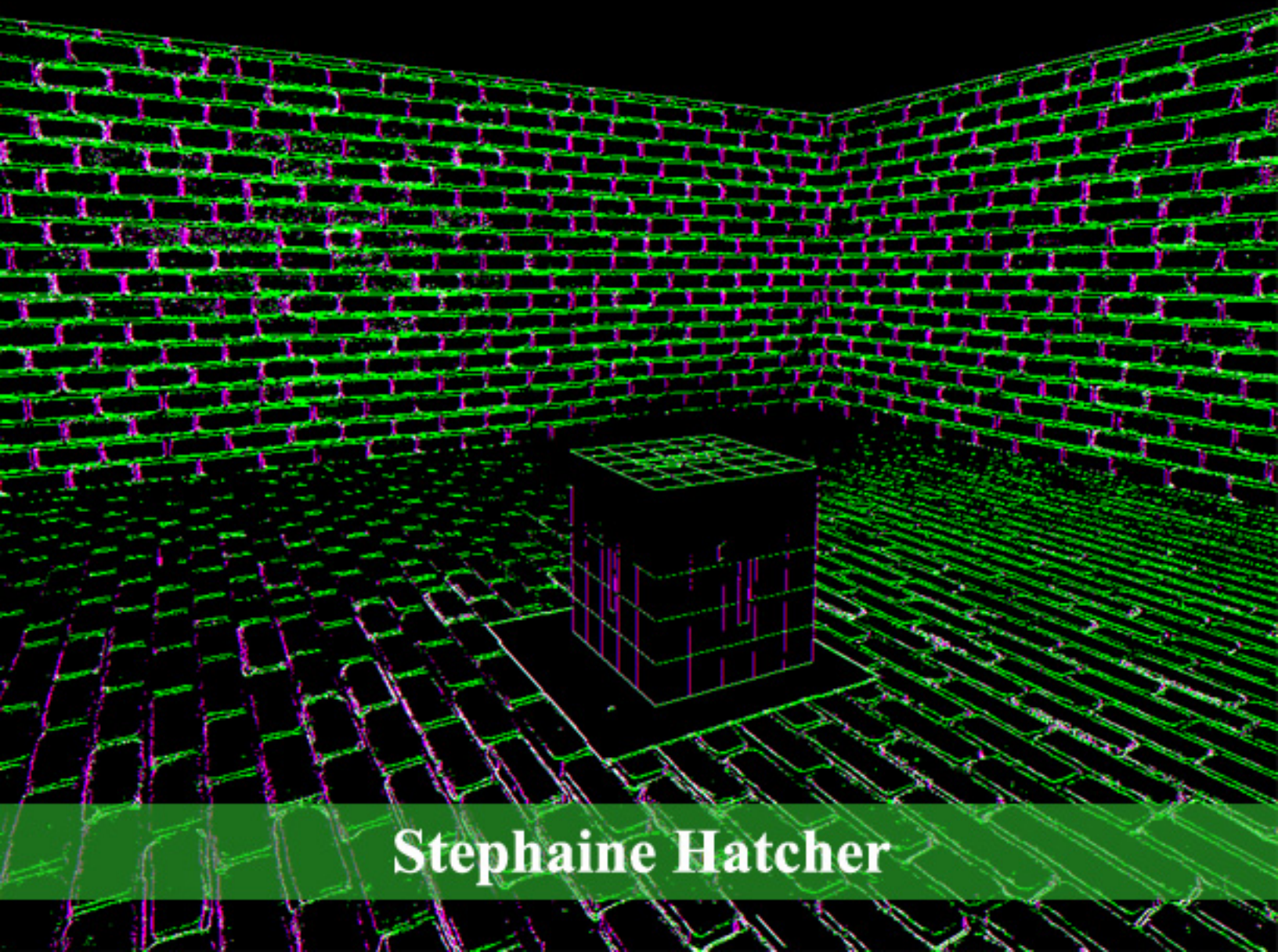
Error Detection and Correction in Information and Coding Theory

(Concepts and Applications)



Stephaine Hatcher

First Edition, 2012

ISBN 978-81-323-3451-4

Published by:
Research World
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter-1

Error Detection and Correction

In information theory and coding theory with applications in computer science and telecommunication, **error detection and correction** or **error control** are techniques that enable reliable delivery of digital data over unreliable communication channels. Many communication channels are subject to channel noise, and thus errors may be introduced during transmission from the source to a receiver. Error detection techniques allow detecting such errors, while error correction enables reconstruction of the original data.

The general definitions of the terms are as follows:

- *Error detection* is the detection of errors caused by noise or other impairments during transmission from the transmitter to the receiver.
- *Error correction* is the detection of errors and reconstruction of the original, error-free data.

Error correction may generally be realized in two different ways:

- *Automatic repeat request (ARQ)* (sometimes also referred to as *backward error correction*): This is an error control technique whereby an error detection scheme is combined with requests for retransmission of erroneous data. Every block of data received is checked using the error detection code used, and if the check fails, retransmission of the data is requested – this may be done repeatedly, until the data can be verified.
- *Forward error correction (FEC)*: The sender encodes the data using an *error-correcting code (ECC)* prior to transmission. The additional information (redundancy) added by the code is used by the receiver to recover the original data. In general, the reconstructed data is what is deemed the "most likely" original data.

ARQ and FEC may be combined, such that minor errors are corrected without retransmission, and major errors are corrected via a request for retransmission: this is called *hybrid automatic repeat-request (HARQ)*.

Introduction

The general idea for achieving error detection and correction is to add some redundancy (i.e., some extra data) to a message, which receivers can use to check consistency of the delivered message, and to recover data determined to be erroneous. Error-detection and correction schemes can be either systematic or non-systematic: In a systematic scheme, the transmitter sends the original data, and attaches a fixed number of *check bits* (or *parity data*), which are derived from the data bits by some deterministic algorithm. If only error detection is required, a receiver can simply apply the same algorithm to the received data bits and compare its output with the received check bits; if the values do not match, an error has occurred at some point during the transmission. In a system that uses a non-systematic code, the original message is transformed into an encoded message that has at least as many bits as the original message.

Good error control performance requires the scheme to be selected based on the characteristics of the communication channel. Common channel models include memory-less models where errors occur randomly and with a certain probability, and dynamic models where errors occur primarily in bursts. Consequently, error-detecting and correcting codes can be generally distinguished between *random-error-detecting/correcting* and *burst-error-detecting/correcting*. Some codes can also be suitable for a mixture of random errors and burst errors.

If the channel capacity cannot be determined, or is highly varying, an error-detection scheme may be combined with a system for retransmissions of erroneous data. This is known as automatic repeat request (ARQ), and is most notably used in the Internet. An alternate approach for error control is hybrid automatic repeat request (HARQ), which is a combination of ARQ and error-correction coding.

Error detection schemes

Error detection is most commonly realized using a suitable hash function (or checksum algorithm). A hash function adds a fixed-length *tag* to a message, which enables receivers to verify the delivered message by recomputing the tag and comparing it with the one provided.

There exists a vast variety of different hash function designs. However, some are of particularly widespread use because of either their simplicity or their suitability for detecting certain kinds of errors (e.g., the cyclic redundancy check's performance in detecting burst errors).

Random-error-correcting codes based on minimum distance coding can provide a suitable alternative to hash functions when a strict guarantee on the minimum number of errors to be detected is desired. Repetition codes, described below, are special cases of error-correcting codes: although rather inefficient, they find applications for both error correction and detection due to their simplicity.

Repetition codes

A **repetition code** is a coding scheme that repeats the bits across a channel to achieve error-free communication. Given a stream of data to be transmitted, the data is divided into blocks of bits. Each block is transmitted some predetermined number of times. For example, to send the bit pattern "1011", the four-bit block can be repeated three times, thus producing "1011 1011 1011". However, if this twelve-bit pattern was received as "1010 1011 1011" – where the first block is unlike the other two – it can be determined that an error has occurred.

Repetition codes are not very efficient, and can be susceptible to problems if the error occurs in exactly the same place for each group (e.g., "1010 1010 1010" in the previous example would be detected as correct). The advantage of repetition codes is that they are extremely simple, and are in fact used in some transmissions of numbers stations.

Parity bits

A **parity bit** is a bit that is added to a group of source bits to ensure that the number of set bits (i.e., bits with value 1) in the outcome is even or odd. It is a very simple scheme that can be used to detect single or any other odd number (i.e., three, five, etc.) of errors in the output. An even number of flipped bits will make the parity bit appear correct even though the data is erroneous.

Extensions and variations on the parity bit mechanism are horizontal redundancy checks, vertical redundancy checks, and "double," "dual," or "diagonal" parity (used in RAID-DP).

Checksums

A **checksum** of a message is a modular arithmetic sum of message code words of a fixed word length (e.g., byte values). The sum may be negated by means of a one's-complement prior to transmission to detect errors resulting in all-zero messages.

Checksum schemes include parity bits, check digits, and longitudinal redundancy checks. Some checksum schemes, such as the Luhn algorithm and the Verhoeff algorithm, are specifically designed to detect errors commonly introduced by humans in writing down or remembering identification numbers.

Cyclic redundancy checks (CRCs)

A **cyclic redundancy check (CRC)** is a single-burst-error-detecting cyclic code and non-secure hash function designed to detect accidental changes to digital data in computer networks. It is characterized by specification of a so-called *generator polynomial*, which is used as the divisor in a polynomial long division over a finite field, taking the input data as the dividend, and where the remainder becomes the result.

Cyclic codes have favorable properties in that they are well suited for detecting burst errors. CRCs are particularly easy to implement in hardware, and are therefore commonly used in digital networks and storage devices such as hard disk drives.

Even parity is a special case of a cyclic redundancy check, where the single-bit CRC is generated by the divisor $x+1$.

Cryptographic hash functions

A **cryptographic hash function** can provide strong assurances about data integrity, provided that changes of the data are only accidental (i.e., due to transmission errors). Any modification to the data will likely be detected through a mismatching hash value. Furthermore, given some hash value, it is infeasible to find some input data (other than the one given) that will yield the same hash value. Message authentication codes, also called *keyed* cryptographic hash functions, provide additional protection against intentional modification by an attacker.

Error-correcting codes

Any error-correcting code can be used for error detection. A code with *minimum Hamming distance*, d, can detect up to $d-1$ errors in a code word. Using minimum-distance-based error-correcting codes for error detection can be suitable if a strict limit on the minimum number of errors to be detected is desired.

Codes with minimum Hamming distance $d=2$ are degenerate cases of error-correcting codes, and can be used to detect single errors. The parity bit is an example of a single-error-detecting code.

The Berger code is an early example of a unidirectional error(-correcting) code that can detect any number of errors on an asymmetric channel, provided that only transitions of cleared bits to set bits *or* set bits to cleared bits can occur.

Error correction

Automatic repeat request

Automatic Repeat reQuest (ARQ) is an error control method for data transmission that makes use of error-detection codes, acknowledgment and/or negative acknowledgment messages, and timeouts to achieve reliable data transmission. An *acknowledgment* is a message sent by the receiver to indicate that it has correctly received a data frame.

Usually, when the transmitter does not receive the acknowledgment before the timeout occurs (i.e., within a reasonable amount of time after sending the data frame), it retransmits the frame until it is either correctly received or the error persists beyond a predetermined number of retransmissions.

Three types of ARQ protocols are Stop-and-wait ARQ, Go-Back-N ARQ, and Selective Repeat ARQ.

ARQ is appropriate if the communication channel has varying or unknown capacity, such as is the case on the Internet. However, ARQ requires the availability of a back channel, results in possibly increased latency due to retransmissions, and requires the maintenance of buffers and timers for retransmissions, which in the case of network congestion can put a strain on the server and overall network capacity.

Error-correcting code

An error-correcting code (ECC) or forward error correction (FEC) code is a system of adding redundant data, or *parity data*, to a message, such that it can be recovered by a receiver even when a number of errors (up to the capability of the code being used) were introduced, either during the process of transmission, or on storage. Since the receiver does not have to ask the sender for retransmission of the data, a back-channel is not required in forward error correction, and it is therefore suitable for simplex communication such as broadcasting. Error-correcting codes are frequently used in lower-layer communication, as well as for reliable storage in media such as CDs, DVDs, hard disks, and RAM.

Error-correcting codes are usually distinguished between convolutional codes and block codes:

- *Convolutional codes* are processed on a bit-by-bit basis. They are particularly suitable for implementation in hardware, and the Viterbi decoder allows optimal decoding.

- *Block codes* are processed on a block-by-block basis. Early examples of block codes are repetition codes, Hamming codes and multidimensional parity-check codes. They were followed by a number of efficient codes, Reed-Solomon codes being the most notable due to their current widespread use. Turbo codes and low-density parity-check codes (LDPC) are relatively new constructions that can provide almost optimal efficiency.

Shannon's theorem is an important theorem in forward error correction, and describes the maximum information rate at which reliable communication is possible over a channel that has a certain error probability or signal-to-noise ratio (SNR). This strict upper limit is expressed in terms of the channel capacity. More specifically, the theorem says that there exist codes such that with increasing encoding length the probability of error on a discrete memoryless channel can be made arbitrarily small, provided that the code rate is smaller than the channel capacity. The code rate is defined as the fraction k/n of k source symbols and n encoded symbols.

The actual maximum code rate allowed depends on the error-correcting code used, and may be lower. This is because Shannon's proof was only of existential nature, and did not

show how to construct codes which are both optimal and have efficient encoding and decoding algorithms.

Hybrid schemes

Hybrid ARQ is a combination of ARQ and forward error correction. There are two basic approaches:

- Messages are always transmitted with FEC parity data (and error-detection redundancy). A receiver decodes a message using the parity information, and requests retransmission using ARQ only if the parity data was not sufficient for successful decoding (identified through a failed integrity check).
- Messages are transmitted without parity data (only with error-detection information). If a receiver detects an error, it requests FEC information from the transmitter using ARQ, and uses it to reconstruct the original message.

The latter approach is particularly attractive on an erasure channel when using a rateless erasure code.

Applications

Applications that require low latency (such as telephone conversations) cannot use Automatic Repeat reQuest (ARQ); they must use Forward Error Correction (FEC). By the time an ARQ system discovers an error and re-transmits it, the re-sent data will arrive too late to be any good.

Applications where the transmitter immediately forgets the information as soon as it is sent (such as most television cameras) cannot use ARQ; they must use FEC because when an error occurs, the original data is no longer available. (This is also why FEC is used in data storage systems such as RAID and distributed data store).

Applications that use ARQ must have a return channel. Applications that have no return channel cannot use ARQ.

Applications that require extremely low error rates (such as digital money transfers) must use ARQ.

The Internet

In a typical TCP/IP stack, error control is performed at multiple levels:

- Each Ethernet frame carries a CRC-32 checksum. Frames received with incorrect checksums are discarded by the receiver hardware.
- The IPv4 header contains a checksum protecting the contents of the header. Packets with mismatching checksums are dropped within the network or at the receiver.

- The checksum was omitted from the IPv6 header in order to minimize processing costs in network routing and because current link layer technology is assumed to provide sufficient error detection.
- UDP has an optional checksum covering the payload and addressing information from the UDP and IP headers. Packets with incorrect checksums are discarded by the operating system network stack. The checksum is optional under IPv4, only, because the IP layer checksum may already provide the desired level of error protection.
- TCP provides a checksum for protecting the payload and addressing information from the TCP and IP headers. Packets with incorrect checksums are discarded within the network stack, and eventually get retransmitted using ARQ, either explicitly (such as through triple-ack) or implicitly due to a timeout.

Deep-space telecommunications

Development of error-correction codes was tightly coupled with the history of deep-space missions due to the extreme dilution of signal power over interplanetary distances, and the limited power availability aboard space probes. Whereas early missions sent their data uncoded, starting from 1968 digital error correction was implemented in the form of (sub-optimally decoded) convolutional codes and Reed-Muller codes. The Reed-Muller code was well suited to the noise the spacecraft was subject to (approximately matching a bell curve), and was implemented at the Mariner spacecraft for missions between 1969 and 1977.

The Voyager 1 and Voyager 2 missions, which started in 1977, were designed to deliver color imaging amongst scientific information of Jupiter and Saturn. This resulted in increased coding requirements, and thus the spacecraft were supported by (optimally Viterbi-decoded) convolutional codes that could be concatenated with an outer Golay (24,12,8) code. The Voyager 2 probe additionally supported an implementation of a Reed-Solomon code: the concatenated Reed-Solomon-Viterbi (RSV) code allowed for very powerful error correction, and enabled the spacecraft's extended journey to Uranus and Neptune.

The CCSDS currently recommends usage of error correction codes with performance similar to the Voyager 2 RSV code as a minimum. Concatenated codes are increasingly falling out of favor with space missions, and are replaced by more powerful codes such as Turbo codes or LDPC codes.

The different kinds of deep space and orbital missions that are conducted suggest that trying to find a "one size fits all" error correction system will be an ongoing problem for some time to come. For missions close to earth the nature of the channel noise is different from that of a spacecraft on an interplanetary mission experiences. Additionally, as a spacecraft increases its distance from earth, the problem of correcting for noise gets larger.

Satellite broadcasting (DVB)

The demand for satellite transponder bandwidth continues to grow, fueled by the desire to deliver television (including new channels and High Definition TV) and IP data. Transponder availability and bandwidth constraints have limited this growth, because transponder capacity is determined by the selected modulation scheme and Forward error correction (FEC) rate.

Overview

- QPSK coupled with traditional Reed Solomon and Viterbi codes have been used for nearly 20 years for the delivery of digital satellite TV.
- Higher order modulation schemes such as 8PSK, 16QAM and 32QAM have enabled the satellite industry to increase transponder efficiency by several orders of magnitude.
- This increase in the information rate in a transponder comes at the expense of an increase in the carrier power to meet the threshold requirement for existing antennas.
- Tests conducted using the latest chipsets demonstrate that the performance achieved by using Turbo Codes may be even lower than the 0.8 dB figure assumed in early designs.

Data storage

Error detection and correction codes are often used to improve the reliability of data storage media.

A "parity track" was present on the first magnetic tape data storage in 1951. The "Optimal Rectangular Code" used in group code recording tapes not only detects but also corrects single-bit errors.

Some file formats, particularly archive formats, include a checksum (most often CRC32) to detect corruption and truncation and can employ redundancy and/or parity files to recover portions of corrupted data.

Reed Solomon codes are used in compact discs to correct errors caused by scratches.

Modern hard drives use CRC codes to detect and Reed-Solomon codes to correct minor errors in sector reads, and to recover data from sectors that have "gone bad" and store that data in the spare sectors.

RAID systems use a variety of error correction techniques, to correct errors when a hard drive completely fails.

Error-correcting memory

DRAM memory may provide increased protection against soft errors by relying on error correcting codes. Such error-correcting memory, known as *ECC* or *EDAC-protected* memory, is particularly desirable for high fault-tolerant applications, such as servers, as well as deep-space applications due to increased radiation.

Error-correcting memory controllers traditionally use Hamming codes, although some use triple modular redundancy.

Interleaving allows distributing the effect of a single cosmic ray potentially upsetting multiple physically neighboring bits across multiple words by associating neighboring bits to different words. As long as a single event upset (SEU) does not exceed the error threshold (e.g., a single error) in any particular word between accesses, it can be corrected (e.g., by a single-bit error correcting code), and the illusion of an error-free memory system may be maintained.

Chapter-2

Coding Theory

Coding theory is the study of the properties of codes and their fitness for a specific application. Codes are used for data compression, cryptography, error-correction and more recently also for network coding. Codes are studied by various scientific disciplines—such as information theory, electrical engineering, mathematics, and computer science—for the purpose of designing efficient and reliable data transmission methods. This typically involves the removal of redundancy and the correction (or detection) of errors in the transmitted data.

There are essentially two aspects to Coding theory:

1. Data compression (or, *source coding*)
2. Error correction (or, *channel coding*).

These two aspects may be studied in combination. Source encoding, attempts to compress the data from a source in order to transmit it more efficiently. This practice is found every day on the Internet where the common "Zip" data compression is used to reduce the network load and make files smaller. The second, channel encoding, adds extra data bits to make the transmission of data more robust to disturbances present on the transmission channel. The ordinary user may not be aware of many applications using channel coding. A typical music CD uses the Reed-Solomon code to correct for scratches and dust. In this application the transmission channel is the CD itself. Cell phones also use coding techniques to correct for the fading and noise of high frequency radio transmission. Data modems, telephone transmissions, and NASA all employ channel coding techniques to get the bits through, for example the turbo code and LDPC codes.

Source coding

The aim of source coding is to take the source data and make it smaller.

Principle

Entropy of a source is the measure of information. Basically source codes try to reduce the redundancy present in the source, and represent the source with fewer bits that carry more information.

Data compression which explicitly tries to minimize the average length of messages according to a particular assumed probability model is called entropy encoding.

Various techniques used by source coding schemes try to achieve the limit of Entropy of the source. $C(x) \geq H(x)$, where $H(x)$ is entropy of source (bitrate), and $C(x)$ is the bitrate after compression. In particular, no source coding scheme can be better than the entropy of the source.

Example

Facsimile transmission uses a simple run length code. Source coding includes also removal of all data that superfluous the need of transmitter, this decreases the bandwidth required for the transmission process.

Channel coding

The aim of channel coding theory is to find codes which transmit quickly, contain many valid code words and can correct or at least detect many errors. While not mutually exclusive, performance in these areas is a trade off. So, different codes are optimal for different applications. The needed properties of this code mainly depend on the probability of errors happening during transmission. In a typical CD, the impairment is mainly dust or scratches. Thus codes are used in an interleaved manner. The data is spread out over the disk. Although not a very good code, a simple repeat code can serve as an understandable example. Suppose we take a block of data bits (representing sound) and send it three times. At the receiver we will examine the three repetitions bit by bit and take a majority vote. The twist on this is that we don't merely send the bits in order. We interleave them. The block of data bits is first divided into 4 smaller blocks. Then we cycle through the block and send one bit from the first, then the second, etc. This is done three times to spread the data out over the surface of the disk. In the context of the simple repeat code, this may not appear effective. However, there are more powerful codes known which are very effective at correcting the "burst" error of a scratch or a dust spot when this interleaving technique is used.

Other codes are more appropriate for different applications. Deep space communications are limited by the thermal noise of the receiver which is more of a continuous nature than a bursty nature. Likewise, narrowband modems are limited by the noise, present in the telephone network and also modeled better as a continuous disturbance. Cell phones are subject to rapid fading. The high frequencies used can cause rapid fading of the signal even if the receiver is moved a few inches. Again there are a class of channel codes that are designed to combat fading.

Linear codes

The term **algebraic coding theory** denotes the sub-field of coding theory where the properties of codes are expressed in algebraic terms and then further researched.

Algebraic coding theory is basically divided into two major types of codes:

1. Linear block codes
2. Convolutional codes.

It analyzes the following three properties of a code – mainly:

- code word length
- total number of valid code words
- the minimum distance between two valid code words, using mainly the Hamming distance, sometimes also other distances like the Lee distance.

Linear block codes

Linear block codes have the property of linearity, i.e the sum of any two codewords is also a code word, and they are applied to the source bits in blocks, hence the name linear block codes. There are block codes that are not linear, but it is difficult to prove that a code is a good one without this property.

Linear block codes are summarized by their symbol alphabets (e.g., binary or ternary) and parameters (n,m,d_{min}) where

1. n is the length of the codeword, in symbols,
2. m is the number of source symbols that will be used for encoding at once,
3. d_{min} is the minimum hamming distance for the code.

There are many types of linear block codes, such as

1. Cyclic codes (e.g., Hamming codes)
2. Repetition codes
3. Parity codes
4. Polynomial codes (e.g., BCH codes)
5. Reed–Solomon codes
6. Algebraic geometric codes
7. Reed–Muller codes
8. Perfect codes.

Block codes are tied to the sphere packing problem, which has received some attention over the years. In two dimensions, it is easy to visualize. Take a bunch of pennies flat on the table and push them together. The result is a hexagon pattern like a bee's nest. But block codes rely on more dimensions which cannot easily be visualized. The powerful

(24,12) Golay code used in deep space communications uses 24 dimensions. If used as a binary code (which it usually is) the dimensions refer to the length of the codeword as defined above.

The theory of coding uses the *N*-dimensional sphere model. For example, how many pennies can be packed into a circle on a tabletop, or in 3 dimensions, how many marbles can be packed into a globe. Other considerations enter the choice of a code. For example, hexagon packing into the constraint of a rectangular box will leave empty space at the corners. As the dimensions get larger, the percentage of empty space grows smaller. But at certain dimensions, the packing uses all the space and these codes are the so-called "perfect" codes. The only nontrivial and useful perfect codes are the distance-3 Hamming codes with parameters satisfying $(2^r - 1, 2^r - 1 - r, 3)$, and the [23,12,7] binary and [11,6,5] ternary Golay codes.

Another code property is the number of neighbors that a single codeword may have. Again, consider pennies as an example. First we pack the pennies in a rectangular grid. Each penny will have 4 near neighbors (and 4 at the corners which are farther away). In a hexagon, each penny will have 6 near neighbors. When we increase the dimensions, the number of near neighbors increases very rapidly. The result is the number of ways for noise to make the receiver choose a neighbor (hence an error) grows as well. This is a fundamental limitation of block codes, and indeed all codes. It may be harder to cause an error to a single neighbor, but the number of neighbors can be large enough so the total error probability actually suffers.

Properties of linear block codes are used in many applications. For example, the syndrome-coset uniqueness property of linear block codes is used in trellis shaping, one of the best known shaping codes. This same property is used in sensor networks for distributed source coding

Convolutional codes

The idea behind a convolutional code is to make every codeword symbol be the weighted sum of the various input message symbols. This is like convolution used in LTI systems to find the output of a system, when you know the input and impulse response.

So we generally find the output of the system convolutional encoder, which is the convolution of the input bit, against the states of the convolution encoder, registers.

Fundamentally, convolutional codes do not offer more protection against noise than an equivalent block code. In many cases, they generally offer greater simplicity of implementation over a block code of equal power. The encoder is usually a simple circuit which has state memory and some feedback logic, normally XOR gates. The decoder can be implemented in software or firmware.

The Viterbi algorithm is the optimum algorithm used to decode convolutional codes. There are simplifications to reduce the computational load. They rely on searching only

the most likely paths. Although not optimum, they have generally found to give good results in the lower noise environments.

Convolutional codes are used in voiceband modems (V.32, V.17, V.34) and in GSM mobile phones, as well as satellite and military communication devices.

Other applications of coding theory

Another concern of coding theory is designing codes that help synchronization. A code may be designed so that a phase shift can be easily detected and corrected and that multiple signals can be sent on the same channel.

Another application of codes, used in some mobile phone systems, is code-division multiple access (CDMA). Each phone is assigned a code sequence that is approximately uncorrelated with the codes of other phones. When transmitting, the code word is used to modulate the data bits representing the voice message. At the receiver, a demodulation process is performed to recover the data. The properties of this class of codes allow many users (with different codes) to use the same radio channel at the same time. To the receiver, the signals of other users will appear to the demodulator only as a low-level noise.

Another general class of codes are the automatic repeat-request (ARQ) codes. In these codes the sender adds redundancy to each message for error checking, usually by adding check bits. If the check bits are not consistent with the rest of the message when it arrives, the receiver will ask the sender to retransmit the message. All but the simplest wide area network protocols use ARQ. Common protocols include SDLC (IBM), TCP (Internet), X.25 (International) and many others. There is an extensive field of research on this topic because of the problem of matching a rejected packet against a new packet. Is it a new one or is it a retransmission? Typically numbering schemes are used, as in TCP."RFC793". *RFCs*. Internet Engineering Task Force (IETF).

Group Testing

Group testing uses codes in a different way. Consider a large group of items in which a very few are different in a particular way (for eg. Defective products or infected test subjects). The idea of group testing is to determine which items are "different" by using as few tests as possible. The origin of the problem has its roots in the Second World War when the United States Army Air Forces needed to test its soldiers for Syphilis. It originated from a ground-breaking paper by Robert Dorfman.

Analog coding

Information is encoded analogously in the neural networks of brains, in analog signal processing, and analog electronics. Aspects of analog coding include analog error correction, analog data compression. analog encryption

Neural coding

Neural coding is a neuroscience-related field concerned with how sensory and other information is represented in the brain by networks of neurons. The main goal of studying neural coding is to characterize the relationship between the stimulus and the individual or ensemble neuronal responses and the relationship among electrical activity of the neurons in the ensemble. It is thought that neurons can encode both digital and analog information, and that neurons follow the principles of information theory and compress information, and detect and correct errors in the signals that are sent throughout the brain and wider nervous system.

Chapter-3

Convolutional Code, Concatenated Error Correction Code and Hadamard Code

Convolutional code

In telecommunication, a **convolutional code** is a type of error-correcting code in which

- each m-bit information symbol (each m-bit string) to be encoded is transformed into an n-bit symbol, where m/n is the code *rate* ($n \geq m$) and
- the transformation is a function of the last k information symbols, where k is the constraint length of the code.

Where convolutional codes are used

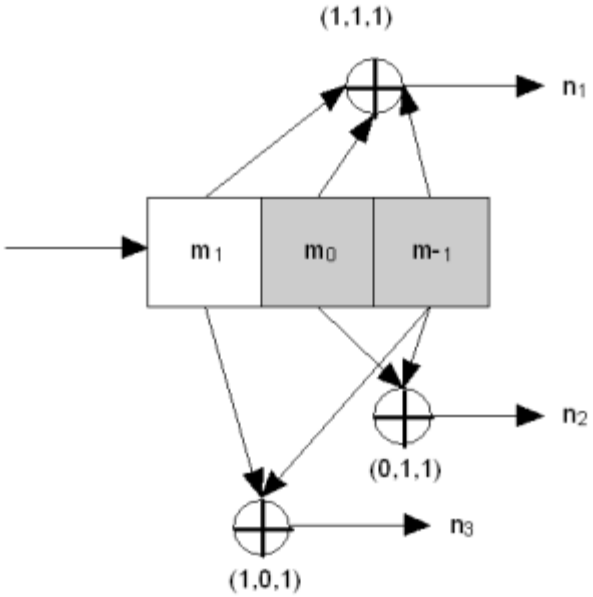
Convolutional codes are used extensively in numerous applications in order to achieve reliable data transfer, including digital video, radio, mobile communication, and satellite communication. These codes are often implemented in concatenation with a hard-decision code, particularly Reed Solomon. Prior to turbo codes, such constructions were the most efficient, coming closest to the Shannon limit.

Convolutional encoding

To convolutionally encode data, start with k memory registers, each holding 1 input bit. Unless otherwise specified, all memory registers start with a value of 0. The encoder has n modulo-2 adders (a modulo 2 adder can be implemented with a single Boolean XOR gate, where the logic is: 0+0 = 0, 0+1 = 1, 1+0 = 1, 1+1 = 0), and n generator polynomials — one for each adder (see figure below). An input bit m_1 is fed into the leftmost register. Using the generator polynomials and the existing values in the remaining registers, the encoder outputs n bits. Now bit shift all register values to the right (m_1 moves to m_0, m_0 moves to m_{-1}) and wait for the next input bit. If there are no remaining input bits, the encoder continues output until all registers have returned to the zero state.

The figure below is a rate 1/3 (m/n) encoder with constraint length (k) of 3. Generator polynomials are $G_1 = (1,1,1)$, $G_2 = (0,1,1)$, and $G_3 = (1,0,1)$. Therefore, output bits are calculated (modulo 2) as follows:

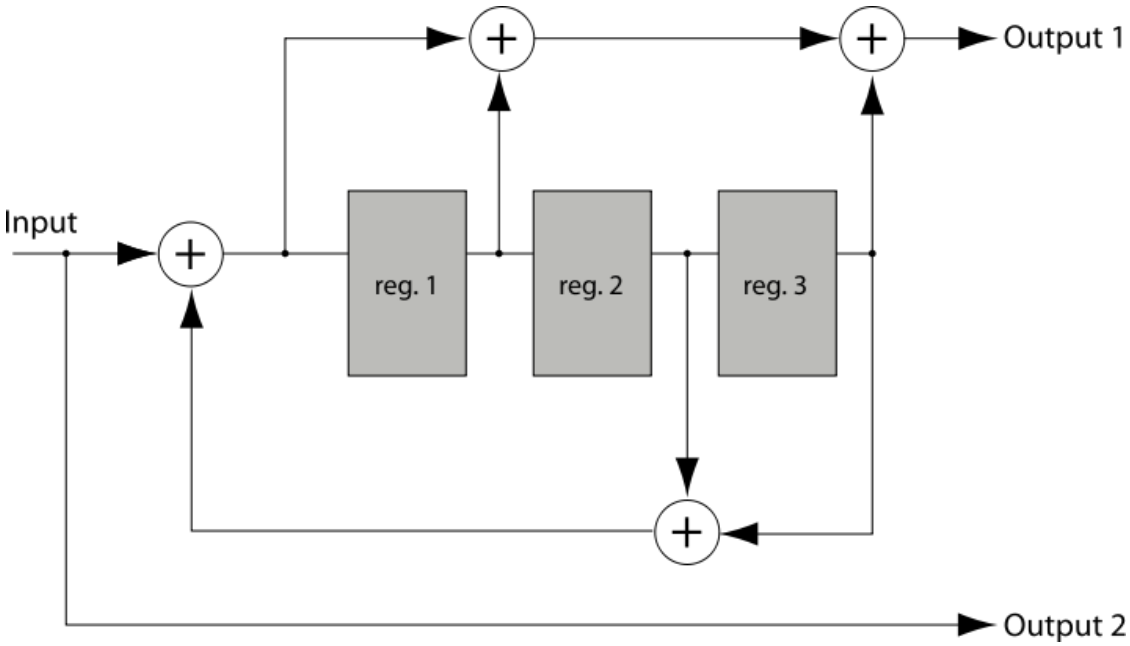
$$n_1 = m_1 + m_0 + m_{-1}$$
$$n_2 = m_0 + m_{-1}$$
$$n_3 = m_1 + m_{-1}.$$



Img.1. Rate 1/3 non-recursive, non-systematic convolutional encoder with constraint length 3

Recursive and non-recursive codes

The encoder on the picture above is a *non-recursive* encoder. Here's an example of a recursive one:



Img.2. Rate 1/2 recursive, systematic convolutional encoder with constraint length 4

One can see that the input being encoded is included in the output sequence too (look at the output 2). Such codes are referred to as *systematic*; otherwise the code is called *non-systematic*.

Recursive codes are almost always systematic and, conversely, non-recursive codes are non-systematic. It isn't a strict requirement, but a common practice.

Impulse response, transfer function, and constraint length

A convolutional encoder is called so because it performs a *convolution* of the input stream with the encoder's *impulse responses*:

$$y_i^j = \sum_{k=0}^{\infty} h_k^j x_{i-k},$$

where $x$is an input sequence, $y^j$is a sequence from output $j$and $h^j$is an impulse response for output j.

A convolutional encoder is a discrete linear time-invariant system. Every output of an encoder can be described by its own transfer function, which is closely related to a generator polynomial. An impulse response is connected with a transfer function through Z-transform.

Transfer functions for the first (non-recursive) encoder are:

- $H_1(z) = 1 + z^{-1} + z^{-2}$,
- $H_2(z) = z^{-1} + z^{-2}$,
- $H_3(z) = 1 + z^{-2}$.

Transfer functions for the second (recursive) encoder are:

- $H_1(z) = \frac{1 + z^{-1} + z^{-3}}{1 - z^{-2} - z^{-3}}$,
- $H_2(z) = 1$.

Define $m$by

$$m = \max_i polydeg(H_i(1/z))$$

where, for any rational function $f(z) = P(z)/Q(z)$,

$$polydeg(f) = \max(deg(P), deg(Q))$$.

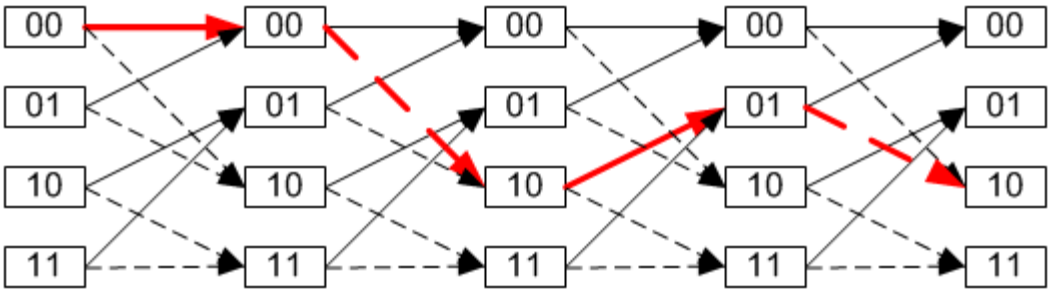
Then $m$is the maximum of the polynomial degrees of the $H_i(1/z)$, and the *constraint length* is defined as $K = m + 1$. For instance, in the first example the constraint length is 3, and in the second the constraint length is 4.

Trellis diagram

A convolutional encoder is a finite state machine. An encoder with *n* binary cells will have 2^n states.

Imagine that the encoder (shown on Img.1, above) has '1' in the left memory cell (m_0), and '0' in the right one (m_{-1}). (m_1 is not really a memory cell because it represents a current value). We will designate such a state as "10". According to an input bit the encoder at the next turn can convert either to the "01" state or the "11" state. One can see that not all transitions are possible (e.g., a decoder can't convert from "10" state to "00" or even stay in "10" state).

All possible transitions can be shown as below:



Img.3. A trellis diagram for the encoder on Img.1. A path through the trellis is shown as a red line. The solid lines indicate transitions where a "0" is input and the dashed lines where a "1" is input.

An actual encoded sequence can be represented as a path on this graph. One valid path is shown in red as an example.

This diagram gives us an idea about *decoding*: if a received sequence doesn't fit this graph, then it was received with errors, and we must choose the nearest *correct* (fitting the graph) sequence. The real decoding algorithms exploit this idea.

Free distance and error distribution

The **free distance** (*d*) is the minimal Hamming distance between different encoded sequences. The *correcting capability* (*t*) of a convolutional code is the number of errors that can be corrected by the code. It can be calculated as

$$t = \left\lfloor \frac{d-1}{2} \right\rfloor .$$

Since a convolutional code doesn't use blocks, processing instead a continuous bitstream, the value of *t* applies to a quantity of errors located relatively near to each other. That is, multiple groups of *t* errors can usually be fixed when they are relatively far apart.

Free distance can be interpreted as the minimal length of an erroneous "burst" at the output of a convolutional decoder. The fact that errors appear as "bursts" should be accounted for when designing a concatenated code with an inner convolutional code. The popular solution for this problem is to interleave data before convolutional encoding, so that the outer block (usually Reed-Solomon) code can correct most of the errors.

Decoding convolutional codes

Several algorithms exist for decoding convolutional codes. For relatively small values of *k*, the Viterbi algorithm is universally used as it provides maximum likelihood

performance and is highly parallelizable. Viterbi decoders are thus easy to implement in VLSI hardware and in software on CPUs with SIMD instruction sets.

Longer constraint length codes are more practically decoded with any of several sequential decoding algorithms, of which the Fano algorithm is the best known. Unlike Viterbi decoding, sequential decoding is not maximum likelihood but its complexity increases only slightly with constraint length, allowing the use of strong, long-constraint-length codes. Such codes were used in the Pioneer program of the early 1970s to Jupiter and Saturn, but gave way to shorter, Viterbi-decoded codes, usually concatenated with large Reed-Solomon error correction codes that steepen the overall bit-error-rate curve and produce extremely low residual undetected error rates.

Both Viterbi and sequential decoding algorithms return hard-decisions: the bits that form the most likely codeword. An approximate confidence measure can be added to each bit by use of the Soft output Viterbi algorithm. Maximum a posteriori (MAP) soft-decisions for each bit can be obtained by use of the BCJR algorithm.

Popular convolutional codes

An especially popular Viterbi-decoded convolutional code, used at least since the Voyager program has a constraint length k of 7 and a rate r of 1/2.

- Longer constraint lengths produce more powerful codes, but the complexity of the Viterbi algorithm increases exponentially with constraint lengths, limiting these more powerful codes to deep space missions where the extra performance is easily worth the increased decoder complexity.

- Mars Pathfinder, Mars Exploration Rover and the Cassini probe to Saturn use a k of 15 and a rate of 1/6; this code performs about 2 dB better than the simpler k=7 code at a cost of 256× in decoding complexity (compared to Voyager mission codes).

Punctured convolutional codes

Puncturing is a technique used to make a m/n rate code from a "basic" rate 1/2 code. It is reached by deletion of some bits in the encoder output. Bits are deleted according to *puncturing matrix*. The following puncturing matrices are the most frequently used:

Code rate	Puncturing matrix	Free distance (for NASA standard K=7 convolutional code)
1/2 (No perf.)	1 1	10
2/3	1 0 1 1	6

3/4	1 0 1 1 1 0	5
5/6	1 0 1 0 1 1 1 0 1 0	4
7/8	1 0 0 0 1 0 1 1 1 1 1 0 1 0	3

For example, if we want to make a code with rate 2/3 using the appropriate matrix from the above table, we should take a basic encoder output and transmit every second bit from the first branch and every bit from the second one. The specific order of transmission is defined by the respective communication standard.

Punctured convolutional codes are widely used in the satellite communications, for example, in INTELSAT systems and Digital Video Broadcasting.

Punctured convolutional codes are also called "perforated".

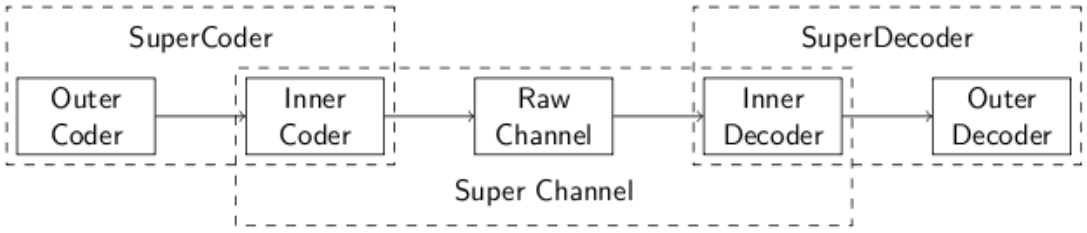
Turbo codes: replacing convolutional codes

Simple Viterbi-decoded convolutional codes are now giving way to turbo codes, a new class of iterated short convolutional codes that closely approach the theoretical limits imposed by Shannon's theorem with much less decoding complexity than the Viterbi algorithm on the long convolutional codes that would be required for the same performance. Concatenation with an outer algebraic code (e.g., Reed-Solomon) addresses the issue of error floors inherent to turbo code designs.

Concatenated error correction code

In coding theory, **concatenated codes** form a class of error-correcting codes that are derived by combining an **inner code** and an **outer code**. They were conceived in 1966 by Dave Forney as a solution for the problem of finding a code that has both exponentially decreasing error probability with increasing block length and polynomial-time decoding complexity.

Description



Let C be a code with length N and rate R over an alphabet A with $K=N*R$ symbols. Let I be another code with length n and rate r over an alphabet B with $k=n*r$ symbols.

The inner code I takes one of k possible inputs, encodes onto an n-tuple from B, transmits, and decodes into one of k possible outputs. We regard this as a (super) channel which can transmit one symbol from the alphabet A, also of size k. We use this channel N times to transmit each of the N symbols in a codeword of C. The *concatenation* of C (as outer code) with I (as inner code) is thus a code of length Nn over the alphabet B.

The *key insight* in this approach is that if I is decoded using a maximum-likelihood approach (thus showing an exponentially decreasing error probability with increasing length), and C is a code with length $N=2^{nr}$ that can be decoded in polynomial time of N, then the concatenated code can be decoded in polynomial time of its combined length $n2^{nr}$ and shows an exponentially decreasing error probability, even if I has exponential decoding complexity.

In a generalization of above concatenation, there are N possible inner codes I_i and the i-th symbol in a codeword of C is transmitted across the inner channel using the i-th inner code. The Justesen codes are examples of generalized concatenated codes, where the outer code is a Reed-Solomon code.

Applications

Concatenated codes were starting to be regularly used for deep space communication with the Voyager program, which launched their first probe in 1977. (A simple concatenation scheme however was already implemented for the 1971 Mariner Mars orbiter mission.) Since then, concatenated codes became the workhorse for efficient error correction coding, and stayed so at least until the invention of turbo codes and LDPC codes.

Typically, the inner code is not a block code but a soft-decision convolutional Viterbi-decoded code with a short constraint length. For the outer code, a longer hard-decision block code, frequently Reed Solomon with 8-bit symbols, is selected. The larger symbol size makes the outer code more robust to burst errors that may occur due to channel

impairments, and because erroneous output of the convolutional code itself is bursty. An interleaving layer is usually added between the two codes to spread burst errors across a wider range.

The combination of an inner Viterbi convolutional code with an outer Reed-Solomon code (known as an RSV code) was first used on Voyager 2, and became a popular construction both within and outside of the space sector. It is still notably used today for satellite communication, such as the DVB-S digital television broadcast standard.

In a more loose sense, any (serial) combination of two or more codes may be referred to as a concatenated code. For example, within the DVB-S2 standard, a highly efficient LDPC code is combined with an algebraic outer code in order to remove any resilient errors left over from the inner LDPC code due to its inherent error floor.

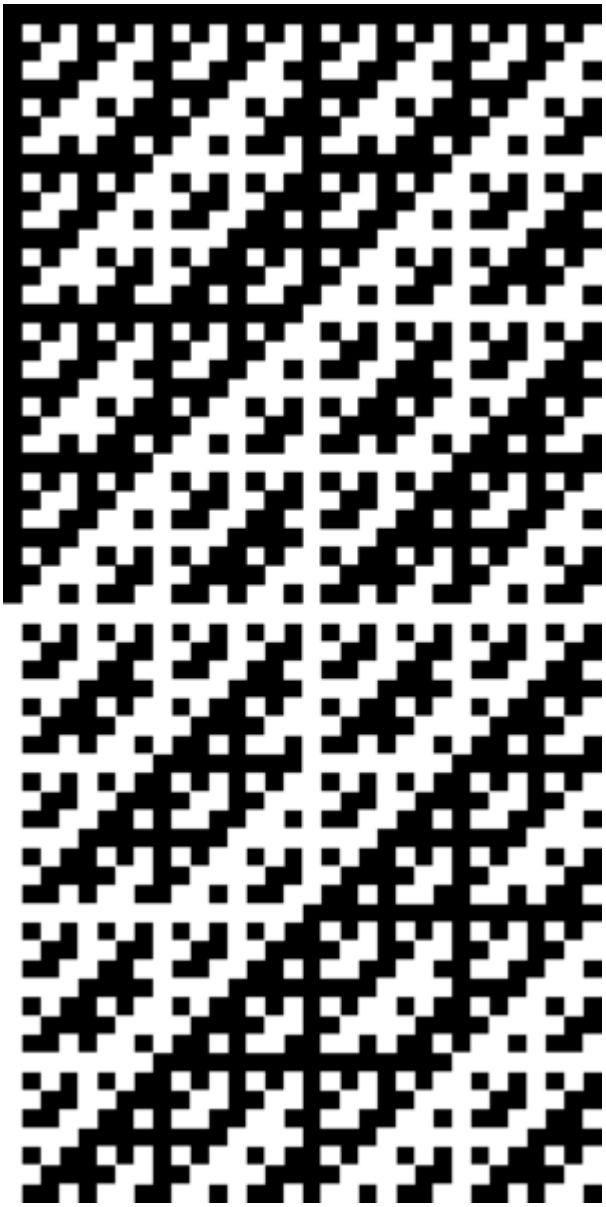
A simple concatenation scheme is also used on the Compact Disc, where an interleaving layer between two Reed-Solomon codes of different sizes effectively spreads errors across different blocks.

Turbo codes: A parallel concatenation approach

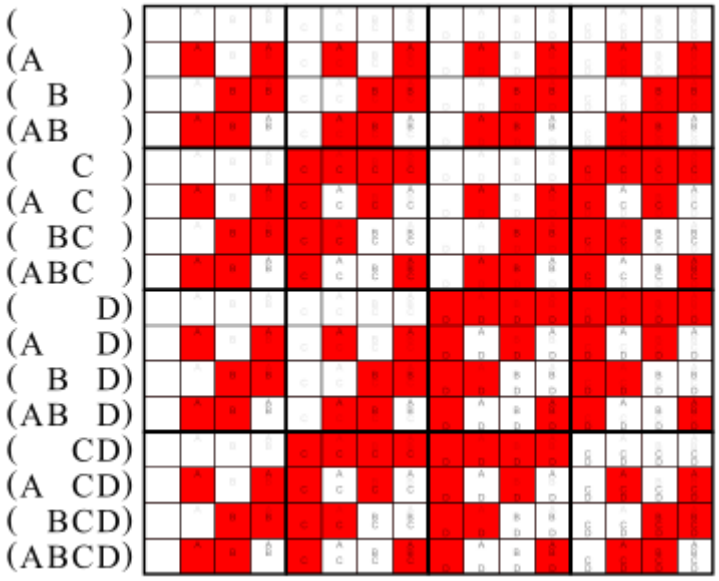
The description above is given for what is now called a serially concatenated code. Turbo codes, as described first in 1993, implemented a parallel concatenation of two convolutional codes, with an interleaver between the two codes and an iterative decoder that would pass information forth and back between the codes. This construction had much higher performance than all previously conceived concatenated codes.

However, a key aspect of turbo codes is their iterated decoding approach. Iterated decoding is now also applied to serial concatenations in order to achieve higher coding gains, such as within serially concatenated convolutional codes (SCCCs). An early form of iterated decoding was notably implemented with 2 to 5 iterations in the "Galileo code" of the Galileo spacecraft.

Hadamard code



Matrix of the Hadamard code (32, 6, 16) for the Reed-Muller Code (1, 5) of the NASA space probe Mariner 9



XOR operations Here the white fields stand for 0 and the red fields for 1

The **Hadamard code**, named after Jacques Hadamard, is a system used for signal error detection and correction. It is one of the family of $[2^n, n + 1, 2^{n-1}]$ codes. Especially for large n it has a poor rate but it is capable of correcting many errors. Hadamard codes can be considered as a special case of Reed–Muller codes. In particular, first order Reed–Muller code are equivalent to Hadamard codes.

Construction

The code is based on Hadamard matrices. If H is a Hadamard matrix of order 2^n the codewords are constructed by taking the rows of H and $-H$ as codewords, where each -1 is replaced by 0. In this way 2^{n+1} code words are constructed that each have a length of 2^n. Since the rows of a Hadamard matrix are orthogonal the minimum distance is 2^{n-1}. In this way a $[2^n, n + 1, 2^{n-1}]$ code is constructed.

The code can also be constructed by creating the parity-check matrix which consists of all 2^{n-1} vectors containing an odd number of 1s, or by using a recursive encoding process.

Decoding

The code has minimum distance 2^{n-1} and hence can correct at most $t = 2^{n-2} - 1$ errors. The algorithm below achieves this.

When a code word is received it is first transformed to a $1/-1$ vector v by changing all 0s to -1. Compute $(v\,H^{\mathrm{T}})$. The entry with the maximum absolute value corresponds to the row taken as a code word. If it is positive the code word came from H, if it is negative the code word came from $-H$.

Proof: If there were no errors the product ($v\,H^{\mathrm{T}}$) would consist of only zero's and one entry of +/−2^n. If there are errors in v then, in absolute value, some of the zeros become larger and the maximum becomes smaller. Each error that occurs can change a zero with 2. So the zeros can become at most $0 + 2t = 2^{n-1} - 2$. The maximum can decrease at most to $2^n - 2t = 2^n - 2^{n-1} + 2 = 2^{n-1} + 2$. So the maximum that points to the correct row will always be larger in absolute value than the other values in the row.

History

A Hadamard code was used during the 1971 Mariner 9 mission to correct for picture transmission errors. The data words used during this mission were 6 bits long, which represented 64 grayscale values. Because of limitations of the quality of the alignment of the transmitter the maximum useful data length was about 30 bits. Instead of using a repetition code, a [32, 6, 16] Hadamard code was used. Errors of up to 7 bits per word could be corrected using this scheme. Compared to a 5-repetition code, the error correcting properties of this Hadamard code are much better, yet its rate is comparable. The efficient decoding algorithm was an important factor in the decision to use this code. The circuitry used was called the "Green Machine". It employed the fast Fourier transform which can increase the decoding speed by a factor of 3.

Optimality

For value of $n \leq 6$ the Hadamard codes have been proven optimal in the sense of minimum distance.

Chapter-4

Check Digit and Coding Gain

Check digit

A **check digit** is a form of redundancy check used for error detection, the decimal equivalent of a binary checksum. It consists of a single digit computed from the other digits in the message.

With a check digit, one can detect simple errors in the input of a series of digits, such as a single mistyped digit or some permutations of two successive digits.

Design

Check digit algorithms are generally designed to capture *human* transcription errors. In order of complexity, these include the following:

- single digit errors, such as 1 → 2
- transposition errors, such as 12 → 21
- twin errors, such as 11 → 22
- jump transpositions errors, such as 132 → 231
- jump twin errors, such as 131 → 232
- phonetic errors, such as 60 → 16 ("sixty" to "sixteen")

In choosing a system, a high probability of catching errors is traded off against implementation difficulty; simple check digit systems are easily understood and implemented by humans but do not catch as many errors as complex ones, which require sophisticated programs to implement.

A desirable feature is that left-padding with zeros should not change the check digit. This allows variable length digits to be used and the length to be changed.

If there is a single check digit added to the original number, the system will not always capture *multiple* errors, such as two replacement errors (12 → 34) though, typically, double errors will be caught 90% of the time (both changes would need to change the output by offsetting amounts).

A very simple check digit method would be to take the sum of all digits (digital sum) modulo 10. This would catch any single-digit error, as such an error would always change the sum, but does not catch any transposition errors (switching two digits) as re-ordering does not change the sum.

A slightly more complex method is to take the weighted sum of the digits, modulo 10, with different weights for each number position.

To illustrate this, for example if the weights for a four digit number were 5, 3, 2, 7 and the number to be coded was 4871, then one would take 5×4 + 3×8 + 2×7 + 7×1 = 65, ie 5 modulo 10, and the check digit would be 5, giving 48715.

Systems with weights of 1, 3, 7, or 9, with the weights on neighboring numbers being different, are widely used: for example, 31 31 weights in UPC codes, 13 13 weights in EAN numbers (GS1 algorithm), and the 371 371 371 weights used in United States bank routing transit numbers. This system detects all single-digit errors and around 90% of transposition errors. 1, 3, 7, and 9 are used because they are coprime to 10, so changing any digit changes the check digit; using a coefficient that is divisible by 2 or 5 would lose information (because $5 \cdot 0 = 5 \cdot 2 = 5 \cdot 4 = 5 \cdot 6 = 5 \cdot 8 = 0 \mod 10$) and thus not catch some single-digit errors. Using different weights on neighboring numbers means that most transpositions change the check digit; however, because all weights differ by an even number, this does not catch transpositions of two digits that differ by 5, (0 and 5, 1 and 6, 2 and 7, 3 and 8, 4 and 9), since the 2 and 5 multiply to yield 10.

The ISBN-10 code instead uses modulo 11, which is prime, and all the number positions have different weights $1, 2, \ldots, 10$. This system thus detects all single digit substitution and transposition errors (including jump transpositions), but at the cost of the check digit possibly being 10, represented by "X". (An alternative is simply to avoid using the serial numbers which result in an "X" check digit.) ISBN-13 instead uses the GS1 algorithm used in EAN numbers.

More complicated algorithms include the Luhn algorithm (1954), which captures 98% of single digit transposition errors (it does not detect 90 ↔ 09), while more sophisticated is the Verhoeff algorithm (1969), which catches all single digit substitution and transposition errors, and many (but not all) more complex errors. Both these methods use a single check digit and will therefore fail to capture around 10% of more complex errors. To reduce this failure rate, it is necessary to use more than one check digit (for example, the modulo 97 check referred to below, which uses two check digits) and/or to use a wider range of characters in the check digit, for example letters plus numbers).

Examples

UPC

The final digit of a Universal Product Code is a check digit computed as follows:

1. Add the digits (up to but not including the check digit) in the odd-numbered positions (first, third, fifth, etc.) together and multiply by three.
2. Add the digits (up to but not including the check digit) in the even-numbered positions (second, fourth, sixth, etc.) to the result.
3. Take the remainder of the result divided by 10 (modulo operation) and subtract this from 10 to derive the check digit.

For instance, the UPC-A barcode for a box of tissues is "036000241457". The last digit is the check digit "7", and if the other numbers are correct then the check digit calculation must produce 7.

1. Add the odd number digits: 0+6+0+2+1+5 = 14
2. Multiply the result by 3: 14 × 3 = 42
3. Add the even number digits: 3+0+0+4+4 = 11
4. Add the two results together: 42 + 11 = 53
5. To calculate the check digit, take the remainder of (53 / 10), which is also known as (53 modulo 10), and subtract from 10. Therefore, the check digit value is 7.

Another example: to calculate the check digit for the following food item "01010101010".

1. Add the odd number digits: 0+0+0+0+0+0 = 0
2. Multiply the result by 3: 0 x 3 = 0
3. Add the even number digits: 1+1+1+1+1 = 5
4. Add the two results together: 0 + 5 = 5
5. To calculate the check digit, take the remainder of (5 / 10), which is also known as (5 modulo 10), and subtract from 10 (i.e. (10 - 5 modulo 10) = 5. Therefore, the check digit value is 5.
6. If the remainder is 0, subtracting from 10 would give 10. In that case, use 0 as the check digit.

ISBN 10

The final character of a ten digit International Standard Book Number is a check digit computed so that multiplying each digit by its position in the number (counting from the right) and taking the sum of these products modulo 11 is 0. The digit the farthest to the right (which is multiplied by 1) is the check digit, chosen to make the sum correct. It may need to have the value 10, which is represented as the letter X. For example, take the ISBN 0-201-53082-1. The sum of products is 0×10 + 2×9 + 0×8 + 1×7 + 5×6 + 3×5 + 0×4 + 8×3 + 2×2 + 1×1 = 99 ≡ 0 modulo 11. So the ISBN is valid.

While this may seem more complicated than the first scheme, it can be validated simply by adding all the products together then dividing by 11. The sum can be computed without any multiplications by initializing two variables, `t` and `sum`, to 0 and repeatedly performing `t = t + digit; sum = sum + t;` (which can be expressed in C as `sum += t += digit;`). If the final `sum` is a multiple of 11, the ISBN is valid.

ISBN 13

ISBN 13 (in use January 2007) is equal to the EAN-13 code found underneath a book's barcode. Its check digit is generated the same way as the UPC except that the even digits are multiplied by 3 instead of the odd digits.

EAN (GLN,GTIN, EAN numbers administered by GS1)

EAN (European Article Number) check digits (administered by GS1) are calculated by summing the even position numbers and multiplying by 3 and then by adding the sum of the odd position numbers. The final digit of the result is subtracted from 10 to calculate the check digit (or left as is if already zero). A GS1 check digit calculator and detailed documentation is online at GS1's website.

Other examples of check digits

- The tenth digit of the National Provider Identifier for the US healthcare industry
- The Australian Tax File Number (based on modulo 11)
- The Guatemalan Tax Number (NIT - Número de Identificación Tributaria) based on modulo 11
- The North American CUSIP number
- The final (ninth) digit of the routing transit number, a bank code used in the United States
- The International SEDOL number
- The International Securities Identifying Number (ISIN)
- The International CAS registry number's final digit.
- Modulo 10 check digits in credit card account numbers, calculated with the Luhn algorithm.
    - Also used in the Norwegian KID (customer identification number) numbers used in bank giros (credit transfer).
- The final character encoded in a magnetic stripe card is a computed Longitudinal redundancy check
- final digit of a POSTNET code

Coding gain

In coding theory and related engineering problems, **coding gain** is the measure in the difference between the signal to noise ratio (SNR) levels between the uncoded system and coded system required to reach the same bit error rate (BER) levels when used with the error correcting code (ECC).

Example

If the uncoded BPSK system in AWGN environment has a Bit error rate (BER) of 10^{-2} at the SNR level 4dB, and the corresponding coded (*e.g.*, BCH) system has the same BER at an SNR level of 2.5dB, then we say the *coding gain* = 4dB-2.5dB = 1.5dB, due to the code used (in this case BCH).

Power-limited regime

In the *power-limited regime* (where the nominal spectral efficiency $\rho \leq 2$[b/2D or b/s/Hz], *i.e.* the domain of binary signaling), the effective coding gain $\gamma_{eff}(A)$ of a signal set A at a given target error probability per bit $P_b(E)$ is defined as the difference in dB between the E_b / N_0 required to achieve the target $P_b(E)$ with A and the E_b / N_0 required to achieve the target $P_b(E)$ with 2-PAM or (2×2)-QAM (*i.e.* no coding). The nominal coding gain $\gamma_c(A)$ is defined as

$$\gamma_c(A) = \frac{d^2_{\min}(A)}{4E_b}.$$

This definition is normalized so that $\gamma_c(A) = 1$ for 2-PAM or (2×2)-QAM. If the average number of nearest neighbors per transmitted bit $K_b(A)$ is equal to one, the effective coding gain $\gamma_{eff}(A)$ is approximately equal to the nominal coding gain $\gamma_c(A)$. However, if $K_b(A) > 1$, the effective coding gain $\gamma_{eff}(A)$ is less than the nominal coding gain $\gamma_c(A)$ by an amount which depends on the steepness of the $P_b(E)$ *vs.* E_b / N_0 curve at the target $P_b(E)$. This curve can be plotted using the union bound estimate (UBE)

$$P_b(E) \approx K_b(A) Q\sqrt{(2\gamma_c(A)E_b/N_0)},$$

where $Q(\cdot)$ denotes the Gaussian probability of error function.

For the special case of a binary linear block code C with parameters (n,k,d), the nominal spectral efficiency is $\rho = 2k / n$ and the nominal coding gain is kd/n.

Example

The table below lists the nominal spectral efficiency, nominal coding gain and effective coding gain at $P_b(E) \approx 10^{-5}$ for Reed-Muller codes of length $n \leq 64$:

Code	ρ	γ_c	γ_c (dB)	K_b	γ_{eff} (dB)
[8,7,2]	1.75	7/4	2.43	4	2.0
[8,4,4]	1.0	2	3.01	4	2.6
[16,15,2]	1.88	15/8	2.73	8	2.1
[16,11,4]	1.38	11/4	4.39	13	3.7
[16,5,8]	0.63	5/2	3.98	6	3.5
[32,31,2]	1.94	31/16	2.87	16	2.1
[32,26,4]	1.63	13/4	5.12	48	4.0
[32,16,8]	1.00	4	6.02	39	4.9
[32,6,16]	0.37	3	4.77	10	4.2
[64,63,2]	1.97	63/32	2.94	32	1.9
[64,57,4]	1.78	57/16	5.52	183	4.0
[64,42,8]	1.31	21/4	7.20	266	5.6
[64,22,16]	0.69	11/2	7.40	118	6.0
[64,7,32]	0.22	7/2	5.44	18	4.6

Bandwidth-limited regime

In the *bandwidth-limited regime* ($\rho > 2b / 2D$, *i.e.* the domain of non-binary signaling), the effective coding gain $\gamma_{eff}(A)$ of a signal set A at a given target error rate $P_s(E)$ is defined as the difference in dB between the SNR_{norm} required to achieve the target $P_s(E)$ with A and the SNR_{norm} required to achieve the target $P_s(E)$ with M-PAM or (M×M)-QAM (*i.e.* no coding). The nominal coding gain $\gamma_c(A)$ is defined as

$$\gamma_c(A) = \frac{(2^\rho - 1) d^2_{\min}(A)}{6E_s}.$$

This definition is normalized so that $\gamma_c(A) = 1$ for M-PAM or (M×M)-QAM. The UBE becomes

$$P_s(E) \approx K_s(A) Q\sqrt{(3\gamma_c(A) SNR_{norm})},$$

where $K_s(A)$ is the average number of nearest neighbors per two dimensions.

Chapter-5

Hamming Code

In telecommunication, a **Hamming code** is a linear error-correcting code named after its inventor, Richard Hamming. Hamming codes can detect up to two simultaneous bit errors, and correct single-bit errors; thus, reliable communication is possible when the Hamming distance between the transmitted and received bit patterns is less than or equal to one. By contrast, the simple parity code cannot correct errors, and can only detect an odd number of errors.

In mathematical terms, Hamming codes are a class of binary linear codes. For each integer $m \geq 2$there is a code with *m* parity bits and $2^m - m - 1$ data bits. The parity-check matrix of a Hamming code is constructed by listing all columns of length *m* that are pairwise independent. Hamming codes are an example of perfect codes, codes that exactly match the theoretical upper bound on the number of distinct code words for a given number of bits and ability to correct errors.

Because of the simplicity of Hamming codes, they are widely used in computer memory (RAM). In particular, a single-error-correcting *and* double-error-detecting variant commonly referred to as **SECDED**.

History

Hamming worked at Bell Labs in the 1940s on the Bell Model V computer, an electromechanical relay-based machine with cycle times in seconds. Input was fed in on punched cards, which would invariably have read errors. During weekdays, special code would find errors and flash lights so the operators could correct the problem. During after-hours periods and on weekends, when there were no operators, the machine simply moved on to the next job.

Hamming worked on weekends, and grew increasingly frustrated with having to restart his programs from scratch due to the unreliability of the card reader. Over the next few

years he worked on the problem of error-correction, developing an increasingly powerful array of algorithms. In 1950 he published what is now known as Hamming Code, which remains in use today in applications such as ECC memory.

Codes predating Hamming

A number of simple error-detecting codes were used before Hamming codes, but none were as effective as Hamming codes in the same overhead of space.

Parity

Parity adds a single bit that indicates whether the number of 1 bits in the preceding data was even or odd. If an odd number of bits is changed in transmission, the message will change parity and the error can be detected at this point. (Note that the bit that changed may have been the parity bit itself!) The most common convention is that a parity value of **1** indicates that there is an **odd** number of ones in the data, and a parity value of **0** indicates that there is an **even** number of ones in the data. In other words: the data and the parity bit **together** should contain an even number of 1s.

Parity checking is not very robust, since if the number of bits changed is even, the check bit will be valid and the error will not be detected. Moreover, parity does not indicate which bit contained the error, even when it can detect it. The data must be discarded entirely and re-transmitted from scratch. On a noisy transmission medium, a successful transmission could take a long time or may never occur. However, while the quality of parity checking is poor, since it uses only a single bit, this method results in the least overhead. Furthermore, parity checking does allow for the restoration of an erroneous bit when its position is known.

Two-out-of-five code

A two-out-of-five code is an encoding scheme which uses five digits consisting of exactly three 0s and two 1s. This provides ten possible combinations, enough to represent the digits 0 - 9. This scheme can detect all single bit-errors and all odd numbered bit-errors. However it still cannot correct for these errors.

Repetition

Another code in use at the time repeated every data bit several times in order to ensure that it got through. For instance, if the data bit to be sent was a 1, an n=3 *repetition code* would send "111". If the three bits received were not identical, an error occurred. If the channel is clean enough, most of the time only one bit will change in each triple. Therefore, 001, 010, and 100 each correspond to a 0 bit, while 110, 101, and 011 correspond to a 1 bit, as though the bits counted as "votes" towards what the original bit was. A code with this ability to reconstruct the original message in the presence of errors is known as an *error-correcting* code. This triple repetition code is actually the simplest Hamming code with $m = 2$, since there are 2 parity bits, and $2^2 - 2 - 1 = 1$ data bit.

Such codes cannot correctly repair all errors, however. In our example, if the channel flipped two bits and the receiver got "001", the system would detect the error, but conclude that the original bit was 0, which is incorrect. If we increase the number of times we duplicate each bit to four, we can detect all two-bit errors but can't correct them (the votes "tie"); at five, we can correct all two-bit errors, but not all three-bit errors.

Moreover, the repetition code is extremely inefficient, reducing throughput by three times in our original case, and the efficiency drops drastically as we increase the number of times each bit is duplicated in order to detect and correct more errors.

Hamming codes

If more error-correcting bits are included with a message, and if those bits can be arranged such that different incorrect bits produce different error results, then bad bits could be identified. In a 7-bit message, there are seven possible single bit errors, so three error control bits could potentially specify not only that an error occurred but also which bit caused the error.

Hamming studied the existing coding schemes, including two-of-five, and generalized their concepts. To start with, he developed a nomenclature to describe the system, including the number of data bits and error-correction bits in a block. For instance, parity includes a single bit for any data word, so assuming ASCII words with 7-bits, Hamming described this as an *(8,7)* code, with eight bits in total, of which 7 are data. The repetition example would be *(3,1)*, following the same logic. The code rate is the second number divided by the first, for our repetition example, 1/3.

Hamming also noticed the problems with flipping two or more bits, and described this as the "distance" (it is now called the *Hamming distance*, after him). Parity has a distance of 2, as any two bit flips will be invisible. The (3,1) repetition has a distance of 3, as three bits need to be flipped in the same triple to obtain another code word with no visible errors. A (4,1) repetition (each bit is repeated four times) has a distance of 4, so flipping two bits can be detected, but not corrected. When three bits flip in the same group there can be situations where the code corrects towards the wrong code word.

Hamming was interested in two problems at once; increasing the distance as much as possible, while at the same time increasing the code rate as much as possible. During the 1940s he developed several encoding schemes that were dramatic improvements on existing codes. The key to all of his systems was to have the parity bits overlap, such that they managed to check each other as well as the data.

General algorithm

The following general algorithm generates a single-error correcting (SEC) code for any number of bits.

1. Number the bits starting from 1: bit 1, 2, 3, 4, 5, etc.

2. Write the bit numbers in binary. 1, 10, 11, 100, 101, etc.
3. All bit positions that are powers of two (have only one 1 bit in the binary form of their position) are parity bits.
4. All other bit positions, with two or more 1 bits in the binary form of their position, are data bits.
5. Each data bit is included in a unique set of 2 or more parity bits, as determined by the binary form of its bit position.
    1. Parity bit 1 covers all bit positions which have the least significant bit set: bit 1 (the parity bit itself), 3, 5, 7, 9, etc.
    2. Parity bit 2 covers all bit positions which have the second least significant bit set: bit 2 (the parity bit itself), 3, 6, 7, 10, 11, etc.
    3. Parity bit 4 covers all bit positions which have the third least significant bit set: bits 4–7, 12–15, 20–23, etc.
    4. Parity bit 8 covers all bit positions which have the fourth least significant bit set: bits 8–15, 24–31, 40–47, etc.
    5. In general each parity bit covers all bits where the binary AND of the parity position and the bit position is non-zero.

The form of the parity is irrelevant. Even parity is simpler from the perspective of theoretical mathematics, but there is no difference in practice.

This general rule can be shown visually:

Bit position		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	
Encoded data bits		**p1**	**p2**	**d1**	**p4**	**d2**	**d3**	**d4**	**p8**	**d5**	**d6**	**d7**	**d8**	**d9**	**d10**	**d11**	**p16**	**d12**	**d13**	**d14**	**d15**	
Parity bit coverage	**p1**	X		X		X		X		X		X		X		X		X		X		
	p2		X	X			X	X			X	X			X	X			X	X		...
	p4				X	X	X	X					X	X	X	X					X	
	p8								X	X	X	X	X	X	X	X						
	p16																X	X	X	X	X	

Shown are only 20 encoded bits (5 parity, 15 data) but the pattern continues indefinitely. The key thing about Hamming Codes that can be seen from visual inspection is that any given bit is included in a unique set of parity bits. To check for errors, check all of the parity bits. The pattern of errors, called the error syndrome, identifies the bit in error. If all parity bits are correct, there is no error. Otherwise, the sum of the positions of the erroneous parity bits identifies the erroneous bit. For example, if the parity bits in positions 1, 2 and 8 indicate an error, then bit 1+2+8=11 is in error. If only one parity bit indicates an error, the parity bit itself is in error.

If, in addition, an overall parity bit (bit 0) is included, the code can detect (but not correct) any two-bit error, making a SECDED code. The overall parity indicates whether the total number of errors is even or odd. If the basic Hamming code detects an error, but

the overall parity says that there are an even number of errors, an uncorrectable 2-bit error has occurred.

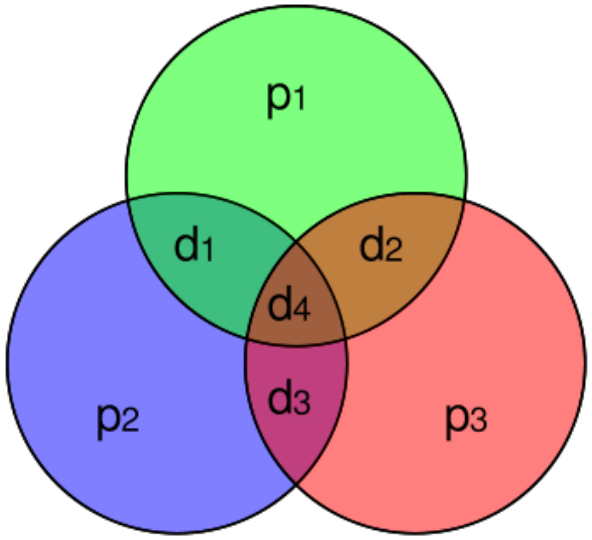
Hamming codes with additional parity (SECDED)

These codes have a minimum distance of 3, which means that the code can detect and correct a single error, but a double bit error is indistinguishable from a different code with a single bit error. Thus, they can detect double-bit errors only if correction is not attempted.

By including an extra parity bit, it is possible to increase the minimum distance of the Hamming code to 4. This gives the code the ability to detect and correct a single error and at the same time detect (but not correct) a double error. (It could also be used to detect up to 3 errors but not correct any.)

This code system is popular in computer memory systems, where it is known as SECDED ("single error correction, double error detection"). Particularly popular is the (72,64) code, a truncated (127,120) Hamming code plus an additional parity bit, which has the same space overhead as a (9,8) parity code.

Hamming(7,4) code



Graphical depiction of the 4 data bits and 3 parity bits and which parity bits apply to which data bits

In 1950, Hamming introduced the (7,4) code. It encodes 4 data bits into 7 bits by adding three parity bits. Hamming(7,4) can detect and correct single-bit errors. With the addition of an overall parity bit, it can also detect (but not correct) double-bit errors.

Construction of G and H

The matrix $\mathbf{G} := \begin{pmatrix} I_k | -A^T \end{pmatrix}$ is called a (Canonical) generator matrix of a linear (n,k) code,

and $\mathbf{H} := \begin{pmatrix} A | I_{n-k} \end{pmatrix}$ is called a parity-check matrix.

This is the construction of **G** and **H** in standard (or systematic) form. Regardless of form, **G** and **H** for linear block codes must satisfy

$\mathbf{H}\,\mathbf{G}^T = \mathbf{0}$, an all-zeros matrix [Moon, p. 89].

Since (7,4,3)=(n,k,d)=[$2^m - 1$, 2^m−1-m, m]. The parity-check matrix **H** of a Hamming code is constructed by listing all columns of length m that are pair-wise independent.

Thus **H** is a matrix whose left side is all of the nonzero n-tuples where order of the n-tuples in the columns of matrix does not matter. The right hand side is just the (n-k)-identity matrix.

So **G** can be obtained from **H** by taking the transpose of the left hand side of **H** with the identity k-identity matrix on the left hand side of **G**.

The code generator matrix $\mathbf{G}$ and the parity-check matrix $\mathbf{H}$ are:

$$\mathbf{H} := \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}_{3,7}$$

and

$$\mathbf{G} := \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}_{4,7}.$$

Finally, these matrices can be mutated into equivalent non-systematic codes by the following operations [Moon, p. 85]:

- Column permutations (swapping columns)
- Elementary row operations (replacing a row with a linear combination of rows)

Encoding

Example

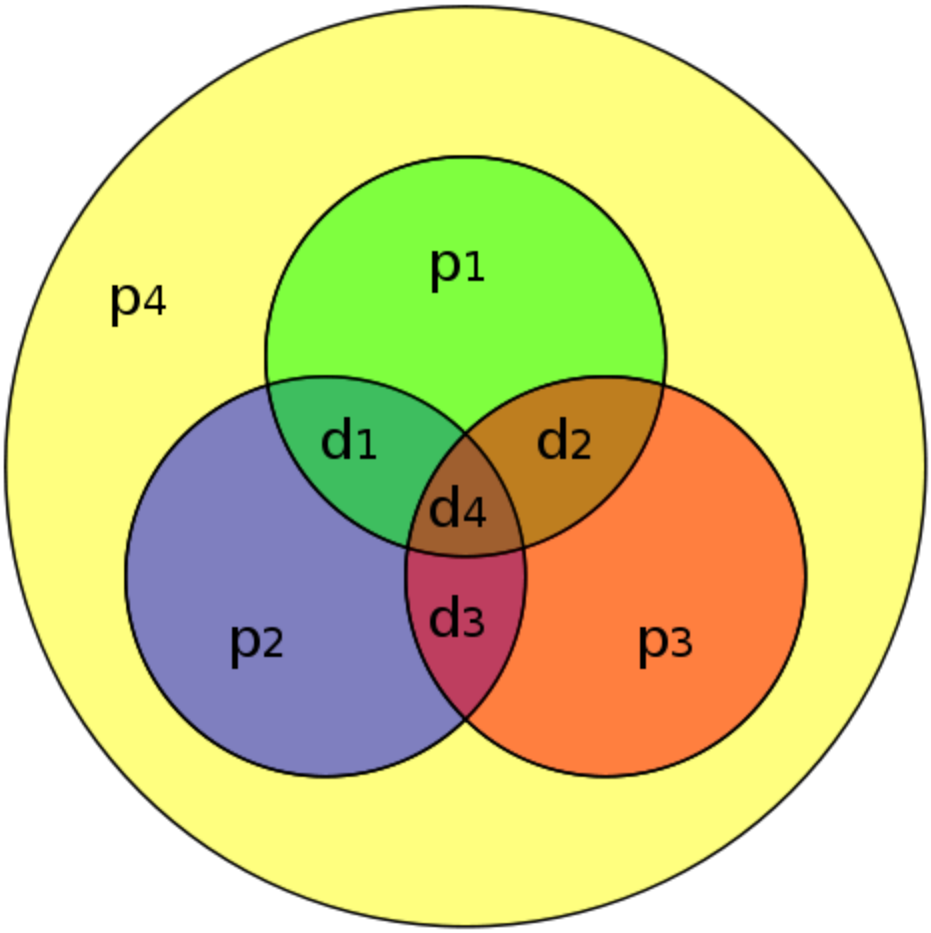
From the above matrix we have $2^k=2^4=16$ codewords. The codewords $\overrightarrow{x}$ of this binary code can be obtained from $\overrightarrow{x} = \overrightarrow{a}G$. With $\overrightarrow{a} = a_1 a_2 a_3 a_4$ with a_i exist in F_2 (A field with two elements namely 0 and 1).

Thus the codewords are all the 4-tuples (k-tuples).

Therefore,

(1,0,1,1) gets encoded as (1,0,1,1,0,1,0).

Hamming(7,4) code with an additional parity bit



The same (7,4) example from above with an extra parity bit

The Hamming(7,4) can easily be extended to an (8,4) code by adding an extra parity bit on top of the (7,4) encoded word. This can be summed up with the revised matrices:

$$\mathbf{G} := \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}_{4,8}$$

and

$$\mathbf{H} := \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}_{4,8}.$$

Note that H is not in standard form. To obtain G, elementary row operations can be used to obtain an equivalent matrix to H in systematic form:

$$\mathbf{H} = \left(\begin{array}{cccc|cccc} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right)_{4,8}.$$

For example, the first row in this matrix is the sum of the second and third rows of H in non-systematic form. Using the systematic construction for Hamming codes from above, the matrix A is apparent and the systematic form of G is written as

$$\mathbf{G} = \left(\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}\right)_{4,8}.$$

The non-systematic form of G can be row reduced (using elementary row operations) to match this matrix.

The addition of the fourth row effectively computes the sum of all the codeword bits (data and parity) as the fourth parity bit.

For example, 1011 is encoded into 01100110 where blue digits are data; red digits are parity from the Hamming(7,4) code; and the green digit is the parity added by Hamming(8,4). The green digit makes the parity of the (7,4) code even.

Finally, it can be shown that the minimum distance has increased from 3, as with the (7,4) code, to 4 with the (8,4) code. Therefore, the code can be defined as Hamming(8,4,4).

Chapter-6

Forward Error Correction and EXIT Chart

Forward error correction

In telecommunication and information theory, **forward error correction (FEC)** (also called **channel coding**) is a system of error control for data transmission, whereby the sender adds systematically generated redundant data to its messages, also known as an **error-correcting code** (ECC). The American mathematician Richard Hamming pioneered this field in the 1940s and invented the first FEC code, the Hamming (7,4) code, in 1950.

The carefully designed redundancy allows the receiver to detect and correct a limited number of errors occurring anywhere in the message without the need to ask the sender for additional data. FEC gives the receiver an ability to correct errors without needing a reverse channel to request retransmission of data, but this advantage is at the cost of a fixed higher forward channel bandwidth. FEC is therefore applied in situations where retransmissions are relatively costly, or impossible such as when broadcasting to multiple receivers. In particular, FEC information is usually added to mass storage devices to enable recovery of corrupted data.

FEC processing in a receiver may be applied to a digital bit stream or in the demodulation of a digitally modulated carrier. For the latter, FEC is an integral part of the initial analog-to-digital conversion in the receiver. The Viterbi decoder implements a soft-decision algorithm to demodulate digital data from an analog signal corrupted by noise. Many FEC coders can also generate a bit-error rate (BER) signal which can be used as feedback to fine-tune the analog receiving electronics.

The maximum fractions of errors or of missing bits that can be corrected is determined by the design of the FEC code, so different forward error correcting codes are suitable for different conditions.

How it works

FEC is accomplished by adding redundancy to the transmitted information using a predetermined algorithm. A redundant bit may be a complex function of many original

information bits. The original information may or may not appear literally in the encoded output; codes that include the unmodified input in the output are **systematic**, while those that do not are **non-systematic**.

A simplistic example of FEC is to transmit each data bit 3 times, which is known as a (3,1) repetition code. Through a noisy channel, a receiver might see 8 versions of the output, see table below.

Triplet received	**Interpreted as**
000	0 (error free)
001	0
010	0
100	0
111	1 (error free)
110	1
101	1
011	1

This allows an error in any one of the three samples to be corrected by "majority vote" or "democratic voting". The correcting ability of this FEC is:

- Up to 1 bit of triplet in error, **or**
- up to 2 bits of triplet omitted (cases not shown in table).

Though simple to implement and widely used, this triple modular redundancy is a relatively inefficient FEC. Better FEC codes typically examine the last several dozen, or even the last several hundred, previously received bits to determine how to decode the current small handful of bits (typically in groups of 2 to 8 bits).

Averaging noise to reduce errors

FEC could be said to work by "averaging noise"; since each data bit affects many transmitted symbols, the corruption of some symbols by noise usually allows the original user data to be extracted from the other, uncorrupted received symbols that also depend on the same user data.

- Because of this "risk-pooling" effect, digital communication systems that use FEC tend to work well above a certain minimum signal-to-noise ratio and not at all below it.
- This *all-or-nothing tendency* — the cliff effect — becomes more pronounced as stronger codes are used that more closely approach the theoretical Shannon limit.
- Interleaving FEC coded data can reduce the all or nothing properties of transmitted FEC codes when the channel errors tend to occur in bursts. However, this method has limits; it is best used on narrowband data.

Most telecommunication systems used a fixed channel code designed to tolerate the expected worst-case bit error rate, and then fail to work at all if the bit error rate is ever worse. However, some systems adapt to the given channel error conditions: hybrid automatic repeat-request uses a fixed FEC method as long as the FEC can handle the error rate, then switches to ARQ when the error rate gets too high; adaptive modulation and coding uses a variety of FEC rates, adding more error-correction bits per packet when there are higher error rates in the channel, or taking them out when they are not needed.

Types of FEC

The two main categories of FEC codes are block codes and convolutional codes.

- Block codes work on fixed-size blocks (packets) of bits or symbols of predetermined size. Practical block codes can generally be decoded in polynomial time to their block length.
- Convolutional codes work on bit or symbol streams of arbitrary length. They are most often 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. A convolutional code can be turned into a block code, if desired, by "tail-biting".

There are many types of block codes, but among the classical ones the most notable is Reed-Solomon coding because of its widespread use on the Compact disc, the DVD, and in hard disk drives. Golay, BCH, Multidimensional parity, and Hamming codes are other examples of classical block codes.

Hamming ECC is commonly used to correct NAND flash memory errors. This provides single-bit error correction and 2-bit error detection. Hamming codes are only suitable for more reliable single level cell (SLC) NAND. Denser multi level cell (MLC) NAND requires stronger multi-bit correcting ECC such as BCH or Reed-Solomon.

Classical block codes are usually implemented using **hard-decision** algorithms, which means that for every input and output signal a hard decision is made whether it corresponds to a one or a zero bit. In contrast, **soft-decision** algorithms like the Viterbi decoder process (discretized) analog signals, which allows for much higher error-correction performance than hard-decision decoding.

Nearly all classical block codes apply the algebraic properties of finite fields.

Concatenated FEC codes for improved performance

Classical (algebraic) block codes and convolutional codes are frequently combined in **concatenated** coding schemes in which a short constraint-length Viterbi-decoded convolutional code does most of the work and a block code (usually Reed-Solomon) with

larger symbol size and block length "mops up" any errors made by the convolutional decoder.

Concatenated codes have been standard practice in satellite and deep space communications since Voyager 2 first used the technique in its 1986 encounter with Uranus.

Low-density parity-check (LDPC)

Low-density parity-check (LDPC) codes are a class of recently re-discovered highly efficient linear block codes. They can provide performance very close to the channel capacity (the theoretical maximum) using an iterated soft-decision decoding approach, at linear time complexity in terms of their block length. Practical implementations can draw heavily from the use of parallelism.

LDPC codes were first introduced by Robert G. Gallager in his PhD thesis in 1960, but due to the computational effort in implementing en- and decoder and the introduction of Reed-Solomon codes, they were mostly ignored until recently.

LDPC codes are now used in many recent high-speed communication standards, such as DVB-S2 (Digital video broadcasting), WiMAX (IEEE 802.16e standard for microwave communications), High-Speed Wireless LAN (IEEE 802.11n), 10GBase-T Ethernet (802.3an) and G.hn/G.9960 (ITU-T Standard for networking over power lines, phone lines and coaxial cable).

Turbo codes

Turbo coding is an iterated soft-decoding scheme that combines two or more relatively simple convolutional codes and an interleaver to produce a block code that can perform to within a fraction of a decibel of the Shannon limit. Predating LDPC codes in terms of practical application, they now provide similar performance.

One of the earliest commercial applications of turbo coding was the CDMA2000 1x (TIA IS-2000) digital cellular technology developed by Qualcomm and sold by Verizon Wireless, Sprint, and other carriers. It is also used for the evolution of CDMA2000 1x specifically for Internet access, 1xEV-DO (TIA IS-856). Like 1x, EV-DO was developed by Qualcomm, and is sold by Verizon Wireless, Sprint, and other carriers (Verizon's marketing name for 1xEV-DO is *Broadband Access*, Sprint's consumer and business marketing names for 1xEV-DO are *Power Vision* and *Mobile Broadband*, respectively.).

Local decoding and testing of codes

Sometimes it is only necessary to decode single bits of the message, or to check whether a given signal is a codeword, and do so without looking at the entire signal. This can make sense in a streaming setting, where codewords are too large to be classically decoded fast enough and where only a few bits of the message are of interest for now.

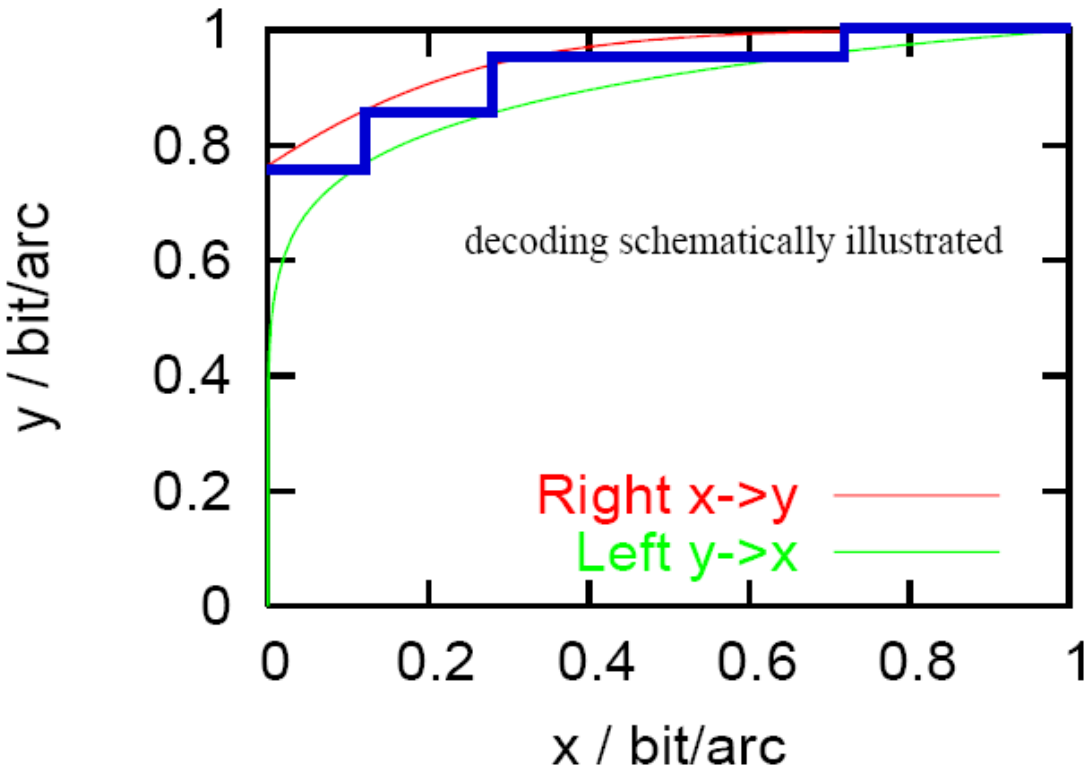
Also such codes have become an important tool in computational complexity theory, e.g., for the design of probabilistically checkable proofs.

Locally decodable codes are error-correcting codes for which single bits of the message can be probabilistically recovered by only looking at a small (say constant) number of positions of a codeword, even after the codeword has been corrupted at some constant fraction of positions. Locally testable codes are error-correcting codes for which it can be checked probabilistically whether a signal is close to a codeword by only looking at a small number of positions of the signal.

List of error-correcting codes

- BCH code
- Constant-weight code
- Convolutional code
- Group codes
- Golay codes, of which the Binary Golay code is of practical interest
- Goppa code, used in the McEliece cryptosystem
- Hadamard code
- Hagelbarger code
- Hamming code
- Latin square based code for non-white noise (prevalent for example in broadband over powerlines)
- Lexicographic code
- Long code
- Low-density parity-check code, also known as Gallager code, as the archetype for sparse graph codes
- LT code, which is a near-optimal rateless erasure correcting code (Fountain code)
- m of n codes
- Online code, a near-optimal rateless erasure correcting code
- Raptor code, a near-optimal rateless erasure correcting code
- Reed–Solomon error correction
- Reed–Muller code
- Repeat-accumulate code
- Repetition codes, such as Triple modular redundancy
- Tornado code, a near-optimal erasure correcting code, and the precursor to Fountain codes
- Turbo code
- Walsh-Hadamard code

EXIT chart



An example EXIT chart showing two components "right" and "left" and an example decoding (blue)

An **Extrinsic information transfer chart**, commonly called an **EXIT chart**, is a technique to aid the construction of good iteratively-decoded error-correcting codes (in particular low-density parity-check (LDPC) codes and Turbo codes).

EXIT charts were developed by Stephan ten Brink, building on the concept of extrinsic information developed in the Turbo coding community. An EXIT chart includes the response of elements of decoder (for example a convolutional decoder of a Turbo code, the LDPC parity-check nodes or the LDPC variable nodes). The response can either be seen as extrinsic information or a representation of the messages in belief propagation.

If there are two components which exchange messages, the behaviour of the decoder can be plotted on a two-dimensional chart. One component is plotted with its input on the horizontal axis and its output on the vertical axis. The other component is plotted with its input on the vertical axis and its output on the horizontal axis. The decoding path followed is found by stepping between the two curves. For a successful decoding, there

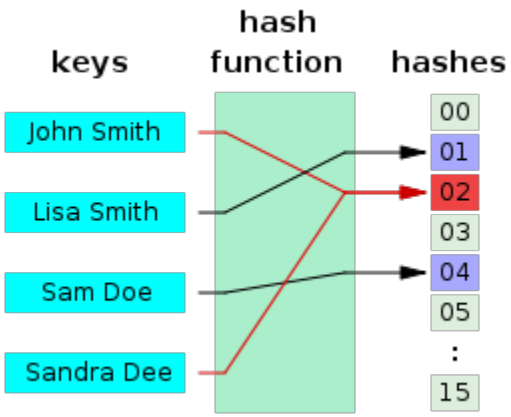
must be a clear swath between the curves so that iterative decoding can proceed from 0 bits of extrinsic information to 1 bit of extrinsic information.

A key assumption is that the messages to and from an element of the decoder can be described by a single number, the extrinsic information. This is true when decoding codes from a binary erasure channel but otherwise the messages are often samples from a Gaussian distribution with the correct extrinsic information. The other key assumption is that the messages are independent (equivalent to an infinite block-size code without local structure between the components)

To make an optimal code, the two transfer curves need to lie close to each other. This observation is supported by the theoretical result that for capacity to be reached for a code over a binary-erasure channel there must be no area between the curves and also by the insight that a large number of iterations are required for information to be spread throughout all bits of a code.

Chapter-7

Hash Function



A hash function that maps names to integers from 0 to 15.. There is a collision between keys "John Smith" and "Sandra Dee".

A **hash function** is any well-defined procedure or mathematical function that converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index to an array (cf. associative array). The values returned by a hash function are called **hash values**, **hash codes**, **hash sums**, **checksums** or simply **hashes**.

Hash functions are mostly used to speed up table lookup or data comparison tasks—such as finding items in a database, detecting duplicated or similar records in a large file, finding similar stretches in DNA sequences, and so on.

A hash function may map two or more keys to the same hash value. In many applications, it is desirable to minimize the occurrence of such collisions, which means that the hash function must map the keys to the hash values as evenly as possible. Depending on the application, other properties may be required as well. Although the idea was conceived in the 1950s, the design of good hash functions is still a topic of active research.

Hash functions are related to (and often confused with) checksums, check digits, fingerprints, randomization functions, error correcting codes, and cryptographic hash functions. Although these concepts overlap to some extent, each has its own uses and requirements and is designed and optimised differently. The HashKeeper database maintained by the American National Drug Intelligence Center, for instance, is more aptly described as a catalog of file fingerprints than of hash values.

Applications

Hash tables

Hash functions are primarily used in hash tables, to quickly locate a data record (for example, a dictionary definition) given its search key (the headword). Specifically, the hash function is used to map the search key to the hash. The index gives the place where the corresponding record should be stored. Hash tables, in turn, are used to implement associative arrays and dynamic sets.

In general, a hashing function may map several different keys to the same index. Therefore, each slot of a hash table is associated with (implicitly or explicitly) a set of records, rather than a single record. For this reason, each slot of a hash table is often called a *bucket*, and hash values are also called *bucket indices*.

Thus, the hash function only hints at the record's location—it tells where one should start looking for it. Still, in a half-full table, a good hash function will typically narrow the search down to only one or two entries.

Caches

Hash functions are also used to build caches for large data sets stored in slow media. A cache is generally simpler than a hashed search table, since any collision can be resolved by discarding or writing back the older of the two colliding items.

Bloom filters

Hash functions are an essential ingredient of the Bloom filter, a compact data structure that provides an enclosing approximation to a set of keys.

Finding duplicate records

When storing records in a large unsorted file, one may use a hash function to map each record to an index into a table T, and collect in each bucket $T[i]$ a list of the numbers of all records with the same hash value i. Once the table is complete, any two duplicate records will end up in the same bucket. The duplicates can then be found by scanning every bucket $T[i]$ which contains two or more members, fetching those records, and comparing them. With a table of appropriate size, this method is likely to be much faster

than any alternative approach (such as sorting the file and comparing all consecutive pairs).

Finding similar records

Hash functions can also be used to locate table records whose key is similar, but not identical, to a given key; or pairs of records in a large file which have similar keys. For that purpose, one needs a hash function that maps similar keys to hash values that differ by at most *m*, where *m* is a small integer (say, 1 or 2). If one builds a table of *T* of all record numbers, using such a hash function, then similar records will end up in the same bucket, or in nearby buckets. Then one need only check the records in each bucket *T*[*i*] against those in buckets *T*[*i*+*k*] where *k* ranges between -*m* and *m*.

This class includes the so-called acoustic fingerprint algorithms, that are used to locate similar-sounding entries in large collection of audio files (as in the MusicBrainz song labeling service). For this application, the hash function must be as insensitive as possible to data capture or transmission errors, and to "trivial" changes such as timing and volume changes, compression, etc.

Finding similar substrings

The same techniques can be used to find equal or similar stretches in a large collection of strings, such as a document repository or a genomic database. In this case, the input strings are broken into many small pieces, and a hash function is used to detect potentially equal pieces, as above.

The Rabin-Karp algorithm is a relatively fast string searching algorithm that works in O(*n*) time on average. It is based on the use of hashing to compare strings.

Geometric hashing

This principle is widely used in computer graphics, computational geometry and many other disciplines, to solve many proximity problems in the plane or in three-dimensional space, such as finding closest pairs in a set of points, similar shapes in a list of shapes, similar images in an image database, and so on. In these applications, the set of all inputs is some sort of metric space, and the hashing function can be interpreted as a partition of that space into a grid of *cells*. The table is often an array with two or more indices (called a *grid file*, *grid index*, *bucket grid*, and similar names), and the hash function returns an index tuple. This special case of hashing is known as geometric hashing or *the grid method*. Geometric hashing is also used in telecommunications (usually under the name vector quantization) to encode and compress multi-dimensional signals.

Properties

Good hash functions, in the original sense of the term, are usually required to satisfy certain properties listed below. Note that different requirements apply to the other related concepts (cryptographic hash functions, checksums, etc.).

Low cost

The cost of computing a hash function must be small enough to make a hashing-based solution more efficient than alternative approaches. For instance, a self-balancing binary tree can locate an item in a sorted table of n items with O(log n) key comparisons. Therefore, a hash table solution will be more efficient than a self-balancing binary tree if the number of items is large and the hash function produces few collisions and less efficient if the number of items is small and the hash function is complex.

Determinism

A hash procedure must be deterministic—meaning that for a given input value it must always generate the same hash value. In other words, it must be a function of the hashed data, in the mathematical sense of the term. This requirement excludes hash functions that depend on external variable parameters, such as pseudo-random number generators or the time of day. It also excludes functions that depend on the memory address of the object being hashed, because that address may change during execution (as may happen on systems that use certain methods of garbage collection), although sometimes rehashing of the item is possible).

Uniformity

A good hash function should map the expected inputs as evenly as possible over its output range. That is, every hash value in the output range should be generated with roughly the same probability. The reason for this last requirement is that the cost of hashing-based methods goes up sharply as the number of *collisions*—pairs of inputs that are mapped to the same hash value—increases. Basically, if some hash values are more likely to occur than others, a larger fraction of the lookup operations will have to search through a larger set of colliding table entries.

Note that this criterion only requires the value to be *uniformly distributed*, not *random* in any sense. A good randomizing function is (barring computational efficiency concerns) generally a good choice as a hash function, but the converse need not be true.

Hash tables often contain only a small subset of the valid inputs. For instance, a club membership list may contain only a hundred or so member names, out of the very large set of all possible names. In these cases, the uniformity criterion should hold for almost all typical subsets of entries that may be found in the table, not just for the global set of all possible entries.

In other words, if a typical set of *m* records is hashed to *n* table slots, the probability of a bucket receiving many more than *m/n* records should be vanishingly small. In particular, if *m* is less than *n*, very few buckets should have more than one or two records. (In an ideal "perfect hash function", no bucket should have more than one record; but a small number of collisions is virtually inevitable, even if *n* is much larger than *m*).

When testing a hash function, the uniformity of the distribution of hash values can be evaluated by the chi-square test.

Variable range

In many applications, the range of hash values may be different for each run of the program, or may change along the same run (for instance, when a hash table needs to be expanded). In those situations, one needs a hash function which takes two parameters—the input data *z*, and the number *n* of allowed hash values.

A common solution is to compute a fixed hash function with a very large range (say, 0 to $2^{32}-1$), divide the result by *n*, and use the division's remainder. If *n* is itself a power of 2, this can be done by bit masking and bit shifting. When this approach is used, the hash function must be chosen so that the result has fairly uniform distribution between 0 and $n-1$, for any *n* that may occur in the application. Depending on the function, the remainder may be uniform only for certain *n*, e.g. odd or prime numbers.

It is possible to relax the restriction of the table size being a power of 2 and not having to perform any modulo, remainder or division operation -as these operation are considered computational costly in some contexts. For example, when *n* is significantly less than 2^b begin with a pseudo random number generator (PRNG) function *P*(*key*), uniform on the interval $[0, 2^b-1]$. Consider the ratio $q = 2^b / n$. Now the hash function can be seen as the value of *P*(*key*) / *q*. Rearranging the calculation and replacing the 2^b-division by bit shifting right (>>) *b* times you end up with hash function $n * P(key) >> b$.

Variable range with minimal movement (dynamic hash function)

When the hash function is used to store values in a hash table that outlives the run of the program, and the hash table needs to be expanded or shrunk, the hash table is referred to as a dynamic hash table.

A hash function that will relocate the minimum number of records when the table is resized is desirable. What is needed is a hash function *H(z,n)* – where *z* is the key being hashed and *n* is the number of allowed hash values – such that $H(z,n+1) = H(z,n)$ with probability close to $n/(n+1)$.

Linear hashing and spiral storage are examples of dynamic hash functions that execute in constant time but relax the property of uniformity to achieve the minimal movement property.

Extendible hashing uses a dynamic hash function that requires space proportional to n to compute the hash function, and it becomes a function of the previous keys that have been inserted.

Several algorithms that preserve the uniformity property but require time proportional to n to compute the value of $H(z,n)$ have been invented.

Data normalization

In some applications, the input data may contain features that are irrelevant for comparison purposes. For example, when looking up a personal name, it may be desirable to ignore the distinction between upper and lower case letters. For such data, one must use a hash function that is compatible with the data equivalence criterion being used: that is, any two inputs that are considered equivalent must yield the same hash value. This can be accomplished by normalizing the input before hashing it, as by upper-casing all letters.

Continuity

A hash function that is used to search for similar (as opposed to equivalent) data must be as continuous as possible; two inputs that differ by a little should be mapped to equal or nearly equal hash values.

Note that continuity is usually considered a fatal flaw for checksums, cryptographic hash functions, and other related concepts. Continuity is desirable for hash functions only in some applications, such as hash tables that use linear search.

Hash function algorithms

For most types of hashing functions the choice of the function depends strongly on the nature of the input data, and their probability distribution in the intended application.

Trivial hash function

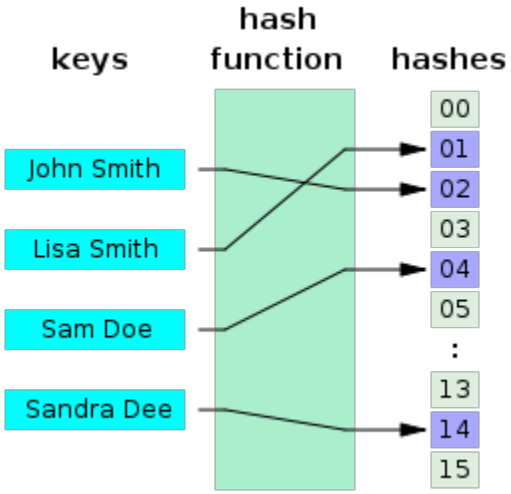
If the datum to be hashed is small enough, one can use the datum itself (reinterpreted as an integer in binary notation) as the hashed value. The cost of computing this "trivial" (identity) hash function is effectively zero.

The meaning of "small enough" depends on how much memory is available for the hash table. A typical PC (as of 2008) might have a gigabyte of available memory, meaning that hash values of up to 30 bits could be accommodated. However, there are many applications that can get by with much less. For example, when mapping character strings between upper and lower case, one can use the binary encoding of each character, interpreted as an integer, to index a table that gives the alternative form of that character ("A" for "a", "8" for "8", etc.). If each character is stored in 8 bits (as in ASCII or ISO

Latin 1), the table has only $2^8 = 256$ entries; in the case of Unicode characters, the table would have $17 \times 2^{16} = 1114112$ entries.

The same technique can be used to map two-letter country codes like "us" or "za" to country names (26^2=676 table entries), 5-digit zip codes like 13083 to city names (100000 entries), etc. Invalid data values (such as the country code "xx" or the zip code 00000) may be left undefined in the table, or mapped to some appropriate "null" value.

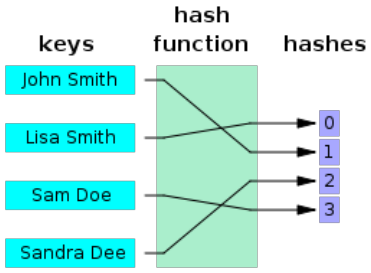
Perfect hashing



A perfect hash function for the four names shown

A hash function that is injective—that is, maps each valid input to a different hash value—is said to be **perfect**. With such a function one can directly locate the desired entry in a hash table, without any additional searching.

Minimal perfect hashing



A minimal perfect hash function for the four names shown

A perfect hash function for n keys is said to be **minimal** if its range consists of n *consecutive* integers, usually from 0 to $n-1$. Besides providing single-step lookup, a minimal perfect hash function also yields a compact hash table, without any vacant slots.

Minimal perfect hash functions are much harder to find than perfect ones with a wider range.

Hashing uniformly distributed data

If the inputs are bounded-length strings (such as telephone numbers, car license plates, invoice numbers, etc.), and each input may independently occur with uniform probability, then a hash function need only map roughly the same number of inputs to each hash value. For instance, suppose that each input is an integer z in the range 0 to $N-1$, and the output must be an integer h in the range 0 to $n-1$, where N is much larger than n. Then the hash function could be $h = z$ **mod** n (the remainder of z divided by n), or $h = (z \times n) \div N$ (the value z scaled down by n/N and truncated to an integer), or many other formulas.

Warning: $h = z$ **mod** n was used in many of the original random number generators, but was found to have a number of issues. One of which is that as n approaches N, this function becomes less and less uniform.

Hashing data with other distributions

These simple formulas will not do if the input values are not equally likely, or are not independent. For instance, most patrons of a supermarket will live in the same geographic area, so their telephone numbers are likely to begin with the same 3 to 4 digits. In that case, if n is 10000 or so, the division formula $(z \times n) \div N$, which depends mainly on the leading digits, will generate a lot of collisions; whereas the remainder formula z **mod** n, which is quite sensitive to the trailing digits, may still yield a fairly even distribution.

Hashing variable-length data

When the data values are long (or variable-length) character strings—such as personal names, web page addresses, or mail messages—their distribution is usually very uneven, with complicated dependencies. For example, text in any natural language has highly non-uniform distributions of characters, and character pairs, very characteristic of the language. For such data, it is prudent to use a hash function that depends on all characters of the string—and depends on each character in a different way.

In cryptographic hash functions, a Merkle–Damgård construction is usually used. In general, the scheme for hashing such data is to break the input into a sequence of small units (bits, bytes, words, etc.) and combine all the units b, b, ..., $b[m]$ sequentially, as follows

```
S ← S0;                          // Initialize the state.
for k in 1, 2, ..., m do          // Scan the input data units:
  S ← F(S, b[k]);                //   Combine data unit k into the
state.
return G(S, n)                   // Extract the hash value from the
state.
```

This schema is also used in many text checksum and fingerprint algorithms. The state variable *S* may be a 32- or 64-bit unsigned integer; in that case, *S0* can be 0, and *G*(*S*,*n*) can be just *S* **mod** *n*. The best choice of *F* is a complex issue and depends on the nature of the data. If the units *b*[*k*] are single bits, then *F*(*S*,*b*) could be, for instance

```
if highbit(S) = 0 then
  return 2 * S + b
else
  return (2 * S + b) ^ P
```

Here *highbit*(*S*) denotes the most significant bit of *S*; the '*' operator denotes unsigned integer multiplication with lost overflow; '^' is the bitwise exclusive or operation applied to words; and *P* is a suitable fixed word.

Special-purpose hash functions

In many cases, one can design a special-purpose (heuristic) hash function that yields many fewer collisions than a good general-purpose hash function. For example, suppose that the input data are file names such as `FILE0000.CHK`, `FILE0001.CHK`, `FILE0002.CHK`, etc., with mostly sequential numbers. For such data, a function that extracts the numeric part *k* of the file name and returns *k* **mod** *n* would be nearly optimal. Needless to say, a function that is exceptionally good for a specific kind of data may have dismal performance on data with different distribution.

Rolling hash

In some applications, such as substring search, one must compute a hash function *h* for every *k*-character substring of a given *n*-character string *t*; where *k* is a fixed integer, and *n* is *k*. The straightforward solution, which is to extract every such substring *s* of *t* and compute *h*(*s*) separately, requires a number of operations proportional to *k*·*n*. However, with the proper choice of *h*, one can use the technique of rolling hash to compute all those hashes with an effort proportional to *k*+*n*.

Universal hashing

A universal hashing scheme is a randomized algorithm that selects a hashing function *h* among a family of such functions, in such a way that the probability of a collision of any two distinct keys is 1/*n*, where *n* is the number of distinct hash values desired—independently of the two keys. Universal hashing ensures (in a probabilistic sense) that the hash function application will behave as well as if it were using a random function, for any distribution of the input data. It will however have more collisions than perfect hashing, and may require more operations than a special-purpose hash function.

Hashing with checksum functions

One can adapt certain checksum or fingerprinting algorithms for use as hash functions. Some of those algorithms will map arbitrary long string data *z*, with any typical real-

world distribution—no matter how non-uniform and dependent—to a 32-bit or 64-bit string, from which one can extract a hash value in 0 through $n-1$.

This method may produce a sufficiently uniform distribution of hash values, as long as the hash range size n is small compared to the range of the checksum or fingerprint function. However, some checksums fare poorly in the avalanche test, which may be a concern in some applications. In particular, the popular CRC32 checksum provides only 16 bits (the higher half of the result) that are usable for hashing. Moreover, each bit of the input has a deterministic effect on each bit of the CRC32, that is one can tell without looking at the rest of the input, which bits of the output will flip if the input bit is flipped; so care must be taken to use all 32 bits when computing the hash from the checksum.

Hashing with cryptographic hash functions

Some cryptographic hash functions, such as SHA-1, have even stronger uniformity guarantees than checksums or fingerprints, and thus can provide very good general-purpose hashing functions.

In ordinary applications, this advantage may be too small to offset their much higher cost. However, this method can provide uniformly distributed hashes even when the keys are chosen by a malicious agent. This feature may help protect services against denial of service attacks.

Origins of the term

The term "hash" comes by way of analogy with its non-technical meaning, to "chop and mix". Indeed, typical hash functions, like the **mod** operation, "chop" the input domain into many sub-domains that get "mixed" into the output range to improve the uniformity of the key distribution.

Donald Knuth notes that Hans Peter Luhn of IBM appears to have been the first to use the concept, in a memo dated January 1953, and that Robert Morris used the term in a survey paper in CACM which elevated the term from technical jargon to formal terminology.

List of hash functions

- Bernstein hash
- Fowler-Noll-Vo hash function (32, 64, 128, 256, 512, or 1024 bits)
- Jenkins hash function (32 bits)
- Pearson hashing (8 bits)
- Zobrist hashing

Chapter-8

Group Code Recording and Binary Golay Code

Group code recording

In computer science, **group code recording (GCR)** refers to several distinct but related encoding methods for magnetic media. The first, used in 6250 cpi magnetic tape, is an error-correcting code combined with a run length limited encoding scheme. The others are different floppy disk encoding methods used in some microcomputers until the late 1980s.

GCR for 9-track reel-to-reel tape

In order to reliably read and write to magnetic tape, several constraints on the signal to be written must be followed. The first is that two adjacent flux reversals must be separated by a certain distance on the media. The second is that there must be a flux reversal often enough to keep the reader's clock in phase with the written signal; that is, the signal must be self-clocking. Prior to 6250 cpi tapes, 1600 cpi tapes satisfied these constraints using a technique called phase encoding, which was only 50% efficient. For 6250 GCR tapes, a (0,2)RLL code is used. This code requires five bits to be written for every four bits of data. The code is structured so that no more than two zero bits (which are represented by lack of a flux reversal) can occur in a row, either within a code or between codes, no matter what the data was. This RLL code is applied independently to the data going to each of the 9 tracks.

Of the 32 5-bit patterns, 8 begin with two consecutive zero bits, 6 others end with two consecutive zero bits, and one more (10001) contains three consecutive zero bits. Removing the all-ones pattern (11111) from the remainder leaves 16 suitable code words.

The 6250 GCR RLL code:

Nibble	Code	Nibble	Code
0000	11001	1000	11010

0001	11011	1001	01001
0010	10010	1010	01010
0011	10011	1011	01011
0100	11101	1100	11110
0101	10101	1101	01101
0110	10110	1110	01110
0111	10111	1111	01111

11 of the nibbles (other than xx00 and 0001) have their code formed by prepending the complement of the msbit. The other 5 values are assigned codes beginning with 11. Nibbles of the form ab00 have codes 11baa□, i.e. the bit reverse of the code for ab11. The code 0001 is assigned the remaining value 11011.

Because of the extremely high density of 6250 cpi tape, the RLL code is not sufficient to ensure reliable data storage. On top of the RLL code, an error-correcting code called the Optimal Rectangular Code (ORC) is applied. This code is a combination of a parity track and polynomial code similar to a CRC, but structured for error correction rather than error detection. For every 7 bytes written to the tape (before RLL encoding), an 8th check byte is calculated and written to the tape. When reading, the parity is calculated on each byte and exclusive-or'd with the contents of the parity track, and the polynomial check code calculated and exclusive-or'd with the received check code, resulting in two 8-bit syndrome words. If these are both zero, the data is error free. Otherwise, error-correction logic in the tape controller corrects the data before it is forwarded to the host. The error correcting code is able to correct any number of errors in any single track, or in any two tracks if the erroneous tracks can be identified by other means.

IBM documents refer to the error correcting code itself as "group coded recording". However, GCR has come to refer to the recording format of 6250 cpi tape as a whole, and later to formats which use similar RLL codes without the error correction code.

GCR for floppy disks

Like magnetic tape drives, floppy disk drives have physical limits on the spacing of flux reversals (also called transitions, represented by 1 bits).

For the Apple II floppy drive, Steve Wozniak invented a floppy controller which (along with the drive itself) imposed two constraints

- Between any two one bits, there may be a maximum of one zero bit.
- Each 8-bit byte must start with a one bit.

The simplest scheme to ensure compliance with these limits is to record an extra "clock" transition between each data bit. This scheme is called FM (Frequency Modulation) or "4 and 4", and allows only 10 256-byte sectors per track to be recorded on a single-density 5¼ floppy.

Wozniak realized that a more complex encoding scheme would allow each 8-bit byte on disk to hold 5 bits of useful data rather than 4 bits. This is because there are 34 bytes which have the top bit set and no two zero bits in a row. This encoding scheme became known as "5 and 3" encoding, and allowed 13 sectors per track; it was used for Apple DOS 3.1, 3.2, and 3.2.1, as well as for the earliest version of Apple CP/M. Later, the design of the floppy drive controller was modified to allow a byte on disk to contain exactly one pair of zero bits in a row. This allowed each 8-bit byte to hold 6 bits of useful data, and allowed 16 sectors per track. This scheme is known as "6 and 2", and was used on Apple Pascal, Apple DOS 3.3 and ProDOS, and later on the 400K and 800K 3½ disks on the Macintosh and Apple II. Apple did not originally call this scheme "GCR", but the term was later applied to it to distinguish it from IBM PC floppies which used the MFM encoding scheme.

Independently, Commodore Business Machines created a Group Code Recording scheme for their Commodore 2040 floppy disk drive (launched in the spring of 1979). The relevant constraint on the 2040 drive was that no more than two zero bits could occur in a row, with no special constraint on the first bit in a byte. This allowed the use of a scheme similar to that used in 6250 tape drives. Every 4 bits of data are translated into 5 bits on disk, according to the following table:

Nibble	Code	Nibble	Code
0000	01010	1000	01001
0001	01011	1001	11001
0010	10010	1010	11010
0011	10011	1011	11011
0100	01110	1100	01101
0101	01111	1101	11101
0110	10110	1110	11110
0111	10111	1111	10101

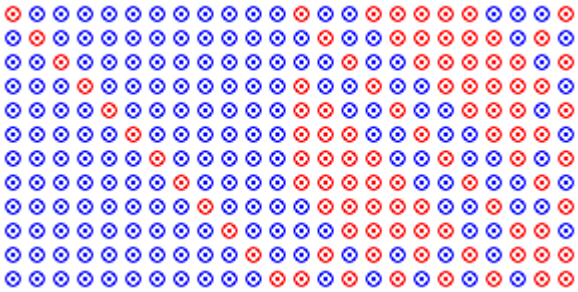
Note no code starts with two zero bits, nor ends with two zero bits. This ensures that regardless of the input data, the encoded data will never contain more than two zero bits in a row. Also note that with this encoding not more than eight one bits in a row are possible. Therefore Commodore used sequences of ten or more one bits in a row as synchronization mark.

Partially because of this more efficient scheme, Commodore was able to fit 170KB on a standard single-density floppy, where Apple fit 140K (6 and 2) or 114K (5 and 3) and an FM-encoded floppy held only 88K.

Binary Golay code

In mathematics and electronics engineering, a **binary Golay code** is a type of error-correcting code used in digital communications. The binary Golay code, along with the ternary Golay code, has a particularly deep and interesting connection to the theory of finite sporadic groups in mathematics. These codes are named in honor of Marcel J. E. Golay.

There are two closely-related binary Golay codes. The **extended binary Golay code** encodes 12 bits of data in a 24-bit word in such a way that any triple-bit error can be corrected and any quadruple-bit error can be detected. The other, the **perfect binary Golay code**, has codewords of length 23 and is obtained from the extended binary Golay code by deleting one coordinate position (conversely, the extended binary Golay code is obtained from the perfect binary Golay code by adding a parity bit). In standard code notation the codes have parameters [24, 12, 8] and [23, 12, 7].



Generator matrix for the extended binary Golay code

Mathematical definition

In mathematical terms, the extended binary Golay code consists of a 12-dimensional subspace W of the space $V=\mathbf{F}_2^{24}$ of 24-bit words such that any two distinct elements of W differ in at least eight coordinates. Equivalently, any non-zero element of W has at least eight non-zero coordinates.

- The possible sets of non-zero coordinates as w ranges over W are called *code words*. In the extended binary Golay code, all code words have the Hamming weights of 0, 8, 12, 16, or 24.
- Up to relabeling coordinates, W is unique.

The perfect binary Golay code is a perfect code. That is, the spheres of radius three around code words form a partition of the vector space.

The automorphism group of the binary Golay code is the Mathieu group M_{23}. The automorphism group of the extended binary Golay code is the Mathieu group M_{24}. The other Mathieu groups occur as stabilizers of one or several elements of W.

The Golay code words of weight eight are elements of the S(5,8,24) Steiner system.

Constructions

1. Lexicographic code: Order the vectors in V lexicographically (i.e., interpret them as unsigned 24-bit binary integers and take the usual ordering). Starting with $w_1 = 0$, define w_2, w_3, ..., w_{12} by the rule that w_n is the smallest integer which differs from all linear combinations of previous elements in at least eight coordinates. Then W can be defined as the span of w_1, ..., w_{12}.
2. Quadratic residue code: Consider the set N of quadratic non-residues (mod 23). This is an 11-element subset of the cyclic group $\mathbf{Z}/23\mathbf{Z}$. Consider the translates $t+N$ of this subset. Augment each translate to a 12-element set S_t by adding an element ∞. Then labeling the basis elements of V by 0, 1, 2, ..., 22, ∞, W can be defined as the span of the words S_t together with the word consisting of all basis vectors. (The perfect code is obtained by leaving out ∞.)
3. As a Cyclic code: The perfect G_{23} code can be constructed via factorization of $x^{23} - 1$, it is the code generated by $x^{11} + x^{10} + x^6 + x^5 + x^4 + x^2 + 1 \mid x^{23} - 1$
4. The Miracle Octad Generator of R. T. Curtis: This uses a 4×6 array of square cells to picture the 759 Hamming-weight-8 code words, or "octads," of the extended binary Golay code. The remaining code words are obtained via symmetric differences of subsets of the 24 cells-- i.e., by binary addition.
5. Winning positions in the mathematical game of Mogul: a position in Mogul is a row of 24 coins. Each turn consists of flipping from one to seven coins such that the leftmost of the flipped coins goes from head to tail. The losing positions are those with no legal move. If heads are interpreted as 1 and tails as 0 then moving to a codeword from the extended binary Golay code guarantees it will be possible to force a win.
6. A generator matrix for the binary Golay code is $\mathbf{I}\ \mathbf{A}$, where $\mathbf{I}$ is the 12×12 identity matrix, and $\mathbf{A}$ is the complement of the adjacency matrix of the icosahedron.

Practical applications of Golay codes

NASA Deep Space Missions

The Voyager 1 & 2 spacecraft needed to transmit hundreds of color pictures of Jupiter and Saturn in their 1979, 1980, and 1981 fly-bys within a constrained telecommunications bandwidth.

- Color image transmission required three times the amount of data, so the Golay (24,12,8) code was used.
- This Golay code is only triple-error correcting, but it could be transmitted at a much higher data rate than the Hadamard code that was used during the Mariner mission.

ALE HF data communications

The new American government standards for automatic link establishment (ALE) in High Frequency (HF) radio systems specifies the use of an extended (24,12) Golay block code for forward error correction (FEC).

- The Extended (24,12) Golay Code specified is a (24,12) block code.
- This code encodes 12 data bits to produce 24-bit code words.
- It is furthermore a systematic code, meaning that the 12 data bits are present in unchanged form in the code word.

The minimum Hamming distance between any two code words (the number of bits by which any pair of code words differs) is eight.

Chapter-9

Casting out Nines and Echo (computing)

Casting out nines

Casting out nines is a sanity check to ensure that hand computations of sums, differences, products, and quotients of integers are correct. By looking at the digital roots of the inputs and outputs, the casting-out-nines method can help one check arithmetic calculations. The method is so simple that most schoolchildren can apply it without understanding its mathematical underpinnings.

Examples

The method involves converting each number into its "casting-out-nines" equivalent, and then redoing the arithmetic. The casting-out-nines answer should equal the casting-out-nines version of the original answer. Below are examples for using casting out nines to check addition, subtraction, multiplication, and division.

Addition

In each addend, cross out all 9's and pairs of digits that total 9, then add together what remains. These new values are called excesses. Add up leftover digits for each addend until one digit is reached. Now process the sum and also the excesses to get a *final* excess.

$$\begin{array}{rl} 3264 \Rightarrow 6 & \text{2 and 4 add up to 6} \\ 8415 \Rightarrow 0 & \text{There are no digits left.} \\ 2946 \Rightarrow 3 & \text{2, 4, and 6 make 12; 1 and 2 make 3.} \\ \underline{+3206} \Rightarrow 2 & \text{2 and 0 are 2.} \\ 17831 \quad \Vert & \text{7, 3, and 1 make 11; 1 and 1 add up to 2.} \\ \Downarrow \quad \Downarrow & \end{array}$$

$2 \quad \Leftrightarrow 2$ The excess from the sum should equal the final excess from the addends.

Subtraction

$5643 \Rightarrow 0(9)$ First, cross out all 9's and digits that total 9 in both minuend and subtrahend (italicized).

$\underline{-2891} \Rightarrow -2$ Add up leftover digits for each value until one digit is reached.

2752 Now follow the same procedure with the difference, coming to a single digit.

$\Downarrow \quad \Downarrow$ Because subtracting 2 from zero gives a negative number, borrow a 9 from the minuend.

$7 \quad \Leftrightarrow 7$ The difference between the minuend and the subtrahend excesses should equal the difference excess.

Multiplication

$548 \Rightarrow 8$ First, cross out all 9's and digits that total 9 in each factor (italicized).

$\underline{\times 629} \Rightarrow 8$ Add up leftover digits for each multiplicand until one digit is reached.

344692 Multiply the two excesses, and then add until one digit is reached.

$\Downarrow \quad \Downarrow$ Do the same with the product, crossing out 9's and getting one digit.

$1 \quad \Leftrightarrow 1^{*}$ The excess from the product should equal the final excess from the factors.

*8 times 8 is 64; 6 and 4 are 10; 1 and 0 are 1

Division

$275462 \div 877 = 314\ r.\ 84$ Cross out all 9's and digits that total 9 in the divisor, quotient, and remainder.

$\Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow$ Add up all uncrossed digits from each value to a single digits.

$8 \quad \Leftrightarrow \quad 4 \times 8 + 3$ The dividend excess should equal the final excess from the other values.

How it works

The method works because the original numbers are 'decimal' (base 10), the modulus is chosen to differ by 1, and casting out is equivalent to taking a digit sum. In general any two 'large' integers, x and y, expressed in any smaller *modulus* as x' and y' (for example,

modulo 7) will always have the same sum, difference, product or quotient as their originals. But this property is also preserved for the 'digit sum' where the 'base' and the 'modulus' differ by 1.

To see this take an example: Both 900 and 630 are exactly divisible by 9 and have the same digit sum - '63' changes into '90' by repeated addition of '09' and the change in the second digit always offsets the change in the first ('63' to '72' to '81' to '90'). For two 'decimal' numbers not generally congruent modulo 9 (like '914' and '673') the picture remains the same; we can consider their congruences *pairwise* ('900' with '630') plus ('09' with '36') plus ('5' with '7'). Thus the digit sum of any number and its congruence modulo 9 are always fixed in base 10.

If a calculation was correct before casting out, casting out on both sides will preserve correctness. However, it is possible that two previously unequal integers will be identical modulo 9 (on average, a ninth of the time).

One should note that the operation does not work on fractions, since a given fractional number does not have a unique representation.

A variation on the explanation

A nice trick for very young children to learn to add nine is to add ten to the digit and to count back one. Since we are adding 1 to the ten's digit and subtracting one from the unit's digit, the sum of the digits should remain the same. For example 9+2=11 with 1+1=2. When adding 9 to itself, we would thus expect the sum of the digits to be 9 as follows: 9+9=18 (1+8=9) and 9+9+9=27 (2+7=9). Let us look at a simple multiplication: 5×7=35 (3+5=8). Now consider (7+9)×5=16×5=80 (8+0=8) or 7×(9+5)=7×14=98 (9+8=17 1+7=8).

Any positive integer can be written as $9 \times n + a$ where 'a' is a single digit 0 to 8 and 'n' is any positive integer. Thus, using the distributive rule $(9 \times n + a)\times(9 \times m + b)= 9 \times 9 \times n \times m + 9 \times(am+bn) +ab$. Since the first two factors are multiplied by 9, their sums will end up being 9 or 0, leaving us with 'ab'. In our example, 'a' was 7 and 'b' was 5. We would expect in any base system the number before that base would behave just like the nine.

History

Abjectio novenaria (Latin for "casting out nines") was known to the Roman bishop Hippolytos as early as the third century. It was employed by Twelfth-century Hindu mathematicians. in the 17th century, Gottfried Wilhelm Leibniz not only used the method extensively, but presented it frequently as a model for rationality: "By means of this, once a reasoning in morality, physics, medicine or metaphysics is reduced to these terms or characters, one will be able to apply to it at any moment a numerical test, so that it will be impossible to be mistaken if one does not so desire...".

Synergetics, R. Buckminster Fuller claims to have used casting out nines "before World War I." Fuller explains how to cast out nines and makes other claims about the resulting 'indigs,' but he fails to note that casting out nines can result in false positives.

The method bears striking resemblance to standard signal processing and computational error detection and error correction methods, typically using similar modular arithmetic in checksums and simpler check digits.

Echo

In computer telecommunications, **echo** is the display or return of sent data at or to the sending end of a transmission. Echo can be either **local echo**, where the sending device itself displays the sent data, or **remote echo**, where the receiving device returns the sent data that it receives to the sender (which is of course simply *no* local echo from the point of view of the sending device itself). That latter, when used as a form of error detection to determine that data received at the remote end of a communications line are the same as data sent, is also known as **echoplex**, **echo check**, or **loop check**. When two modems are communicating in **echoplex mode**, for example, the remote modem echoes whatever it receives from the local modem.

Terminological confusion: echo is not duplex

Whilst local echo and remote echo are sometimes referred to as "half-duplex" and "full-duplex", those latter appellations are strictly incorrect, and are misleading. for their correct meanings.) Whilst local echo is *often used with* half-duplex transmission, so that bandwidth is not wasted upon remotely echoing data from the remote end of the communications channel, it is not *the same as* half-duplex transmission. A full-duplex communications channel can still employ local echo, and a half-duplex channel can still (albeit wastefully) employ remote echo, or no echo at all.

Indeed, for example, that is the case for echoplex error checking, which *requires* full-duplex communication, so that received data can be echoed back as they are being received.

Similarly, for another example, in the case of the TELNET communications protocol a local echo protocol operates on top of a full-duplex underlying protocol. The TCP connection over which the TELNET protocol is layered provides a full-duplex connection, with no echo, across which data may be sent in either direction simultaneously. Whereas the *Network Virtual Terminal* that the TELNET protocol itself incorporates is a half-duplex device with (by default) local echo.

The devices that echo locally

terminals are one of the things that may perform echoing for a connection. Others include modems, some form of intervening communications processor, or even the host system itself. For several common computer operating systems, it is the host system itself that performs the echoing, if appropriate (which it isn't for, say, entry of a user password when a terminal first connects and a user is prompted to log in). On OpenVMS, for example, echoing is performed as necessary by the host system. Similarly, on BSD version 4, local echo is performed by the operating system kernel's *terminal device driver*, according to the state of a device control flag, maintained in software and alterable by applications programs via an `ioctl()` system call. The actual terminals and modems connected to such systems should have *their* local echo facilities switched off (so that they operate in *no echo* mode), lest passwords be locally echoed at password prompts, and all other input appear echoed twice. This is as true for terminal emulator programs, such as C-Kermit, running on a computer as it is for real terminals.

Controlling local echo

Terminal emulators

Most terminal emulator programs have the ability to perform echo locally (which sometimes they mis-name "half-duplex"):

- In the C-Kermit terminal emulator program, local echo is controlled by the `SET TERMINAL ECHO` command, which can be either `SET TERMINAL ECHO LOCAL` (which enables local echoing within the terminal emulator program itself) or `SET TERMINAL ECHO REMOTE` (where disables local echoing, leaving that up to another device in the communications channel — be that the modem or the remote host system — to perform as appropriate).
- In ProComm it is the Alt+E combination, which is a hot key that may be used at any time to toggle local echo on and off.
- In the Terminal program that came with Microsoft Windows 3.1, local echo is controlled by a checkbox in the "Terminal Preferences" dialogue box accessed from the menu of the terminal program's window.

Modems

In the Hayes command set, the `AT` command that switches local echo off (in command mode) is `E0` and the command to switch it on is `E1`. For local echo in data mode, the commands are `F1` and `F0` respectively. (Note the reversal of the numbers. The "F" commands are not part of the EIA/TIA-602 standard, but the "E" commands are.)

Host systems

Some host systems perform local echo themselves, in their device drivers and so forth.

- In Unix and POSIX-compatible systems, local echo is a flag in the POSIX terminal interface, settable programmatically with the tcsetattr() function.[fn 1] The echoing is performed by the operating system's terminal device (in some way that is not specified by the POSIX standard). The standard utility program that alters this flag programmatically is the stty command, using which the flag may be altered from shell scripts or an interactive shell. The command to turn local echo (by the host system) on is stty echo and the command to turn it off is stty -echo.[fn 2]
- On OpenVMS systems, the operating system's terminal driver normally performs echoing. The *terminal characteristic* that controls whether it does this is the `ECHO` characteristic, settable with the DCL command SET TERMINAL /ECHO and unsettable with SET TERMINAL /NOECHO.

Chapter-10

BCH Code

In coding theory the **BCH codes** form a class of parameterised error-correcting codes which have been the subject of much academic attention in the last fifty years. BCH codes were invented in 1959 by Hocquenghem, and independently in 1960 by Bose and Ray-Chaudhuri. The acronym *BCH* comprises the initials of these inventors' names.

The principal advantage of BCH codes is the ease with which they can be decoded, via an elegant algebraic method known as syndrome decoding. This allows very simple electronic hardware to perform the task, obviating the need for a computer, and meaning that a decoding device may be made small and low-powered. As a class of codes, they are also highly flexible, allowing control over block length and acceptable error thresholds, meaning that a custom code can be designed to a given specification (subject to mathematical constraints).

Reed–Solomon codes, which are BCH codes, are used in applications such as satellite communications, compact disc players, DVDs, disk drives, and two-dimensional bar codes.

In technical terms a BCH code is a multilevel cyclic variable-length digital error-correcting code used to correct multiple random error patterns. BCH codes may also be used with multilevel phase-shift keying whenever the number of levels is a prime number or a power of a prime number. A BCH code in 11 levels has been used to represent the 10 decimal digits plus a sign digit.

BCH codes are also useful in theoretical computer science, for instance in the MAXEkSAT problem.

Construction

A BCH code is a polynomial code over a finite field with a particularly chosen generator polynomial. It is also a cyclic code.

Simplified BCH codes

For ease of exposition, we first describe a special class of BCH codes. General BCH codes are described in the next section.

Definition. Fix a finite field $GF(q^m)$, where q is a prime. Also fix positive integers n, and d such that $n = q^m - 1$ and $2 \leq d \leq n$. We will construct a polynomial code over $GF(q)$ with code length n, whose minimum Hamming distance is at least d. What remains to be specified is the generator polynomial of this code.

Let α be a primitive nth root of unity in $GF(q^m)$. For all i, let $m_i(x)$ be the minimal polynomial of α^i with coefficients in $GF(q)$. The generator polynomial of the BCH code is defined as the least common multiple $g(x) = \operatorname{lcm}(m_1(x), \ldots, m_{d-1}(x))$.

Example

Let $q = 2$ and $m = 4$ (therefore $n = 15$). We will consider different values of d. There is a primitive root $\alpha \in GF(16)$satisfying

$$\alpha^4 + \alpha + 1 = 0 \ (1)$$

its minimal polynomial over $GF(2)$ is :$m_1(x) = x^4 + x + 1$. Note that in $GF(2^4)$, the equation $(a + b)^2 = a^2 + 2ab + b^2 = a^2 + b^2$ holds, and therefore $m_1(\alpha^2) = m_1(\alpha)^2 = 0$. Thus α^2 is a root of $m_1(x)$, and therefore

$$m_2(x) = m_1(x) = x^4 + x + 1.$$

To compute $m_3(x)$, notice that, by repeated application of (1), we have the following linear relations:

- $1 = 0\alpha^3 + 0\alpha^2 + 0\alpha + 1$
- $\alpha^3 = 1\alpha^3 + 0\alpha^2 + 0\alpha + 0$
- $\alpha^6 = 1\alpha^3 + 1\alpha^2 + 0\alpha + 0$
- $\alpha^9 = 1\alpha^3 + 0\alpha^2 + 1\alpha + 0$
- $\alpha^{12} = 1\alpha^3 + 1\alpha^2 + 1\alpha + 1$

Five right-hand-sides of length four must be linearly dependent, and indeed we find a linear dependency $\alpha^{12} + \alpha^9 + \alpha^6 + \alpha^3 + 1 = 0$. Since there is no smaller degree dependency, the minimal polynomial of α^3 is :$m_3(x) = x^4 + x^3 + x^2 + x + 1$. Continuing in a similar manner, we find

$$\begin{aligned}
m_4(x) &= m_2(x) = m_1(x) = x^4 + x + 1, \\
m_5(x) &= x^2 + x + 1, \\
m_6(x) &= m_3(x) = x^4 + x^3 + x^2 + x + 1, \\
m_7(x) &= x^4 + x^3 + 1.
\end{aligned}$$

The BCH code with $d = 1,2,3$ has generator polynomial

$$g(x) = m_1(x) = x^4 + x + 1.$$

It has minimal Hamming distance at least 3 and corrects up to 1 error. Since the generator polynomial is of degree 4, this code has 11 data bits and 4 checksum bits.

The BCH code with $d = 4,5$ has generator polynomial

$$g(x) = \operatorname{lcm}(m_1(x), m_3(x)) = (x^4+x+1)(x^4+x^3+x^2+x+1) = x^8+x^7+x^6+x^4+1.$$

It has minimal Hamming distance at least 5 and corrects up to 2 errors. Since the generator polynomial is of degree 8, this code has 7 data bits and 8 checksum bits.

The BCH code with $d = 6,7$ has generator polynomial

$$\begin{aligned} g(x) &= \operatorname{lcm}(m_1(x), m_3(x), m_5(x)) \\ &= (x^4 + x + 1)(x^4 + x^3 + x^2 + x + 1)(x^2 + x + 1) \\ &= x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1. \end{aligned}$$

It has minimal Hamming distance at least 7 and corrects up to 3 errors. This code has 5 data bits and 10 checksum bits.

The BCH code with $d = 8$ and higher have generator polynomial

$$\begin{aligned} g(x) &= \operatorname{lcm}(m_1(x), m_3(x), m_5(x), m_7(x)) \\ &= (x^4 + x + 1)(x^4 + x^3 + x^2 + x + 1)(x^2 + x + 1)(x^4 + x^3 + 1) \\ &= x^{14} + x^{13} + x^{12} + \cdots + x^2 + x + 1. \end{aligned}$$

This code has minimal Hamming distance 8 and corrects up to 3 errors. It has 1 data bit and 14 checksum bits. In fact, this code has only two codewords: 000000000000000 and 111111111111111.

General BCH codes

General BCH codes differ from the simplified case discussed above in two respects. First, one replaces the requirement $n = q^m - 1$ by a more general condition. Second, the consecutive roots of the generator polynomial may run from $\alpha^c, \ldots, \alpha^{c+d-2}$ instead of $\alpha, \ldots, \alpha^{d-1}$.

Definition. Fix a finite field $GF(q)$, where q is a prime power. Choose positive integers m,n,d,c such that $2 \leq d \leq n$, $\gcd(n,q) = 1$, and m is the multiplicative order of q modulo n.

As before, let α be a primitive nth root of unity in $GF(q^m)$, and let $m_i(x)$ be the minimal polynomial over $GF(q)$ of α^i for all i. The generator polynomial of the BCH code is defined as the least common multiple $g(x) = \operatorname{lcm}(m_c(x), \ldots, m_{c+d-2}(x))$.

Note: if $n = q^m - 1$ as in the simplified definition, then gcd(n,q) is automatically 1, and the order of q modulo n is automatically m. Therefore, the simplified definition is indeed a special case of the general one.

Properties

1. The generator polynomial of a BCH code has degree at most $(d - 1)m$. Moreover, if $q = 2$ and $c = 1$, the generator polynomial has degree at most $dm / 2$.

> Proof: each minimal polynomial $m_i(x)$ has degree at most m. Therefore, the least common multiple of $d - 1$ of them has degree at most $(d - 1)m$. Moreover, if $q = 2$, then $m_i(x) = m_{2i}(x)$ for all i. Therefore, $g(x)$ is the least common multiple of at most $d / 2$ minimal polynomials $m_i(x)$ for odd indices i, each of degree at most m.

2. A BCH code has minimal Hamming distance at least d. Proof: We only give the proof in the simplified case; the general case is similar. Suppose that $p(x)$ is a code word with fewer than d non-zero terms. Then

$$p(x) = b_1 x^{k_1} + \cdots + b_{d-1} x^{k_{d-1}}, \text{ where } k_1 < k_2 < \cdots < k_{d-1}.$$

Recall that $\alpha^1, \ldots, \alpha^{d-1}$ are roots of $g(x)$, hence of $p(x)$. This implies that $b_1, \ldots, b_{d-1}$ satisfy the following equations, for $i = 1, \ldots, d-1$:

$$p(\alpha^i) = b_1 \alpha^{ik_1} + b_2 \alpha^{ik_2} + \cdots + b_{d-1} \alpha^{ik_{d-1}} = 0$$

In matrix form, we have

$$\begin{bmatrix} \alpha^{k_1} & \alpha^{k_2} & \cdots & \alpha^{k_{d-1}} \\ \alpha^{2k_1} & \alpha^{2k_2} & \cdots & \alpha^{2k_{d-1}} \\ \vdots & \vdots & & \vdots \\ \alpha^{(d-1)k_1} & \alpha^{(d-1)k_2} & \cdots & \alpha^{(d-1)k_{d-1}} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{d-1} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

The determinant of this matrix equals

$$\left(\prod_{i=1}^{d-1}\alpha^{k_i}\right)\det\begin{pmatrix}1 & 1 & \cdots & 1\\ \alpha^{k_1} & \alpha^{k_2} & \cdots & \alpha^{k_{d-1}}\\ \vdots & \vdots & & \vdots\\ \alpha^{(d-2)k_1} & \alpha^{(d-2)k_2} & \cdots & \alpha^{(d-2)k_{d-1}}\end{pmatrix} = \left(\prod_{i=1}^{d-1}\alpha^{k_i}\right)\det(V)$$

The matrix V is seen to be a Vandermonde matrix, and its determinant is

$$\det(V) = \prod_{1\le i<j\le d-1}(\alpha^{k_j} - \alpha^{k_i}),$$

which is non-zero. It therefore follows that $b_1, \ldots, b_{d-1} = 0$, hence $p(x) = 0$.

3. A BCH code is cyclic.

Proof: A polynomial code of length n is cyclic if and only if its generator polynomial divides $x^n - 1$. Since $g(x)$ is the minimal polynomial with roots $\alpha^c, \ldots, \alpha^{c+d-2}$, it suffices to check that each of $\alpha^c, \ldots, \alpha^{c+d-2}$ is a root of $x^n - 1$. This follows immediately from the fact that α is, by definition, an nth root of unity.

Special cases

- A BCH code with $c = 1$ is called a *narrow-sense BCH code*.
- A BCH code with $n = q^m - 1$ is called *primitive*.

Therefore, the "simplified" BCH codes we considered above were just the primitive narrow-sense codes.

- A narrow-sense BCH code with $n = q^m - 1$ is called a *Reed-Solomon code*.

Decoding

There are many algorithms for decoding BCH codes. The most common ones follow this general outline:

1. Calculate the syndrome values for the received vector
2. Calculate the error locator polynomials
3. Calculate the roots of this polynomial to get error location positions.
4. Calculate the error values at these error locations.

Calculate the syndromes

The received vector R is the sum of the correct codeword C and an unknown error vector E. The syndrome values are formed by considering R as a polynomial and evaluating it at $\alpha^c, \ldots, \alpha^{c+d-2}$. Thus the syndromes are

$$s_j = R(\alpha^{c+j-1}) = C(\alpha^{c+j-1}) + E(\alpha^{c+j-1})$$

for $j = 1$ to $d - 1$. Since α^{c+j-1} are the zeros of $g(x)$, of which $C(x)$ is a multiple, $C(\alpha^{c+j-1}) = 0$. Examining the syndrome values thus isolates the error vector so we can begin to solve for it.

If there is no error, $s_j = 0$ for all j. If the syndromes are all zero, then the decoding is done.

Calculate the error location polynomial

If there are nonzero syndromes, then there are errors. The decoder needs to figure out how many errors and the location of those errors.

If there is a single error, write this as $E(x) = e\,x^i$, where i is the location of the error and e is its magnitude. Then the first two syndromes are

$$\begin{aligned} s_1 &= e\,\alpha^{ci} \\ s_2 &= e\,\alpha^{(c+1)\,i} = \alpha^i s_1 \end{aligned}$$

so together they allow us to calculate e and provide some information about i (completely determining it in the case of Reed-Solomon codes).

If there are two or more errors,

$$E(x) = e_1 x^{i_1} + e_2 x^{i_2} + \ldots$$

It is not immediately obvious how to begin solving the resulting syndromes for the unknowns e_k and i_k. Two popular algorithms for this task are:

1. Peterson-Gorenstein-Zierler algorithm
2. Berlekamp-Massey algorithm

Peterson Gorenstein Zierler algorithm

Peterson's algorithm is the step 2 of the generalized BCH decoding procedure. We use Peterson's algorithm to calculate the error locator polynomial coefficients $\lambda_1, \lambda_2, \ldots, \lambda_t$ of a polynomial $\Lambda(x) = 1 + \lambda_1 x + \lambda_2 x^2 + \cdots + \lambda_t x^t$

Now the procedure of the Peterson Gorenstein Zierler algorithm for a given (n,k,d_{min}) BCH code designed to correct $[t = \frac{d_{min} - 1}{2}]$ errors is

- First generate the Matrix of $2t$ syndromes
- Next generate the $S_{t \times t}$ matrix with elements that are syndrome values

$$S_{t \times t} = \begin{bmatrix} s_1 & s_2 & s_3 & \dots & s_t \\ s_2 & s_3 & s_4 & \dots & s_{t+1} \\ s_3 & s_4 & s_5 & \dots & s_{t+2} \\ \dots & \dots & \dots & \dots & \dots \\ s_t & s_{t+1} & s_{t+2} & \dots & s_{2t-1} \end{bmatrix}$$

- Generate a c_{tx1} vector with elements

$$C_{t \times 1} = \begin{bmatrix} s_{t+1} \\ s_{t+2} \\ \vdots \\ \vdots \\ s_{2t} \end{bmatrix}$$

- Let Λ denote the unknown polynomial coefficients, which are given by

$$\Lambda_{t \times 1} = \begin{bmatrix} \lambda_t \\ \lambda_{t-1} \\ \vdots \\ \lambda_2 \\ \lambda_1 \end{bmatrix}$$

- Form the matrix equation

$$S_{t \times t} \Lambda_{t \times 1} = C_{t \times 1}$$

- If the determinant of matrix $S_{t \times t}$ exists, then we can actually find an inverse of this matrix and solve for the values of unknown Λ values.
- If $\det(S_{t \times t}) = 0$, then follow

```
if t = 0
```

```
then
      declare an empty error locator polynomial
      stop Peterson procedure.
end
set t ← t − 1
continue from the beginning of Peterson's decoding
```

- After you have values of Λ you have with you the error locator polynomial.
- Stop Peterson procedure.

Factor error locator polynomial

Now that you have the Λ(*x*) polynomial, you can find its roots in the form $\Lambda(x) = (\alpha^i x + 1)(\alpha^j x + 1) \cdots (\alpha^k x + 1)$ using the Chien search algorithm. The exponential powers of the primitive element α will yield the positions where errors occur in the received word; hence the name 'error locator' polynomial.

Calculate error values

Once the error locations are known, the next step is to determine the error values at those locations. The error values are then used to correct the received values at those locations to recover the original codeword.

For the case of binary BCH, this is trivial; just flip the bits for the received word at these positions, and we have the corrected code word. In the more general case, the error weights e_j can be determined by solving the linear system

$$\begin{aligned} s_1 &= e_1 \alpha^{c\,i_1} + e_2 \alpha^{c\,i_2} + \ldots \\ s_2 &= e_1 \alpha^{(c+1)\,i_1} + e_2 \alpha^{(c+1)\,i_2} + \ldots \\ &\ldots \end{aligned}$$

However, there is a more efficient method known as the Forney Algorithm, which is based on Lagrange interpolation. First calculate the error evaluator polynomial

$$\omega(x) = S(x)\,\Lambda(x) \pmod{x^{d-1}}$$

Then evaluate the error values.

$$e_j = \frac{\omega(\alpha^{-c\,i_j})}{\prod_{k \neq j}(1 - \alpha^{c\,i_k}\alpha^{-c\,i_j})}$$

Chapter-11

Viterbi Decoder

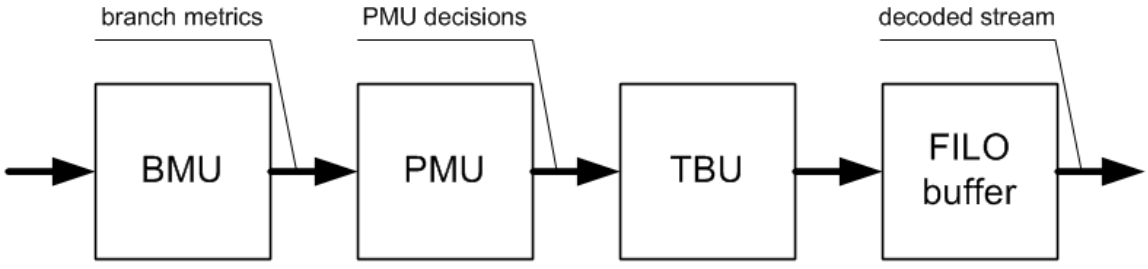
A **Viterbi decoder** uses the Viterbi algorithm for decoding a bitstream that has been encoded using forward error correction based on a convolutional code.

There are other algorithms for decoding a convolutionally encoded stream (for example, the Fano algorithm). The Viterbi algorithm is the most resource-consuming, but it does the maximum likelihood decoding. It is most often used for decoding convolutional codes with constraint lengths k<=10, but values up to k=15 are used in practice.

Viterbi decoding was developed by Andrew J. Viterbi and published in the paper "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm", IEEE Transactions on Information Theory, Volume IT-13, pages 260-269, in April, 1967.

There are both hardware (in modems) and software implementations of a Viterbi decoder.

Hardware implementation



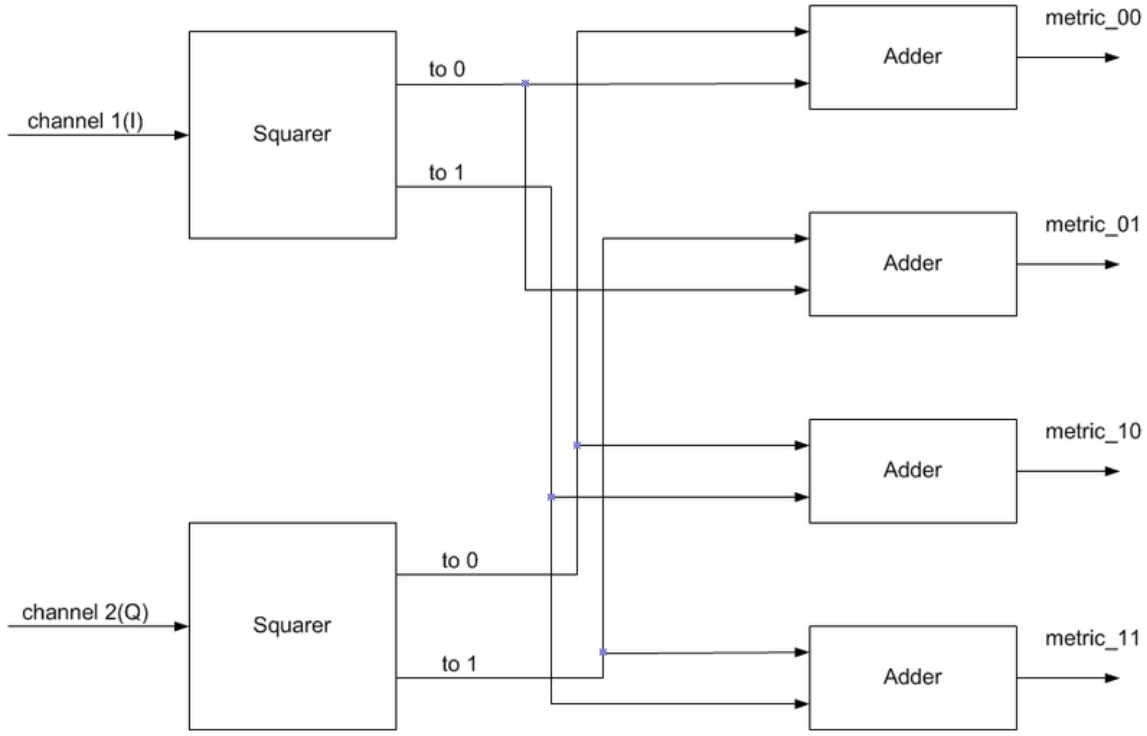
A common way to implement a hardware viterbi decoder

A hardware Viterbi decoder for basic (not perforated) code usually consists of the following major blocks:

- Branch metric unit (BMU)
- Path metric unit (PMU)

- Traceback unit (TBU)

Branch metric unit (BMU)



A sample implementation of a branch metric unit

A branch metric unit's function is to calculate *branch metrics*, which are normed distances between every possible symbol in the code alphabet, and the received symbol.

There are hard decision and soft decision Viterbi decoders. A hard decision Viterbi decoder receives a simple bitstream on its input, and a Hamming distance is used as a metric. A soft decision Viterbi decoder receives a bitstream containing information about the *reliability* of each received symbol. For instance, in a 3-bit encoding, this *reliability* information is encoded as follows:

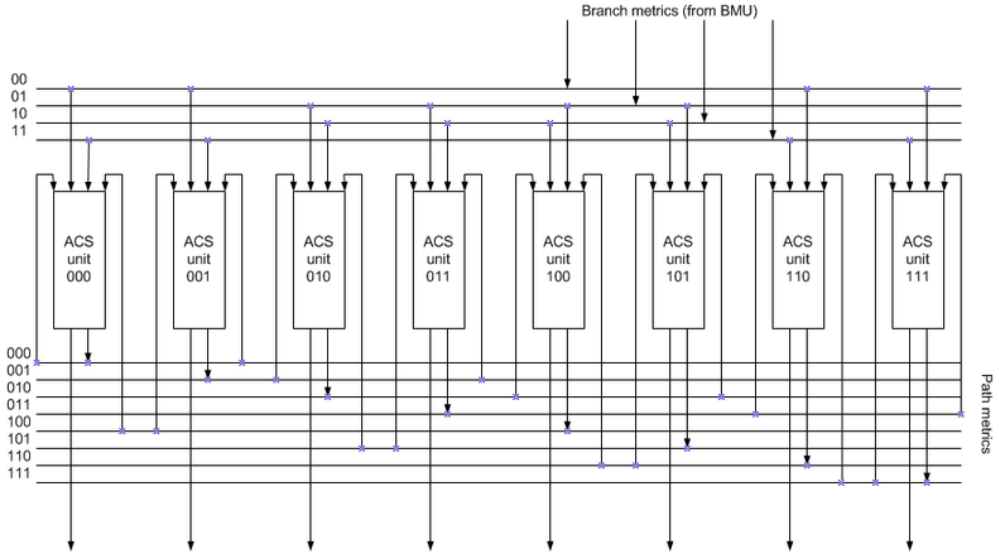
value	meaning
000	strongest **0**
001	relatively strong **0**
010	relatively weak **0**
011	weakest **0**
100	weakest **1**
101	relatively weak **1**

110	relatively strong **1**
111	strongest **1**

Of course, it is not the only way to encode reliability data.

The *squared* Euclidean distance is used as a metric for soft decision decoders.

Path metric unit (PMU)

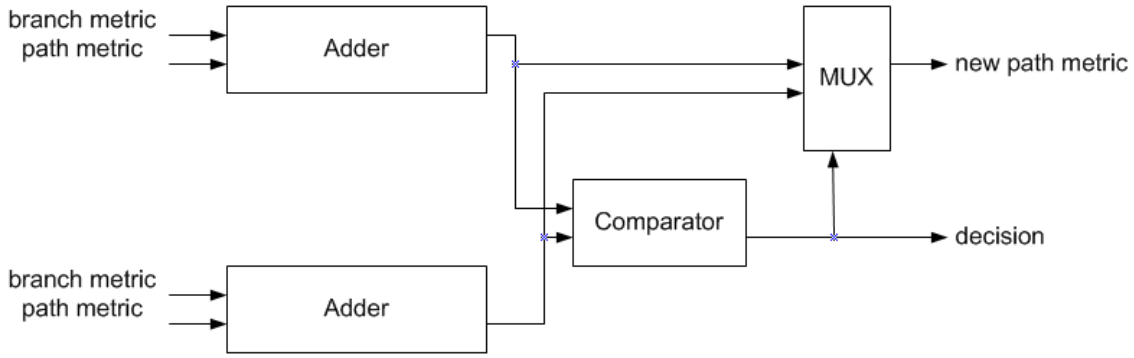


A sample implementation of a path metric unit for a specific K=4 decoder

A path metric unit summarizes branch metrics to get metrics for 2^{K-1} paths, one of which can eventually be chosen as *optimal*. Every clock it makes 2^{K-1} decisions, throwing off wittingly nonoptimal paths. The results of these decisions are written to the memory of a traceback unit.

The core elements of a PMU are *ACS (Add-Compare-Select)* units. The way in which they are connected between themselves is defined by a specific code's trellis diagram.

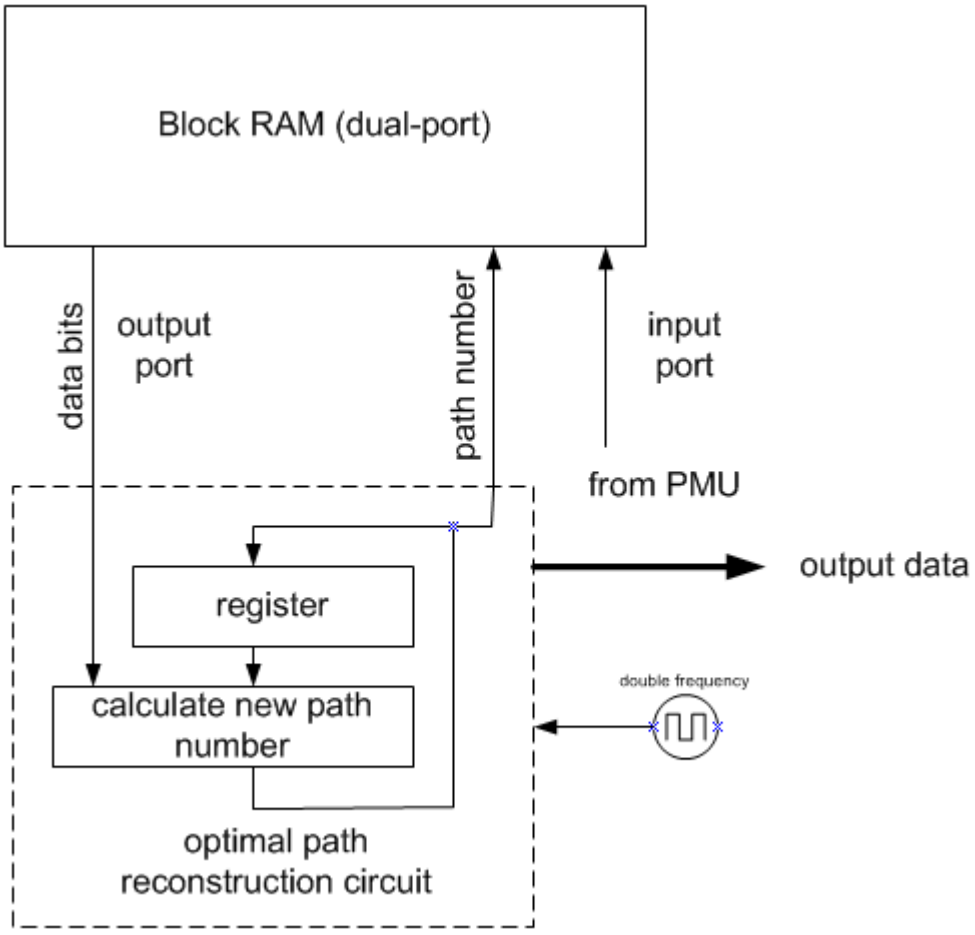
Since branch metrics are always ≥ 0, there must be an additional circuit preventing metric counters from overflow (it isn't shown on the image). An alternate method that eliminates the need to monitor the path metric growth is to allow the path metrics to "roll over", to use this method it is necessary to make sure the path metric accumulators contain enough bits to prevent the "best" and "worst" values from coming within $2^{(n-1)}$ of each other. The compare circuit is essentially unchanged.



A sample implementation of an ACS unit

It is possible to monitor the noise level on the incoming bit stream by monitoring the rate of growth of the "best" path metric, a simpler way to do this is to monitor a single location or "state" and watch it pass "upward" through say four discrete levels within the range of the accumulator. As it passes upward through each of these thresholds, a counter is incremented that reflects the "noise" present on the incoming signal.

Traceback unit (TBU)



A sample implementation of a traceback unit

Back-trace unit restores an (almost) maximum-likelihood path from the decisions made by PMU. Since it does it in inverse direction, a viterbi decoder comprises a FILO (first-in-last-out) buffer to reconstruct a correct order.

Note that the implementation shown on the image requires double frequency. There are some tricks that eliminate this requirement.

Implementation issues

Quantization for soft decision decoding

In order to fully exploit benefits of soft decision decoding, one needs to quantize input signal properly. The optimal quantization zone width is defined by the following formula:

$$T = \sqrt{N_0/2^k},$$

where N_0 is a noise power spectral density, and k is a number of bits for soft decision.

Euclidean metric computation

The squared norm distance (l_2) distance between the received and the actual symbols in the code alphabet may be further simplified into a linear sum/difference form, which makes it less computationally intensive.

Consider a 1/2 convolutional coder, which generates 2 bits (*00*, *01*, *10* or *11*) for every input bit (*1* or *0*). These *Return-to-Zero* signals are translated into a *Non-Return-to-Zero* form shown alongside.

code alphabet	vector mapping
00	*1, 1*
01	*1, -1*
10	*-1, 1*
11	*-1, -1*

Each received symbol may be represented in vector form as $\mathbf{v_r} = \{r_0, r_1\}$, where r_0 and r_1 are soft decision values, whose magnitudes signify the *joint reliability* of the received vector, $\mathbf{v_r}$.

Every symbol in the code alphabet may, likewise, be represented by the vector $\mathbf{v_i} = \{\pm 1, \pm 1\}$.

The actual computation of the Euclidean distance metric is:

$$D = (\overrightarrow{v_r} - \overrightarrow{v_i})^2 = \overrightarrow{v_r}^2 - 2\overrightarrow{v_r}\overrightarrow{v_i} + \overrightarrow{v_i}^2$$

Each square term is a normed distance, depicting the *energy* of the symbol. For ex., the *energy* of the symbol $\mathbf{v_i} = \{\pm 1, \pm 1\}$ may be computed as

$$\overrightarrow{v_i}^2 = (\pm 1)^2 + (\pm 1)^2 = 2$$

Thus, the energy term of all symbols in the code alphabet is constant (at (*normalized*) value 2).

The *Add-Compare-Select* (*ACS*) operation compares the metric distance between the received symbol $\|\mathbf{v_r}\|$ and any 2 symbols in the code alphabet whose paths merge at a node in the corresponding trellis, $\|\mathbf{v_i}^{(0)}\|$ and $\|\mathbf{v_i}^{(1)}\|$. This is equivalent to comparing

$$D_0 = \overrightarrow{v_r}^2 - 2\overrightarrow{v_r}\overrightarrow{v_i^0} + \overrightarrow{v_i^0}^2$$

and

$$D_1 = \overrightarrow{v_r}^2 - 2\overrightarrow{v_r}\overrightarrow{v_i^1} + \overrightarrow{v_i^1}^2$$

But, from above we know that the *energy* of $\mathbf{v_i}$ is constant (equal to (normalized) value of 2), and the *energy* of $\mathbf{v_r}$ is the same in both cases. This reduces the comparison to a minima function between the 2 (middle) *dot product* terms,

$$min(-2\overrightarrow{v_r}\overrightarrow{v_i^0}, -2\overrightarrow{v_r}\overrightarrow{v_i^1}) = max(\overrightarrow{v_r}\overrightarrow{v_i^0}, \overrightarrow{v_r}\overrightarrow{v_i^1})$$

since a *min* operation on negative numbers may be interpreted as an equivalent *max* operation on positive quantities.

Each *dot product* term may be expanded as

$$max(\pm r_0 \pm r_1, \pm r_0 \pm r_1)$$

where, the signs of each term depend on symbols, $\mathbf{v_i^{(0)}}$ and $\mathbf{v_i^{(1)}}$, being compared. Thus, the *squared* Euclidean metric distance calculation to compute the *branch metric* may be performed with a simple add/subtract operation.

Traceback

The general approach to traceback is to accumulate path metrics for up to five times the constraint length (*5 * (K - 1)*), find the node with the largest accumulated cost, and begin traceback from this node.

However, computing the node which has accumulated the largest cost (either the largest or smallest integral path metric) involves finding the *maxima* or *minima* of several (usually 2^{K-1}) numbers, which may be time consuming when implemented on embedded hardware systems.

Most communication systems employ Viterbi decoding involving data packets of fixed sizes, with a fixed bit/byte pattern either at the beginning or/and at the end of the data packet. By using the known bit/byte pattern as reference, the start node may be set to a fixed value, thereby obtaining a perfect Maximum Likelihood Path during traceback.

Limitations

A physical implementation of a viterbi decoder will not yield an *exact* maximum-likelihood stream due to quantization of the input signal, branch and path metrics, and finite *traceback length*. Practical implementations do approach within 1dB of the ideal.

Perforated codes

A hardware viterbi decoder of *perforated codes* is commonly implemented in such a way:

- A deperforator, which transforms the input stream into the stream which looks like an original (imperforated) stream with ERASE marks at the places where bits were erased.
- A basic viterbi decoder understanding these ERASE marks (that is, not using them for branch metric calculation).

Software implementation

One of the most time-consuming operations is an ACS butterfly, which is usually implemented using an assembly language and appropriate instruction set extensions (such as SSE2) to speed up the decoding time.

Applications

The Viterbi decoding algorithm is widely used in the following areas:

- Decoding trellis-coded modulation (TCM), the technique used in telephone-line modems to squeeze high spectral efficiency out of 3 kHz-bandwidth analog telephone lines. The TCM is also used in the PSK31 digital mode for amateur radio and sometimes in the radio relay and satellite communications.
- Automatic speech recognition
- Decoding convolutional codes in satellite communications.
- Computer storage devices such as hard disk drives.

Chapter-12

Viterbi Algorithm

The **Viterbi algorithm** is a dynamic programming algorithm for finding the most likely sequence of hidden states – called the **Viterbi path** – that results in a sequence of observed events, especially in the context of Markov information sources, and more generally, hidden Markov models. The forward algorithm is a closely related algorithm for computing the probability of a sequence of observed events. These algorithms belong to the realm of information theory.

The algorithm makes a number of assumptions.

- First, both the observed events and hidden events must be in a sequence. This sequence often corresponds to time.
- Second, these two sequences need to be aligned, and an instance of an observed event needs to correspond to exactly one instance of a hidden event.
- Third, computing the most likely hidden sequence up to a certain point t must depend only on the observed event at point t, and the most likely sequence at point $t - 1$.

These assumptions are all satisfied in a first-order hidden Markov model.

The terms "Viterbi path" and "Viterbi algorithm" are also applied to related dynamic programming algorithms that discover the single most likely explanation for an observation. For example, in statistical parsing a dynamic programming algorithm can be used to discover the single most likely context-free derivation (parse) of a string, which is sometimes called the "Viterbi parse".

The Viterbi algorithm was conceived by Andrew Viterbi in 1967 as a decoding algorithm for convolutional codes over noisy digital communication links. The algorithm has found universal application in decoding the convolutional codes used in both CDMA and GSM digital cellular, dial-up modems, satellite, deep-space communications, and 802.11 wireless LANs. It is now also commonly used in speech recognition, keyword spotting, computational linguistics, and bioinformatics. For example, in speech-to-text (speech recognition), the acoustic signal is treated as the observed sequence of events, and a

string of text is considered to be the "hidden cause" of the acoustic signal. The Viterbi algorithm finds the most likely string of text given the acoustic signal.

Overview

The assumptions listed above can be elaborated as follows. The Viterbi algorithm operates on a state machine assumption. That is, at any time the system being modeled is in one of a finite number of states. While multiple sequences of states (paths) can lead to a given state, at least one of them is a most likely path to that state, called the "survivor path". This is a fundamental assumption of the algorithm because the algorithm will examine all possible paths leading to a state and only keep the one most likely. This way the algorithm does not have to keep track of all possible paths, only one per state.

A second key assumption is that a transition from a previous state to a new state is marked by an incremental metric, usually a number. This transition is computed from the event. The third key assumption is that the events are cumulative over a path in some sense, usually additive. So the crux of the algorithm is to keep a number for each state. When an event occurs, the algorithm examines moving forward to a new set of states by combining the metric of a possible previous state with the incremental metric of the transition due to the event and chooses the best. The incremental metric associated with an event depends on the transition possibility from the old state to the new state. For example in data communications, it may be possible to only transmit half the symbols from an odd numbered state and the other half from an even numbered state. Additionally, in many cases the state transition graph is not fully connected. A simple example is a car that has three states — forward, stop and reverse — and is not allowed to undergo a transition from forward to reverse without first entering the stop state. After computing the combinations of incremental metric and state metric, only the best survives and all other paths are discarded. There are modifications to the basic algorithm which allow for a forward search in addition to the backwards one described here.

Path history must be stored. In some cases, the search history is complete because the state machine at the encoder starts in a known state and there is sufficient memory to keep all the paths. In other cases, a programmatic solution must be found for limited resources: one example is convolutional encoding, where the decoder must truncate the history at a depth large enough to keep performance to an acceptable level. Although the Viterbi algorithm is very efficient and there are modifications that reduce the computational load, the memory requirements tend to remain constant.

Algorithm

Suppose we are given a Hidden Markov Model (HMM) with states Y, initial probabilities π_i of being in state i and transition probabilities $a_{i,j}$ of transitioning from state i to state j. Say we observe outputs $x_0, \ldots, x_T$. The state sequence $y_0, \ldots, y_T$ most likely to have produced the observations is given by the recurrence relations:

$$
\begin{aligned}
V_{0,k} &= \mathrm{P}\left(x_0 \mid k\right) \cdot \pi_k \\
V_{t,k} &= \mathrm{P}\left(x_t \mid k\right) \cdot \max_{y \in Y} \left(a_{y,k} V_{t-1,y}\right)
\end{aligned}
$$

Here $V_{t,k}$ is the probability of the most probable state sequence responsible for the first $t + 1$ observations (we add one because indexing started at 0) that has k as its final state. The Viterbi path can be retrieved by saving back pointers that remember which state y was used in the second equation. Let Ptr(k,t) be the function that returns the value of y used to compute $V_{t,k}$ if $t > 0$, or k if $t = 0$. Then:

$$
\begin{aligned}
y_T &= \arg\max_{y \in Y} (V_{T,y}) \\
y_{t-1} &= \mathrm{Ptr}(y_t, t)
\end{aligned}
$$

The complexity of this algorithm is $O(T \times |Y|^2)$.

Example

Consider two friends, Alice and Bob, who live far apart from each other and who talk together daily over the telephone about what they did that day. Bob is only interested in three activities: walking in the park, shopping, and cleaning his apartment. The choice of what to do is determined exclusively by the weather on a given day. Alice has no definite information about the weather where Bob lives, but she knows general trends. Based on what Bob tells her he did each day, Alice tries to guess what the weather must have been like.

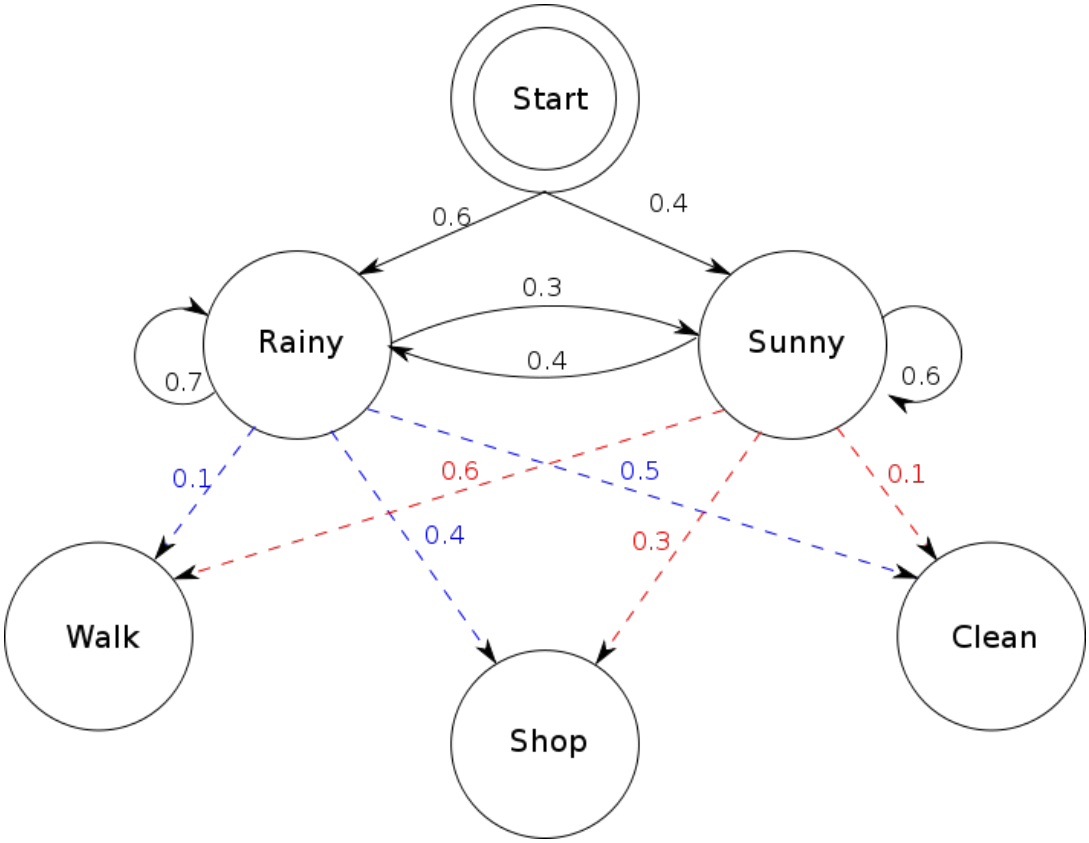
Alice believes that the weather operates as a discrete Markov chain. There are two states, "Rainy" and "Sunny", but she cannot observe them directly, that is, they are *hidden* from her. On each day, there is a certain chance that Bob will perform one of the following activities, depending on the weather: "walk", "shop", or "clean". Since Bob tells Alice about his activities, those are the *observations*. The entire system is that of a hidden Markov model (HMM).

Alice knows the general weather trends in the area, and what Bob likes to do on average. In other words, the parameters of the HMM are known. They can be represented as follows in the Python programming language:

```
states = ('Rainy', 'Sunny')

observations = ('walk', 'shop', 'clean')

start_probability = {'Rainy': 0.6, 'Sunny': 0.4}

transition_probability = {
   'Rainy' : {'Rainy': 0.7, 'Sunny': 0.3},
   'Sunny' : {'Rainy': 0.4, 'Sunny': 0.6},
   }

emission_probability = {
```

```
    'Rainy' : {'walk': 0.1, 'shop': 0.4, 'clean': 0.5},
    'Sunny' : {'walk': 0.6, 'shop': 0.3, 'clean': 0.1},
    }
```

In this piece of code, `start_probability` represents Alice's belief about which state the HMM is in when Bob first calls her (all she knows is that it tends to be rainy on average). The particular probability distribution used here is not the equilibrium one, which is (given the transition probabilities) approximately `{'Rainy': 0.57, 'Sunny': 0.43}`. The `transition_probability` represents the change of the weather in the underlying Markov chain. In this example, there is only a 30% chance that tomorrow will be sunny if today is rainy. The `emission_probability` represents how likely Bob is to perform a certain activity on each day. If it is rainy, there is a 50% chance that he is cleaning his apartment; if it is sunny, there is a 60% chance that he is outside for a walk.



Alice talks to Bob three days in a row and discovers that on the first day he went for a walk, on the second day he went shopping, and on the third day he cleaned his apartment. Alice has a question: what is the most likely sequence of rainy/sunny days that would explain these observations? This is answered by the Viterbi algorithm.

```
# Helps visualize the steps of Viterbi.
def print_dptable(V):
    print "    ",
    for i in range(len(V)): print "%7s" % ("%d" % i),
    print
```

```
    for y in V[0].keys():
        print "%.5s: " % y,
        for t in range(len(V)):
            print "%.7s" % ("%f" % V[t][y]),
        print

def viterbi(obs, states, start_p, trans_p, emit_p):
    V = [{}]
    path = {}

    # Initialize base cases (t == 0)
    for y in states:
        V[0][y] = start_p[y] * emit_p[y][obs[0]]
        path[y] = [y]

    # Run Viterbi for t > 0
    for t in range(1,len(obs)):
        V.append({})
        newpath = {}

        for y in states:
            (prob, state) = max([(V[t-1][y0] * trans_p[y0][y] *
emit_p[y][obs[t]], y0) for y0 in states])
            V[t][y] = prob
            newpath[y] = path[state] + [y]

        # Don't need to remember the old paths
        path = newpath

    print_dptable(V)
    (prob, state) = max([(V[len(obs) - 1][y], y) for y in states])
    return (prob, path[state])
```

The function `viterbi` takes the following arguments: `obs` is the sequence of observations, e.g. `['walk', 'shop', 'clean']`; `states` is the set of hidden states; `start_p` is the start probability; `trans_p` are the transition probabilities; and `emit_p` are the emission probabilities. For simplicity of code, we assume that the observation sequence `obs` is non-empty and that `trans_p[i][j]` and `emit_p[i][j]` is defined for all states i,j.

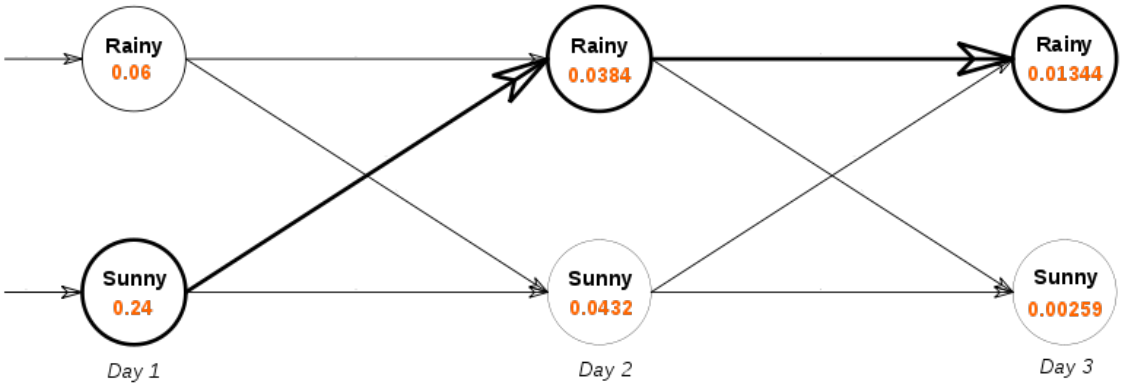
In the running example, the forward/Viterbi algorithm is used as follows:

```
def example():
    return viterbi(observations,
                   states,
                   start_probability,
                   transition_probability,
                   emission_probability)
print example()
```

This reveals that the observations `['walk', 'shop', 'clean']` were most likely generated by states `['Sunny', 'Rainy', 'Rainy']`, with score 0.01344 (to be

normalized). In other words, given the observed activities, it was most likely sunny when Bob went for a walk and then it started to rain the next day and kept on raining.

The operation of Viterbi's algorithm can be visualized by means of a trellis diagram. The Viterbi path is essentially the shortest path through this trellis. The trellis for the weather example is shown below; the corresponding Viterbi path is in bold:



When implementing Viterbi's algorithm, it should be noted that many languages use Floating Point arithmetic - as p is small, this may lead to underflow in the results. A common technique to avoid this is to take the logarithm of the probabilities and use it throughout the computation, the same technique used in the Logarithmic Number System. Once the algorithm has terminated, an accurate value can be obtained by performing the appropriate exponentiation.

Java implementation

```
import java.util.Hashtable;

public class Viterbi
{
        static final String RAINY = "Rainy";
        static final String SUNNY = "Sunny";

        static final String WALK = "walk";
        static final String SHOP = "shop";
        static final String CLEAN = "clean";

        public static void main(String[] args)
        {
                String[] states = new String[] {RAINY, SUNNY};

                String[] observations = new String[] {WALK, SHOP,
CLEAN};

                Hashtable<String, Float> start_probability = new
Hashtable<String, Float>();
                start_probability.put(RAINY, 0.6f);
                start_probability.put(SUNNY, 0.4f);

                // transition_probability
```

```
                Hashtable<String, Hashtable<String, Float>>
transition_probability =
                        new Hashtable<String, Hashtable<String,
Float>>();
                        Hashtable<String, Float> t1 = new
Hashtable<String, Float>();
                        t1.put(RAINY, 0.7f);
                        t1.put(SUNNY, 0.3f);
                        Hashtable<String, Float> t2 = new
Hashtable<String, Float>();
                        t2.put(RAINY, 0.4f);
                        t2.put(SUNNY, 0.6f);
                transition_probability.put(RAINY, t1);
                transition_probability.put(SUNNY, t2);

                // emission_probability
                Hashtable<String, Hashtable<String, Float>>
emission_probability =
                        new Hashtable<String, Hashtable<String,
Float>>();
                        Hashtable<String, Float> e1 = new
Hashtable<String, Float>();
                        e1.put(WALK, 0.1f);
                        e1.put(SHOP, 0.4f);
                        e1.put(CLEAN, 0.5f);
                        Hashtable<String, Float> e2 = new
Hashtable<String, Float>();
                        e2.put(WALK, 0.6f);
                        e2.put(SHOP, 0.3f);
                        e2.put(CLEAN, 0.1f);
                emission_probability.put(RAINY, e1);
                emission_probability.put(SUNNY, e2);

                Object[] ret = forward_viterbi(observations,
                           states,
                           start_probability,
                           transition_probability,
                           emission_probability);
                System.out.println(((Float) ret[0]).floatValue());
                System.out.println((String) ret);
                System.out.println(((Float) ret).floatValue());
        }

        public static Object[] forward_viterbi(String[] obs, String[]
states,
                        Hashtable<String, Float> start_p,
                        Hashtable<String, Hashtable<String, Float>>
trans_p,
                        Hashtable<String, Hashtable<String, Float>>
emit_p)
        {
                Hashtable<String, Object[]> T = new Hashtable<String,
Object[]>();
                for (String state : states)
                        T.put(state, new Object[] {start_p.get(state),
state, start_p.get(state)});
```

```
                for (String output : obs)
                {
                        Hashtable<String, Object[]> U = new
Hashtable<String, Object[]>();
                        for (String next_state : states)
                        {
                                float total = 0;
                                String argmax = "";
                                float valmax = 0;

                                float prob = 1;
                                String v_path = "";
                                float v_prob = 1;

                                for (String source_state : states)
                                {
                                        Object[] objs =
T.get(source_state);
                                        prob = ((Float)
objs[0]).floatValue();
                                        v_path = (String) objs;
                                        v_prob = ((Float)
objs).floatValue();

                                        float p =
emit_p.get(source_state).get(output) *

trans_p.get(source_state).get(next_state);
                                        prob *= p;
                                        v_prob *= p;
                                        total += prob;
                                        if (v_prob > valmax)
                                        {
                                                argmax = v_path + "," +
next_state;
                                                valmax = v_prob;
                                        }
                                }
                                U.put(next_state, new Object[] {total,
argmax, valmax});
                        }
                        T = U;
                }

                float total = 0;
                String argmax = "";
                float valmax = 0;

                float prob;
                String v_path;
                float v_prob;

                for (String state : states)
                {
                        Object[] objs = T.get(state);
                        prob = ((Float) objs[0]).floatValue();
                        v_path = (String) objs;
```

```
                        v_prob = ((Float) objs).floatValue();
                        total += prob;
                        if (v_prob > valmax)
                        {
                                argmax = v_path;
                                valmax = v_prob;
                        }
                }
                return new Object[]{total, argmax, valmax};
        }
}
```

Extensions

With the algorithm called iterative Viterbi decoding one can find the subsequence of an observation that matches best (on average) to a given HMM. Iterative Viterbi decoding works by iteratively invoking a modified Viterbi algorithm, reestimating the score for a filler until convergence.

An alternate algorithm, the Lazy Viterbi algorithm, has been proposed recently. This works by not expanding any nodes until it really needs to, and usually manages to get away with doing a lot less work (in software) than the ordinary Viterbi algorithm for the same result - however, it is not so easy to parallelize in hardware.

The Viterbi algorithm has been extended to operate with a deterministic finite automaton in order to quickly generate the trellis with state transitions pointing back at variable amount of history.

Chapter-13

Turbo Code

In information theory, **turbo codes** (originally in French *Turbocodes*) are a class of high-performance forward error correction (FEC) codes developed in 1993, which were the first practical codes to closely approach the channel capacity, a theoretical maximum for the code rate at which reliable communication is still possible given a specific noise level. Turbo codes are finding use in (deep space) satellite communications and other applications where designers seek to achieve reliable information transfer over bandwidth or latency constrained communication links in the presence of data-corrupting noise. Turbo codes are nowadays competing with LDPC codes, which provide similar performance.

History

Prior to turbo codes, the best constructions were serial concatenated codes based on an outer Reed-Solomon error correction code combined with an inner Viterbi-decoded short constraint length convolutional code, also known as RSV codes.

In 1993, turbo codes were introduced by Berrou, Glavieux, and Thitimajshima (from Telecom-Bretagne, former ENST Bretagne, France) in their paper: "*Near Shannon Limit Error-correcting Coding and Decoding: Turbo-codes*" published in the Proceedings of IEEE International Communications Conference. In a later paper, Berrou gave credit to the "intuition" of "G. Battail, J. Hagenauer and P. Hoeher, who, in the late 80s, highlighted the interest of probabilistic processing.". He adds "R. Gallager and M. Tanner had already imagined coding and decoding techniques whose general principles are closely related," although the necessary calculations were impractical at that time.

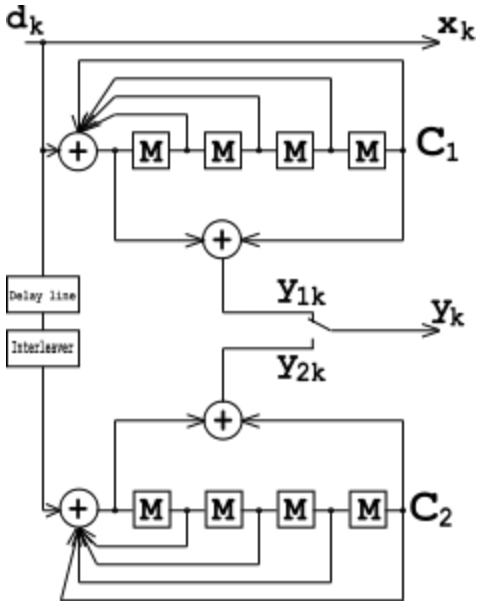
Since the introduction of the original parallel turbo codes in 1993, many other classes of turbo code have been discovered, including serial versions and repeat-accumulate codes. Iterative Turbo decoding methods have also been applied to more conventional FEC systems, including Reed-Solomon corrected convolutional codes.

An example encoder

There are many different instantiations of turbo codes, using different component encoders, input/output ratios, interleavers, and puncturing patterns. This example encoder implementation describes a 'classic' turbo encoder, and demonstrates the general design of parallel turbo codes.

This encoder implementation sends three sub-blocks of bits. The first sub-block is the *m*-bit block of payload data. The second sub-block is *n/2* parity bits for the payload data, computed using a recursive systematic convolutional code (RSC code). The third sub-block is *n/2* parity bits for a known permutation of the payload data, again computed using an RSC convolutional code. Thus, two redundant but different sub-blocks of parity bits are sent with the payload. The complete block has *m+n* bits of data with a code rate of *m/(m+n)*. The permutation of the payload data is carried out by a device called an interleaver.

Hardware-wise, this turbo-code encoder consists of two identical RSC coders, C_1 and C_2, as depicted in the figure, which are connected to each other using a concatenation scheme, called *parallel concatenation*:



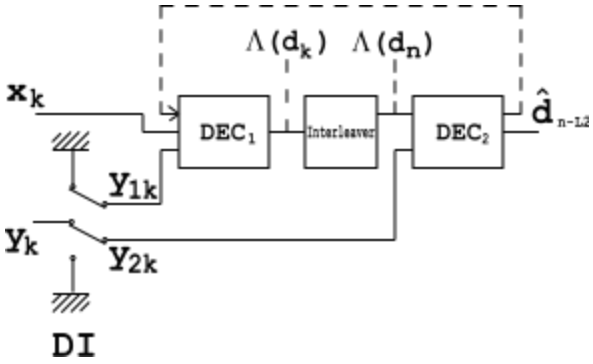
In the figure, *M* is a memory register. The delay line and interleaver force input bits d_k to appear in different sequences. At first iteration, the input sequence d_k appears at both outputs of the encoder, x_k and y_{1k} or y_{2k} due to the encoder's systematic nature. If the encoders C_1 and C_2 are used respectively in n_1 and n_2 iterations, their rates are respectively equal to

$$R_1 = \frac{n_1 + n_2}{2n_1 + n_2},$$

$$R_2 = \frac{n_1 + n_2}{2n_2 + n_1}.$$

The decoder

The decoder is built in the similar way as the above encoder - two elementary decoders are interconnected to each other, but in serial way, not in parallel. The DEC_1 decoder operates on lower speed (i.e. R_1), thus, it is intended for the C_1 encoder, and DEC_2 is for C_2 correspondingly. DEC_1 yields a soft decision which causes L_1 delay. The same delay is caused by the delay line in the encoder. The DEC_2's operation causes L_2 delay.



An interleaver installed between the two decoders is used here to scatter error bursts coming from DEC_1 output. *DI* block is a demultiplexing and insertion module. It works as a switch, redirecting input bits to DEC_1 at one moment and to DEC_2 at another. In OFF state, it feeds both y_{1k} and y_{2k} inputs with padding bits (zeros).

Consider a memoryless AWGN channel, and assume that at k-th iteration, the decoder receives a pair of random variables:

$$x_k = (2d_k - 1) + a_k,$$
$$y_k = 2(Y_k - 1) + b_k$$

where a_k and b_k are independent noise components having the same variance σ^2. Y_k is a k-th bit from y_k encoder output.

Redundant information is demultiplexed and sent through *DI* to DEC_1 (when $y_k = y_{1k}$) and to DEC_2 (when $y_k = y_{2k}$).

DEC_1 yields a soft decision, i.e.:

$$\Lambda(d_k) = \log \frac{p(d_k = 1)}{p(d_k = 0)}$$

and delivers it to DEC_2. $\Lambda(d_k)$ is called the *logarithm of the likelihood ratio* (LLR). $p(d_k = i), i \in \{0, 1\}$ is the *a posteriori probability* (APP) of the d_k data bit which shows the probability of interpreting a received d_k bit as i. Taking the *LLR* into account, DEC_2 yields a hard decision, i.e. a decoded bit.

It is known that the Viterbi algorithm is unable to calculate APP, thus it cannot be used in DEC_1. Instead of that, a modified BCJR algorithm is used. For DEC_2, the Viterbi algorithm is an appropriate one.

However, the depicted structure is not an optimal one, because DEC_1 uses only a proper fraction of the available redundant information. In order to improve the structure, a feedback loop is used.

Soft decision approach

The decoder front-end produces an integer for each bit in the data stream. This integer is a measure of how likely it is that the bit is a 0 or 1 and is also called *soft bit*. The integer could be drawn from the range [-127, 127], where:

- -127 means "certainly 0"
- -100 means "very likely 0"
- 0 means "it could be either 0 or 1"
- 100 means "very likely 1"
- 127 means "certainly 1"
- etc.

This introduces a probabilistic aspect to the data-stream from the front end, but it conveys more information about each bit than just 0 or 1.

For example, for each bit, the front end of a traditional wireless-receiver has to decide if an internal analog voltage is above or below a given threshold voltage level. For a turbo-code decoder, the front end would provide an integer measure of how far the internal voltage is from the given threshold.

To decode the $m+n$-bit block of data, the decoder front-end creates a block of likelihood measures, with one likelihood measure for each bit in the data stream. There are two parallel decoders, one for each of the $n/2$-bit parity sub-blocks. Both decoders use the sub-block of m likelihoods for the payload data. The decoder working on the second parity sub-block knows the permutation that the coder used for this sub-block.

Solving hypotheses to find bits

The key innovation of turbo codes is how they use the likelihood data to reconcile differences between the two decoders. Each of the two convolutional decoders generates a hypothesis (with derived likelihoods) for the pattern of m bits in the payload sub-block. The hypothesis bit-patterns are compared, and if they differ, the decoders exchange the

derived likelihoods they have for each bit in the hypotheses. Each decoder incorporates the derived likelihood estimates from the other decoder to generate a new hypothesis for the bits in the payload. Then they compare these new hypotheses. This iterative process continues until the two decoders come up with the same hypothesis for the *m*-bit pattern of the payload, typically in 15 to 18 cycles.

An analogy can be drawn between this process and that of solving cross-reference puzzles like crossword or sudoku. Consider a partially completed, possibly garbled crossword puzzle. Two puzzle solvers (decoders) are trying to solve it: one possessing only the "down" clues (parity bits), and the other possessing only the "across" clues. To start, both solvers guess the answers (hypotheses) to their own clues, noting down how confident they are in each letter (payload bit). Then, they compare notes, by exchanging answers and confidence ratings with each other, noticing where and how they differ. Based on this new knowledge, they both come up with updated answers and confidence ratings, repeating the whole process until they converge to the same solution.

Performance

Turbo codes perform well due to the attractive combination of the code's random appearance on the channel together with the physically realisable decoding structure. Turbo codes are affected by an error floor.

Practical applications using turbo codes

Telecommunications:

- Turbo codes are used extensively in 3G and 4G mobile telephony standards e.g. in HSPA, EV-DO and LTE.
- MediaFLO, terrestrial mobile television system from Qualcomm.
- The interaction channel of satellite communication systems, such as DVB-RCS.
- New NASA missions such as Mars Reconnaissance Orbiter now use turbo codes, as an alternative to RS-Viterbi codes.
- Turbo coding such as block turbo coding and convolutional turbo coding are used in IEEE 802.16 (WiMAX), a wireless metropolitan network standard.

Bayesian formulation

From an artificial intelligence viewpoint, turbo codes can be considered as an instance of loopy belief propagation in Bayesian networks.