Information Theory

Chapter III: Error control coding

Chapter structure

Chapter structure

1. **General presentation**
2. Analyzing linear block codes with the Hamming distance
3. Analyzing linear block codes with matrix algebra
4. Hamming codes
5. Cyclic codes

What is error control coding?

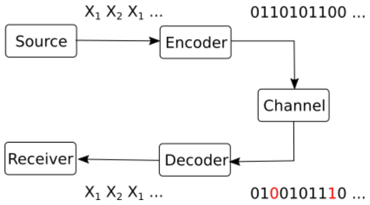

Figure 1: Communication system

- The second main task of coding: error control
- Protect information against channel errors

The need for error control coding

- In a transmission, the bits go through a **transmission channel**
  - The transmission channel is not ideal, it introduces some bit errors
  - Usually it is required that *all* bits are received correctly, no errors are allowed
- So what do to? **Error control coding**

Modelling the errors on the channel

- We consider only binary codes/ channels (symbols $= \{0, 1\}$)
- An **error** = a bit that has changed from 0 to 1 or viceversa while going through channel
- Errors can appear:
  - **independently**: sporadic errors, each bit has a random chance of error, independent of all the others
  - in **packets of errors**: groups of consecutive errors

Modelling the errors on the channel

- Changing the value of a bit = modulo-2 sum with 1
- Value of a bit remains the same = modulo-2 sum with 0

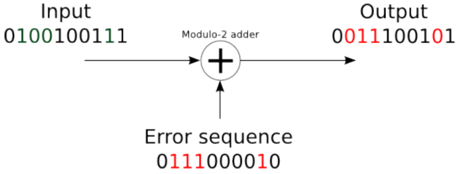

Figure 2: Channel error model

- Channel model we use (simple):
  - The transmitted sequence is summed modulo-2 with an **error sequence**

Modelling the errors on the channel

- Channel model we use (simple):
  - The transmitted sequence is summed modulo-2 with an **error sequence**
  - Error sequence has same length as the transmitted sequence
  - Where the error sequence is 1, there is a bit error
  - Where the error sequence is 0, there is no error

$$\mathbf{r} = \mathbf{c} \oplus \mathbf{e}$$

Mathematical properties of modulo-2 arithmetic

- Product is the same as for normal arithmetic
- Multiplication is distributive just like in normal case

$$a(b \oplus c) = ab \oplus ac$$

- Subtraction = addition. There is no negativation. Each number is its own negative

$$a \oplus a = 0$$

Error detection vs correction

What can we do about errors?

- **Error detection**: find out if there is any error in the received sequence
  - don't know exactly where, so cannot correct the bits, but can discard whole sequence
  - perhaps ask the sender to retransmit (examples: TCP/IP, internet communication etc)
  - easier to do
- **Error correction**: find out exactly which bits have errors, if any
  - locating the error = correcting error (for binary channels)
  - can correct all errored bits by inverting them
  - useful when can't retransmit (data is stored: on HDD, AudioCD etc.)
  - harder to do than mere detection

Overview of error control coding process

The process of error control:

1. Want to send a sequence of k bits = **information word**

$$\mathbf{i} = i_1 i_2 ... i_k$$

2. For each possible information word, the coder assigns a **codeword** of length $n > k$:

$$\mathbf{c} = c_1 c_2 ... c_n$$

3. The codeword is sent on the channel instead of the original information word

4. The receiver receives a sequence $\hat{\mathbf{c}} \approx \mathbf{c}$, with possible errors:

$$\hat{\mathbf{c}} = \hat{c}_1 \hat{c}_2 ... \hat{c}_n$$

5. The decoding algorithm detects/corrects the errors in $\hat{\mathbf{c}}$

Definitions

- An **error correcting code** is an association between the set of all possible information words to a set of codewords
  - Each possible information word $\mathbf{i}$ has a certain codeword $\mathbf{c}$
- The association can be done:
  - randomly: codewords are selected and associated randomly to the information words
  - based on a certain rule: the codeword is computed with some algorithm from the information word
- A code is a **block code** if it operates with words of *fixed size*
  - Size of information word $\mathbf{i} = k$, size of codeword $\mathbf{c} = n$, $n > k$
  - Otherwise it is a *non-block code*
- A code is **linear** if any linear combination of codewords is also a codeword

Definitions

- A code is called **systematic** if the codeword contains all the information bits explicitly, unaltered
  - coding merely adds supplementary bits besides the information bits
  - codeword has two parts: the information bits and the parity bits
  - example: parity bit added after the information bits
- Otherwise the code is called **non-systematic**
  - the information bits are not explicitly visible in the codeword
- The **coding rate** of a code is:

$$R = k/n$$

Definitions

- A code C is an t-**error-detecting** code if it is able to **detect** t or less errors
- A code C is an t-**error-correcting** code if it is able to **correct** t or less errors
- Examples: at blackboard

A first example: parity bit

- Add parity bit to a 8-bit long information word, before sending on a channel
  - coding rate $R = 8/9$
  - can detect 1 error in a 9-bit codeword
  - detection algorithm: check if parity bit matches data
  - fails for 2 errors
  - cannot correct error (don't know where it is located)
- Add more parity bits to be able to locate the error
  - Example at blackboard
  - coding rate $R = 8/12$
  - can detect and correct 1 error in a 9-bit codeword

A second example: repetition code

- Repeat same block of data n times
  - want to send a k-bit information word
  - codeword to send = the information word repeated $n = 5$ times
  - coding rate $R = k/n = 1/5$
  - can detect and correct 2 errors, and maybe even more if they do not affect the same bit
  - error correcting algorithm = majority rule
  - not very efficient

Redundancy

- Merriam-Webster: **“redundant”** definition:
  a. *exceeding what is necessary or normal : superfluous*
  b. *characterized by or containing an excess; specifically : using more words than necessary*
- Because $k < n$, error control coding introduces **redundancy**
  - to transmit k bits of information we actually send more bits (n)

Redundancy

- Error control coding adds redundancy, while source coding aims to reduce redundancy. Contradiction?
- No:
  - Source coding reduces existing redundancy from the data, which served no purpose
  - Error control coding adds redundancy **in a controlled way**, with a purpose
- Source coding and error control coding in practice: do sequentially, independently
  1. First perform source coding, eliminating redundancy in representation of data
  2. Then perform error control coding, adding redundancy for protection

Transmission channels preview

- In Chapter IV we will study Transmission Channels = mathematical model of how information is handled from the sender to the receiver
- Each channel has a certain **capacity** value = the maximum amount of information than can be sent over the channel
  - e.g. a channel may have capacity $C = 0.8$ bits
- More about this in Chapter IV

Shannon's noisy channel theorem (second theorem, channel coding theorem)

- A coding rate is called **achievable** for a channel if, for that rate, there exists a coding and decoding algorithm guaranteed to correct all possible errors on the channel

Shannon's noisy channel coding theorem (second theorem)

For a given channel, all rates below capacity $R < C$ are achievable. All rates above capacity, $R > C$, are not achievable.

Channel coding theorem explained

In layman terms:

- For all coding rates $R < C$, **there is a way** to recover the transmitted data perfectly (decoding algorithm will detect and correct all errors)
- For all coding rates $R > C$, **there is no way** to recover the transmitted data perfectly

Channel coding theorem example

- We send bits on a channel with capacity 0.7 bits/message
- For any coding rate $R < 0.7$ there exists an error correction code that allows fixing of all errors
  - $R < 0.7$ means we send more than 10 bits for every 7 information bits, on average
- With less than 10 bits for every 7 information bits $=>$ no code exists that can fix all errors
- The theorem makes it clear when it is possible to fix all errors, and guarantees that a code exists in this case

Ideas behind channel coding theorem

- The rigorous proof of the theorem is too complex to present
- Key ideas of the proof:
  - Use very long information words, $k \to \infty$
  - Use random codes, compute the probability of having error after decoding
  - If $R < C$, *in average for all possible codes*, the probability of error after decoding goes to 0
  - If the average for all codes goes to 0, there exists at least on code better than the average
  - That is the code we should use

Ideas behind channel coding theorem

- **The theorem does not tell what code to use**, only that some code exists
  - There is no clue of how to actually find the code in practice
  - Only some general principles:
    - using longer information words is better
    - random codewords are generally good
- In practice, we cannot use infinitely long codewords, so we will only get a *good enough* code

Chapter structure

1. General presentation
2. **Analyzing linear block codes with the Hamming distance**
3. Analyzing linear block codes with matrix algebra
4. Hamming codes
5. Cyclic codes

Distance between codewords

Practical ideas for error correcting codes:

- If a codeword $\mathbf{c_1}$ has errors and thus becomes identical to another codeword $\mathbf{c_2}$ $==>$ cannot detect any errors
  - Receiver will think it received a correct codeword $\mathbf{c_2}$, but actually it was $\mathbf{c_1}$
- We want codewords **as different as possible** from each other
- How to measure this difference? **Hamming distance**

Hamming distance

- The **Hamming distance** of two binary sequences **a**, **b** of length $n =$ the total number of bit differences between them

$$d_H(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{N} a_i \oplus b_i$$

- We need at least $d_H(a, b)$ bit changes to convert one sequence into another
- Example at blackboard

Hamming distance

- It satisfies the 3 properties of a metric function:
  1. $d_H(\mathbf{a}, \mathbf{b}) \geq 0 \quad \forall \mathbf{a}, \mathbf{b}$, with $d_H(\mathbf{a}, \mathbf{b}) = 0 \Leftrightarrow \mathbf{a} = \mathbf{b}$
  2. $d_H(\mathbf{a}, \mathbf{b}) = d_H(\mathbf{b}, \mathbf{a}), \forall \mathbf{a}, \mathbf{b}$
  3. $d_H(\mathbf{a}, \mathbf{c}) \leq d_H(\mathbf{a}, \mathbf{b}) + d_H(\mathbf{b}, \mathbf{c}), \forall \mathbf{a}, \mathbf{b}, \mathbf{c}$
- The **minimum Hamming distance of a code**, $d_{H\,min}$ = the minimum Hamming distance between any two codewords $\mathbf{c_1}$ and $\mathbf{c_2}$

Nearest-neighbor decoding scheme

Coding:

- Design a code with large $d_{H\,min}$
- Send a codeword **c** of the code

Decoding:

- Receive a word **r**, that may have errors
- Error detecting:
  - check if r is part of the codewords of the code C:
  - if r is part of the code, decide that there have been no errors
  - if r is not a codeword, decide that there have been errors
- Error correcting:
  - if **r** is a codeword, decide there are no errors
  - else, choose codeword **nearest** to the received **r**, in terms of Hamming distance
  - this is known as **nearest-neighbor decoding**

Performance of nearest neighbor decoding

Theorem:

- If the minimum Hamming distance of a code is $d_{H\,min}$, then:
  1. the code can *detect* up to $d_{H\,min} - 1$ errors
  2. the code can *correct* up to $\left\lfloor \frac{d_{H\,min}-1}{2} \right\rfloor$ errors using nearest-neighbor decoding

Consequence:

- It is good to have $d_{H\,min}$ as large as possible
  - This implies longer codewords, i.e. smaller coding rate, i.e. more redundancy

Performance of nearest neighbor decoding

Proof:

1. at least d_{Hmin} binary changes are needed to change one codeword into another, $d_{Hmin} - 1$ is not enough $=>$ the errors are detected
2. the received word $\mathbf{r}$ is closer to the original codeword than to any other codeword $=>$ nearest-neighbor algorithm will find the correct one
   - because $\left\lfloor \frac{d_{Hmin}-1}{2} \right\rfloor$ = less than half the distance to another codeword

Note: if the number of errors is higher, can fail:

- Detection failure: decide that there were no errors, even if they were (more than $d_{Hmin} - 1$)
- Correction failure: choose a wrong codeword

Example: blackboard

Computational complexity

- **Computational complexity** $=$ the amount of computational resources required by an algorithm
  - only refers to the **order of magnitude of the dominant term**
    - neglects the other terms
    - neglects actual coefficient values in front
- Computational complexity with respect to number of information bits k, of the search-based nearest neighbor decoding (as presented earlier), is

$$\mathcal{O}(k) = 2^k$$

- Proof: Requires comparing with all codewords, and there are 2^k codewords in total

Computational complexity

- This implementation is **very inefficient**
  - k doubles $=>$ the amount of computations is squared
  - k increases 10 times $=>$ computations are raised to a power of 10
  - k increases 100 times $=>$ computations are raised to a power of 1000
  - for $k = 256$ you'd need all the energy of the Sun

- Need to find ways to make it simpler

Chapter structure

1. General presentation
2. Analyzing linear block codes with the Hamming distance
3. **Analyzing linear block codes with matrix algebra**
4. Hamming codes
5. Cyclic codes

Review of basic algebra

Informal definitions:

- **Vector space** = a set such that:
  a. one element + another element = still an element from the set
  b. one element $\times$ a constant = still an element from the set
  - Examples: Euclidian vector spaces: a line, points in 2D, 3D
  - Elements are called "vectors"
- **Basis** = a set of *n* independent vectors $\mathbf{e_1}, \ldots \mathbf{e_n}$
  - Any vector $\mathbf{v}$ can be expressed as a linear combination of the basis elements

$$\mathbf{v} = \mathbf{e_1} \cdot \alpha_1 + \ldots + \mathbf{e_n} \cdot \alpha_n$$

Review of basic algebra

- **Subspace** = a smaller dimensional vector space inside a larger vector space
  - Examples: a line in a plane
    - sum of two vectors on a line = still on the line
    - size of subspace = 1
    - size of larger space = 2
  - A plane in 3D space
    - sum of two vectors from the plane = still on the plane
    - size of subspace = 2
    - size of larger space = 3

Binary sequences form a vector space

- The set of all binary sequences of size n is a vector space of size n
  - sum of two sequences of size n is still a sequence of size n
- The sum operation = modulo-2 sum $\oplus$
- Multiplication with 0 and 1 = as in usual arithmetic

How to look at matrix-vector multiplications

- Matrix-vector multiplication
  - Output vector = linear combination of the matrix columns
- Vector-matrix multiplication
  - Output vector = linear combination of the matrix rows
- Explain at the blackboard, draw picture

How to look at matrix-vector multiplications

- Vector spaces can be perfectly described with matrix-vector multiplications
  - Matrix columns/rows = elements of the basis
  - The output vector = the vector
  - The multiplicated vector = the coefficients of the linear combination
- Any vector $\mathbf{v}$ can be expressed as a linear combination of the basis elements

$$\mathbf{v} = \mathbf{e_1} \cdot \alpha_1 + ... + \mathbf{e_n} \cdot \alpha_n$$

$$\mathbf{v} = \begin{bmatrix} \mathbf{e_1} & \mathbf{e_2} & ... & \mathbf{e_n} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ ... \\ \alpha_n \end{bmatrix}$$

- $\mathbf{e_1}, ...\mathbf{e_n}$ are **column vectors**
- Equation can be transposed => all vectors become row vectors

Codewords form a vector space

- The set of all binary codewords of a linear block code is a vector subspace of dimension k
- Proof:
  - code is linear $=>$ because sum (XOR) of two codewords is still a codeword
  - codeword $\times$ a constant (0 or 1) $=>$ still a codeword
  - total number of codewords is 2^k $=>$ dimension is k
- Length of codewords is n, but size of space is k $=>$ they form a **subspace** of the larger space of all binary sequences of length n

Codewords form a vector space

- Since all codewords form a (sub)space $=>$ all codewords can be expressed as matrix-vector multiplications
- Need to find a basis for the codewords

Generator matrix

- All codewords for a linear block code can be generated via a **matrix-vector multiplication**:

$$\mathbf{i} \cdot [G] = \mathbf{c}$$

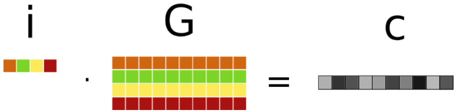

Figure 3: Codeword construction with generator matrix

- $[G]$ = **generator matrix** of size $k \times n$ ("fat" matrix, $k < n$)
  - it is fixed, it fully defines the whole code

Generator matrix

- Row-wise interpretation:
  - Any codeword $\mathbf{c}$ = a linear combination of rows in $[G]$
  - The rows of $[G]$ = a *basis* for the linear block code
  - Could also be transposed, i.e. use column vectors instead
- All operations are done in modulo-2 arithmetic
- There exists a separate codeword for all possible information words $\mathbf{i}$

Proof of linearity

- Prove that a codeword + another codeword = also codeword:

$$\mathbf{i_1} \cdot [G] = \mathbf{c_1}$$

$$\mathbf{i_2} \cdot [G] = \mathbf{c_2}$$

$$\mathbf{c_1} \oplus \mathbf{c_2} = (\mathbf{i_1} \oplus \mathbf{i_2}) \cdot [G] = \mathit{codeword}$$

Parity check matrix

- Every generator matrix $[G]$ has a related **parity-check matrix** $[H]$ such that
$$\mathbf{0} = [H] \cdot [G]^T$$
  - also known as **control matrix**
  - size of $[H]$ is $(n-k) \times n$
  - $[G]$ and $[H]$ are related, one can be deduced from the other
- $[H]$ is very useful to check if a binary word is a codeword or not (i.e. for nearest neighbor error detection)

Using the parity check matrix

- Theorem: every codeword $\mathbf{c}$ generated with $[G]$ ($\mathbf{i} \cdot [G] = \mathbf{c}$) will produce a 0 vector when multiplied with the corresponding $[H]$ matrix:

$$\mathbf{0} = [H] \cdot \mathbf{c}^T$$

- Proof:

$$\mathbf{i} \cdot [G] = \mathbf{c}$$
$$[G]^T \cdot \mathbf{i}^T = \mathbf{c}^T$$
$$[H] \cdot \mathbf{c}^T = [H] \cdot [G]^T \cdot \mathbf{i}^T = \mathbf{0}$$

- All codewords generated with $[G]$ will produce 0 when multiplied with $[H]$
- All binary sequences that are not codewords will produce $\neq 0$ when multiplied with $[H]$

Relation between [G] and [H]

- [G] and [H] are related
  - The codewords form a k-dimensional subspace inside the larger n-dimensional vector space
  - The rows of $[H]$ are the "missing" dimensions of the subspace (the "orthogonal complement")
- Together $[G]$ and $[H]$ form a full square matrix $n \times n$, which is a basis for the full n-dimensional vector space
  - size of $[H]$ is $(n-k) \times n$
  - size of $[G]$ is $k \times n$
- Examples:
  - line in a 2D plane, has one orthogonal dimension
  - plane in 3D space, has one orthogonal dimension
  - line in 3D space, has 2 orthogonal dimension

Standard [G] and [H] for systematic codes

- For systematic codes, [G] and [H] have special forms (known as **"standard"** forms)
- Generator matrix
  - first part = identity matrix
  - second part = some matrix Q

$$[G]_{k\times n} = [I_{k\times k} \;\; Q_{k\times(n-k)}]$$

- Parity-check matrix
  - first part = same Q, but **transposed**
  - second part = identity matrix

$$[H]_{(n-k)\times n} = [Q^T_{(n-k)\times k} \;\; I_{(n-k)\times(n-k)}]$$

- Can easily compute one from the other
- Example at blackboard

Interpretation as parity bits

- Multiplication with G in standard form produces the codeword as
  - first part = information bits (since first part of $[G]$ is identity matrix)
  - additional bits = combinations of information bits = *parity bits*
- The additional bits added by coding are actually just parity bits
  - Proof: write the generation equations (example)
- Parity-check matrix in standard form $[H]$ checks if parity bits correspond to information bits
  - Proof: write down the parity check equation (see example)
- If all parity bits match the data, the result of multiplying with $[H]$ is 0
  - otherwise it is $\neq 0$

Interpretation as parity bits

- Generator & parity-check matrices are just mathematical tools for easy computation and checking of parity bits
- We're still just computing and checking parity bits, but we do it easier with matrices

Syndrome

- Nearest neighbor error detection = check if received word **r** is a codeword
- We do this easily by multiplying with $[H]$
- The resulting vector $z = [H] \cdot [r]^T$ is known as **syndrome**
- Column-wise interpretation of multiplication:

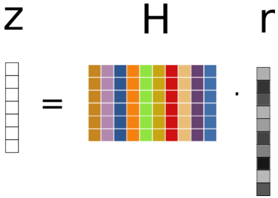

Figure 4: Codeword checking with parity-check matrix

Nearest neighbor error detection with matrices

Nearest neighbor error **detection** with matrices:

1. generate codewords with generator matrix:

$$\mathbf{i} \cdot [G] = \mathbf{c}$$

2. send codeword $\mathbf{c}$ on the channel
3. a random error word $\mathbf{e}$ is applied on the channel
4. receive word $\mathbf{r} = \mathbf{c} \oplus \mathbf{e}$
5. compute **syndrome** of $\mathbf{r}$:

$$\mathbf{z} = [H] \cdot \mathbf{r}^T$$

6. Decide:
   - If $\mathbf{z} = 0 => \mathbf{r}$ has no errors
   - If $\mathbf{z} \neq 0 => \mathbf{r}$ has errors

Nearest neighbor error correction with matrices

Nearest neighbor error **correction** with matrices:

- Syndrome $\mathbf{z} \neq 0 => \mathbf{r}$ has errors, we need to locate them
- The syndrome is the effect only of the error word:

$$\mathbf{z} = [H] \cdot \mathbf{r}^T = [H] \cdot (\mathbf{c}^T \oplus \mathbf{e}^T) = [H] \cdot \mathbf{e}^T$$

7. Create a **syndrome lookup table**:
   - for every possible error word $\mathbf{e}$, compute the syndrome $\mathbf{z} = [H] \cdot \mathbf{e}^T$
   - start with error words with 1 error (most likely), then with 2 errors (less likely), and so on
8. Locate the syndrome $\mathbf{z}$ in the table, read the corresponding error word $\widehat{\mathbf{e}}$
9. Find the correct word:
   - adding the error word again will invert the errored bits back to the originals

$$\widehat{\mathbf{c}} = \mathbf{r} \oplus \widehat{\mathbf{e}}$$

Example

Example: at blackboard

Computational complexity

- Computational complexity for error detection
  - Error detection $=$ multiplication with $[H]$
  - Complexity is $\mathcal{O}(n^2)$ (size of $[H]$ is $(n-k) \times n$
  - Much more efficient!
- Computational complexity for error correction
  - Need to check all possible error words $=>$ bad performance
  - In practice, other tricks are used to make it much faster (see Hamming codes for example)

Conditions on [H] for error detection and correction

- How to design a good matrix $[H]$?
- Conditions on [H] for successful error **detection**:
  - We can detect errors if the syndrome is **non-zero**
  - To detect a single error: every column of $[H]$ must be non-zero
  - To detect two error: sum of any two columns of $[H]$ cannot be zero
    - that means all columns are different
  - To detect n errors: sum of any n or less columns of $[H]$ cannot be zero

Conditions on [H] for error detection and correction

- Conditions for syndrome-based error **correction**:
  - We can correct errors if the syndrome is **unique**
  - To correct a single error: all columns of $[H]$ are different
    - so the syndromes, for a single error, are all different
  - To correct n errors: sum of any n or less columns of $[H]$ are all different
    - much more difficult to obtain than for decoding
- Conditions for error correction are more demanding than for detection
- Note: Rearranging the columns of $[H]$ (the order of bits in the codeword) does not affect performance

Chapter structure

Chapter structure

1. General presentation
2. Analyzing linear block codes with the Hamming distance
3. Analyzing linear block codes with matrix algebra
4. **Hamming codes**
5. Cyclic codes

Hamming codes

- A particular class of linear error-correcting codes
- Definition: a **Hamming code** is a linear block code where the columns of $[H]$ are *the binary representation of all numbers from 1 to* $2^r - 1, \forall r \geq 2$
- Example (blackboard): (7,4) Hamming code
- Systematic: arrange the bits in the codeword, such that the control bits correspond to the columns having a single 1
  - no big difference from the usual systematic case, just a rearrangement of bits
  - makes implementation easier
- Example codeword for Hamming(7,4):

$$c_1 c_2 i_3 c_4 i_5 i_6 i_7$$

Properties of Hamming codes

- From definition of $[H]$ it follows:
  1. Codeword has length $n = 2^r - 1$
  2. r bits are parity bits (also known as ***control bits***)
  3. $k = 2^r - r - 1$ bits are information bits

- Notation: **(n,k) Hamming code**
  - n = codeword length $= 2^r - 1$,
  - k = number of information bits $= 2^r - r - 1$
  - Example: (7,4) Hamming code, (15,11) Hamming code, $(127, 120)$ Hamming code

Properties of Hamming codes

- Can detect two errors
  - All columns are different => can detect 2 errors
  - Sum of two columns equal to a third => cannot correct 3

OR

- Can correct one error
  - All columns are different => can correct 1 error
  - Sum of two columns equal to a third => cannot correct 2
  - Non-systematic: syndrome = error position

BUT

- Not simultaneously!
  - same non-zero syndrome can be obtained with 1 or 2 errors, can't distinguish

Coding rate of Hamming codes

Coding rate of a Hamming code:

$$R = \frac{k}{n} = \frac{2^r - r - 1}{2^r - 1}$$

The Hamming codes can correct 1 OR detect 2 errors in a codeword of size n

- (7,4) Hamming code: $n = 7$
- (15,11) Hamming code: $n = 15$
- (31,26) Hamming code: $n = 31$

Longer Hamming codes are progressively weaker:

- weaker error correction capability
- better efficiency (higher coding rate)
- more appropriate for smaller error probabilities

Encoding & decoding example for Hamming(7,4)

See whiteboard.

In this example, encoding is done without the generator matrix G, directly with the matrix H, by finding the values of the parity bits c_1, c_2, c_4 such that

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = [H] \begin{bmatrix} c_1 \\ c_2 \\ i_3 \\ c_4 \\ i_5 \\ i_6 \\ i_7 \end{bmatrix}$$

For a single error, the syndrome **is the binary representation of the location of the error**.

Circuit for encoding Hamming(7,4)

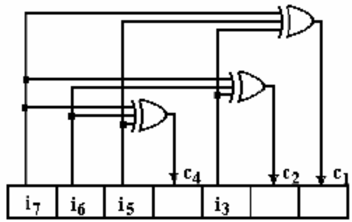

Figure 5: Hamming Encoder

- Components:
  - A **shift register** to hold the codeword
  - Logic OR gates to compute the parity bits

Circuit for decoding Hamming(7,4)

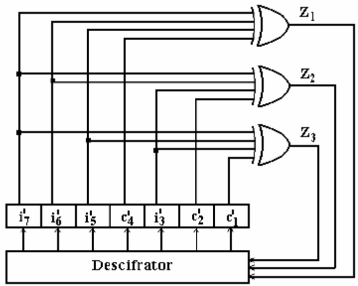

Figure 6: Hamming Encoder

- Components:
  - A **shift register** to hold the received word
  - Logic OR gates to compute the bits of the syndrome (z_i)
  - **Binary decoder**: activates the output corresponding to the binary input value, fixing the error

SECDED Hamming codes

- Hamming codes can correct 1 error OR can detect 2 errors, but we cannot differentiate the two cases
- Example:
  - the syndrome $\mathbf{z} = [H] \cdot \mathbf{r}^T = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$ can be caused by:
    - a single error in location 3 (bit i_3)
    - two errors in location 1 and 2 (bits c_1, bits c_2)
  - if we know it is a single error, we can go ahead and correct it, then use the corrected data
  - if we know there are two errors, we should NOT attempt to correct them, because we cannot locate the errors correctly
- Unfortunately, it is **not possible to differentiate** between the two cases.
- **Solution?** Add additional parity bit $\rightarrow$ SECDED Hamming codes

SECDED Hamming codes

- Add an additional parity bit to differentiate the two cases
  - c_0 = sum of all n bits of the codeword
- For (7,4) Hamming codes:

$$\mathbf{c_0} c_1 c_2 i_3 c_4 i_5 i_6 i_7$$

- The parity check matrix is extended by 1 row and 1 column

$$\tilde{H} = \begin{bmatrix} 1 & 1 \\ 0 & \mathbf{H} \end{bmatrix}$$

- Known as SECDED Hamming codes
  - **S**ingle **E**rror **C**orrection - **D**ouble **E**rror **D**etection

Encoding and decoding of SECDED Hamming codes

- Encoding:
  - compute codeword using $\tilde{H}$
  - alternatively, prepend $\mathbf{c_0}$ = sum of all other bits

Encoding and decoding of SECDED Hamming codes

- Decoding
  - Compute syndrome of the received word using $\tilde{H}$

$$\tilde{\mathbf{z}} = \begin{bmatrix} z_0 \\ \mathbf{z} \end{bmatrix} = [\tilde{H}] \cdot \mathbf{r}^T$$

  - z_0 is an additional bit in the syndrome corresponding to c_0
  - z_0 tells us whether the received c_0 matches the parity of the received word
    - $z_0 = 0$: the additional parity bit c_0 matches the parity of the received word
    - $z_0 = 1$: the additional parity bit c_0 does not match the parity of the received word

Encoding and decoding of SECDED Hamming codes

- Decoding (continued):
  - Decide which of the following cases happened:
    - If no error happened: $z_1 = z_2 = z_3 = 0, z_0 = \forall$
    - If 1 error happened: syndrome is non-zero, $z_0 = 1$ (does not match)
    - If 2 errors happened: syndrome is non-zero, $z_0 = 0$ (does match, because the two errors cancel each other out)
    - If 3 errors happened: same as 1, can't differentiate
- Now can simultaneously differentiate between:
  - 1 error: $\rightarrow$ perform correction
  - 2 errors: $\rightarrow$ detect, but do not perform correction
- Also, if correction is never attempted, can detect up to 3 errors
  - minimum Hamming distance $= 4$ (no proof given)
  - don't know if 1 error, 2 errors or 3 errors, so can't try correction

Summary until now

- Systematic codes: information bits + parity bits
- Generator matrix: use to generate codeword

$$\mathbf{i} \cdot [G] = \mathbf{c}$$

- Parity-check matrix: use to check if a codeword

$$0 = [H] \cdot \mathbf{c}^T$$

- Syndrome:

$$\mathbf{z} = [H] \cdot \mathbf{r}^T$$

- Syndrome-based error detection: syndrome non-zero
- Syndrome-based error correction: lookup table
- Hamming codes: $[H]$ contains all numbers $1 ... 2^r - 1$
- SECDED Hamming codes: add an extra parity bit

Chapter structure

Chapter structure

1. General presentation
2. Analyzing linear block codes with the Hamming distance
3. Analyzing linear block codes with matrix algebra
4. Hamming codes
5. **Cyclic codes**

Cyclic codes

Definition: **cyclic codes** are a particular class of linear block codes for which *every cyclic shift of a codeword is also a codeword*

- Cyclic shift: cyclic rotation of a sequence of bits (any direction)
- Are a particular class of linear block codes, so all the theory up to now still applies
  - they have a generator matrix, parity check matrix etc.
- But they can be implemented more efficient than general linear block codes (e.g. Hamming)
- Used **everywhere** under the common name **CRC** (**C**yclic **R**edundancy **C**heck)
  - Network communications (Ethernet), data storage in Flash memory

Usage example: Ethernet frame

- CRC codes are used in Ethernet frames:

802.3 Ethernet packet and frame structure

Layer	Preamble	Start of frame delimiter	MAC destination	MAC source	802.1Q tag (optional)	Ethertype (Ethernet II) or length (IEEE 802.3)	Payload	Frame check sequence (32-bit CRC)	Interpacket gap
	7 octets	1 octet	6 octets	6 octets	(4 octets)	2 octets	46–1500 octets	4 octets	12 octets
Layer 2 Ethernet frame			← 64–1522 octets →						
Layer 1 Ethernet packet & IPG	← 72–1530 octets →								← 12 oct. →

Figure 7: CRC value in an Ethernet frame

Binary polynomials

- Every binary sequence **a** corresponds to a polynomial $\mathbf{a}(\mathbf{x})$ with binary coefficients

$$a_0 a_1 ... a_{n-1} \rightarrow \mathbf{a}(\mathbf{x}) = a_0 \oplus a_1 x \oplus ... \oplus a_{n-1} x^{n-1}$$

- Example:

$$10010111 \rightarrow 1 \oplus x^3 \oplus x^5 \oplus x^6 \oplus x^7$$

- From now on, by "codeword" we also mean the corresponding polynomial.
- Can perform all mathematical operations with these polynomials:
  - addition, multiplication, division etc. (examples)
- There are efficient circuits for performing multiplications and divisions.

Generator polynomial

Theorem:

- All the codewords of a cyclic code are multiples of a certain polynomial $g(x)$, known as **generator polynomial**.

Properties of generator polynomial

The generator polynomial $g(x)$ must satisfy the following:

- $g(x)$ must have first and last coefficient equal to 1
- $g(x)$ must be a factor of $X^n \oplus 1$
- The *degree* of $g(x)$ is $n - k$, where:
  - n = the size of codeword (codeword polynomial has degree $n - 1$)
  - k = the size of the information word (information polynomial has degree $k - 1$)

$$(k-1) + (n-k) = n - 1$$

- **The degree of $g(x)$ is the number of parity bits of the code.**

Example of generator polynomials

Example:

$$1 \oplus x^7 = (1 \oplus x)(1 \oplus x + \oplus x^3)(1 \oplus x^2 \oplus x^3)$$

Each factor can generate a code:

- $1 \oplus x$ generates a (7,6) cyclic code
- $1 \oplus x \oplus x^3$ generates a (7,4) cyclic code
- $1 \oplus x^2 \oplus x^3$ generates a (7,4) cyclic code

Popular polynomials

```
Some popular polys are:
16 bits: (16,12,5,0)                                 [X25 standard]
         (16,15,2,0)                                 ["CRC-16"]
32 bits: (32,26,23,22,16,12,11,10,8,7,5,4,2,1,0)     [Ethernet]
```

Figure 8: Popular generator polynomials $g(x)$

- Image from *http://www.ross.net/crc/download/crc_v3.txt*
- Your turn: write the polynomials in mathematical form

Proving the cyclic property

Theorem:

- Any cyclic shift of a codeword is also a codeword.

Proof:

- It is enough to consider a cyclic shift by 1 position
- Original codeword

$$c_0 c_1 c_2 \ldots c_{n-1} \rightarrow \mathbf{c}(\mathbf{x}) = c_0 \oplus c_1 x \oplus \ldots \oplus c_{n-1} x^{n-1}$$

- Cyclic shift to the right by 1 position

$$c_{n-1} c_0 c_1 \ldots c_{n-2} \rightarrow \mathbf{c}'(\mathbf{x}) = c_{n-1} \oplus c_0 x \oplus \ldots \oplus c_{n-2} x^{n-1}$$

- We can rewrite:

$$\begin{aligned} \mathbf{c}'(\mathbf{x}) &= x \cdot \mathbf{c}(\mathbf{x}) \oplus c_{n-1} x^n \oplus c_{n-1} \\ &= x \cdot \mathbf{c}(\mathbf{x}) \oplus c_{n-1}(x^n \oplus 1) \end{aligned}$$

Proving the cyclic property

Proof (continued):

- Since $\mathbf{c}(\mathbf{x})$ is a multiple of $g(x)$, so is $x \cdot \mathbf{c}(\mathbf{x})$
- Also $(x^n \oplus 1)$ is always a multiple of $g(x)$
- $=>$ It follows that their sum $\mathbf{c}'(\mathbf{x})$ is a also a multiple of $g(x)$, which means it is a codeword.

QED

- Note that we relied on two properties mentioned before:
  - that a codeword $\mathbf{c}(\mathbf{x})$ is always a multiple of $g(x)$
  - that $g(x)$ is a factor of $(x^n \oplus 1)$

Coding and decoding of cyclic codes

- Cyclic codes can be used for detection or correction
- In practice, they are used mostly for **detection only** (e.g. in Ethernet)
  - because there are other codes with better performance for correection
- Can be systematic / non-systematic
  - In practice, the systematic variant is much preferred
- We study coding/decoding from 3 perspectives:
  - The mathematical way, with polynomials
  - The programming way, e.g. as a programming algorithm
  - The hardware way, via schematics

1. Coding and decoding - The mathematical way

Reminder: polynomial multiplication and division

- Two polynomials $a(x)$ and $b(x)$ can be multiplied
  - the result has degree = degree of $a(x)$ + degree of $b(x)$
- A polynomials $a(x)$ can be divided to another polynomial $b(x)$:

$$a(x) = b(x)q(x) \oplus r(x)$$

  - $q(x)$ = the quotient ("câtul")
  - $r(x)$ = the remainder ("restul")
  - the degree of $r(x)$ is strictly smaller than the degree of $b(x)$

1. Coding and decoding - The mathematical way

Coding

- We want to encode the **information word** with k bits

$$i_0 i_1 i_2 \ldots i_{k-1} \rightarrow i(x) = i_0 \oplus i_1 x \oplus \ldots \oplus i_{k-1} x^{k-1}$$

- **Non-systematic** codeword generation:

$$\boxed{c(x) = i(x) \cdot g(x)}$$

- The degrees match:
  - $i(x)$ has degree $k-1$ (k bits)
  - $g(x)$ has degree $n-k$ ($n-k+1$ bits)
  - $c(x)$ has degree $n-1 = (n-k)+(k-1)$ (n bits)

Systematic coding - The mathematical way

- **Systematic** codeword generation:

$$\boxed{c(x) = x^{n-k} \cdot i(x) \oplus b(x)}$$

- $b(x)$ is the remainder of dividing $x^{n-k}i(x)$ to $g(x)$:

$$x^{n-k}i(x) = a(x)g(x) \oplus b(x)$$

  - $b(x)$ is known as "the CRC value"

- Is this $c(x)$ really a multiple of $g(x)$? Yes, because:

$$c(x) = x^{n-k} \cdot i(x) \oplus b(x) = a(x)g(x) \oplus b(x) \oplus b(x) = a(x)g(x)$$

Interpretation

- Why is the code systematic?
- Let's analyze the systematic codeword generation step by step
- Consider the information word/polynomial

$$\mathbf{i} = [\underbrace{i_0 i_1 \ldots i_{k-1}}_{k}] \rightarrow i(x) = i_0 \oplus i_1 x \oplus \ldots \oplus i_{k-1} x^{k-1}$$

- Multiplying $x^{n-k} \cdot i(x)$ shifts all bits to the right with $(n-k)$ positions

$$[\underbrace{00 \ldots 0}_{n-k} \underbrace{i_0 i_1 \ldots i_{k-1}}_{k}] \rightarrow i(x) = i_0 x^{n-k} \oplus i_1 x^{n-k+1} \oplus \ldots \oplus i_{k-1} x^{n-1}$$

Interpretation (continued)

- The remainder $b(x)$ has degree strictly less than $n-k$ (degree of $g(x)$), so at most $n-k$ bits
- Therefore adding $b(x)$ will not overlap with $x^{n-k} \cdot i(x)$
  - the $(n-k)$ bits of $b(x)$ will fit in the first $n-k$ locations

$$\mathbf{c} = [\underbrace{b_0 b_1 \ldots b_{n-k}}_{n-k} \underbrace{i_0 i_1 \ldots i_{k-1}}] \rightarrow$$

$$\rightarrow c(x) = b_0 \oplus b_1 x \oplus \ldots \oplus b_{n-k-1} x^{n-k-1} \oplus i_0 x^{n-k} \oplus i_1 x^{n-k+1} \oplus \ldots \oplus i_k$$

- Hence the code is systematic: the information bits are in the codeword
- The code adds $b(x)$ (the remainder) = the **CRC value**

Interpretation

- Systematic cyclic codeword = compute a CRC value and append it to the data
- Different writing conventions:
  - when writing the codewords from LSB -> MSB (increasing order of degrees), the CRC appears in front
    - like in lecture slides
  - when writing the codewords from MSB -> LSB (decreasing order of degrees), the CRC appears at the end
    - like in laboratory
  - same thing, just bit ordering is reversed
  - (LSB = Least Significant Bit, MSB = Most Significant Bit)

Decoding - The mathematical way

Decoding

- We receive $\mathbf{r} = r_0 r_1 r_2 ... r_{n-1} \rightarrow \mathbf{r(x)} = r_0 \oplus r_1 x \oplus ... \oplus r_{n-1} x^{n-1}$\$
- Error **detection**: check if $r(x)$ is a codeword or not
- Check if the received $\mathbf{r(x)}$ still is a multiple of $g(x)$
  - Divide $\mathbf{r(x)}$ to $g(x)$:
    - If remainder of $r(x) : g(x)$ is 0 => it is a codeword, no errors present
    - If remainder is non-zero => it's not a true codeword, **errors detected**
- Computing the remainder = computing the CRC of the received data
  - Remember lab: decoding = compute CRC of all coded data, if 0 => OK, if non-zero => NOK

Decoding - The mathematical way

- Error **correction**: use a lookup table (just like with matrices)
  - build a lookup table for all possible error words (like with matrix codes)
  - for each error code, divide by $g(x)$ and compute the remainder
  - when the remainder is identical to the remainder obtained with $\mathbf{r}(\mathbf{x})$, we found the error word $=>$ correct errors
- Example: at blackboard

2. Coding and decoding - The programming way

- Only for systematic codes (mostly used)
- Steps:
  - 1. Compute the CRC $= b(x) =$ remainder of $x^{n-k} i(x)$ divided to $g(x)$
  - 2. Put the CRC in front of the information word, mirrored
- Good reference: *"A Painless Guide to CRC Error Detection Algorithms"*, Ross N. Williams
  - http://www.ross.net/crc/download/crc_v3.txt

Coding

- The mathematical polynomial division = just like XOR-ing succesively with $g(x)$
  - align the binary sequence of $g(x)$ under the leftmost 1
  - XOR the sequences
  - repeat
  - just like in the lab
- See example at blackboard / lab

Example

```
 ______________
 11010110110000
 10011,,.,,....
 -----,,.,,....
  10011,.,,....
  10011,.,,....
  -----,.,,....
   00001.,,....
   00000.,,....
   -----.,,....
    00010,,....
    00000,,....
    -----,,....
     00101,....
     00000,....
     -----,....
      01011....
      00000....
      -----....
       10110...
       10011...
```

Figure 9: Polynomial division = XORing succesively with $g(x)$

Decoding

- We receive $\mathbf{r} = r_0 r_1 r_2 ... r_{n-1} \rightarrow \mathbf{r(x)} = r_0 \oplus r_1 x \oplus ... \oplus r_{n-1} x^{n-1}$
- Step 1: Mirror the sequence **r** (CRC must be at the end!)
- Error detection:
  - compute the CRC of all sequence **r**
    - If the remainder is 0 => no errors
    - If the remainder is non-zero => errors detected!
- Error correction:
  - use a lookup table (just like with matrices)
    - build a lookup table for all possible error words (same as with matrix codes)
    - for each error word, compute the CRC
    - when the resulting remainder is identical to the remainder obtained with **r**, we found the error word => correct errors

Skip next slides for 2018-2019

The remaining slides in this file are skipped for the class of 2018-2019.

3. Coding and encoding - The hardware way

- Coding = based on polynomial multiplications and divisions
- Efficient circuits for multiplication / division exist, that can be used for systematic or non-systematic codeword generation (draw on blackboard)

Circuits for multiplication of binary polynomials

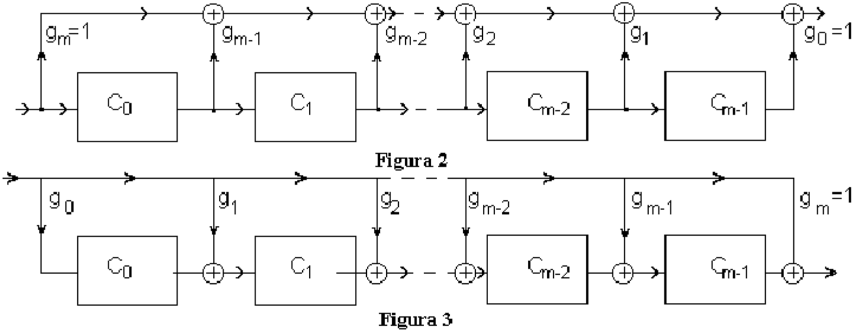

Figure 10: Circuits for polynomial multiplication

Operation of multiplication circuits

- The circuits multiply an input polynomial $a(x)$ with a polynomial $g(x)$ defined by their structure
- The input polynomial is applied at the input, 1 bit at a time, starting from highest degree
- The output polynomial is obtained at the output, 1 bit at a time, starting from highest degree
- Because output polynomial has larger degree, the circuit needs to operate a few more samples until the final result is obtained. During this time the input is 0.
- Examples: at the whiteboard

Circuits for division binary polynomials

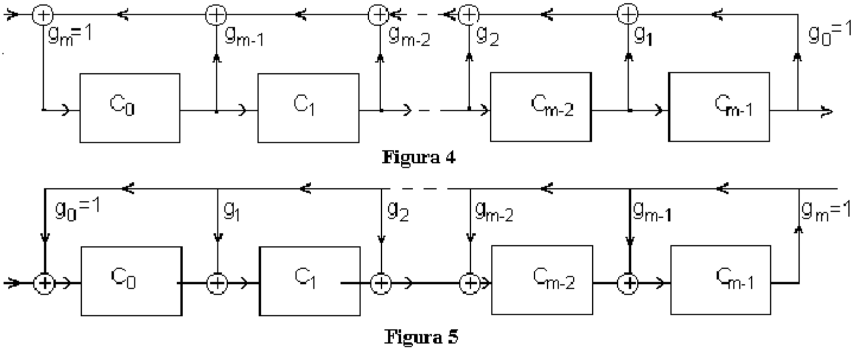

Figure 11: Circuits for polynomial division

Operation of division circuits

- The circuits divide an input polynomial $a(x)$ to a polynomial $g(x)$ defined by their structure
- The input polynomial is applied at the input, 1 bit at a time, starting from highest degree
- The output polynomial is obtained at the output, 1 bit at a time, starting from highest degree
- Because output polynomial has smaller degree, the circuit first outputs some zero values, until starting to output the result.
- If the remainder is 0, all the cells remain with 0 at the end
- Examples: at the whiteboard

Non-systematic cyclic encoder circuit

- Non-systematic cyclic encoder circuit:
  - simply a polynomial multiplication circuit
  - input is $i(x)$, output is $c(x)$

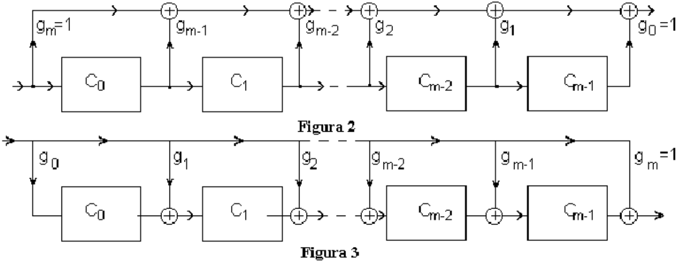

Figure 12: Circuits for polynomial multiplication

Systematic cyclic encoder circuit

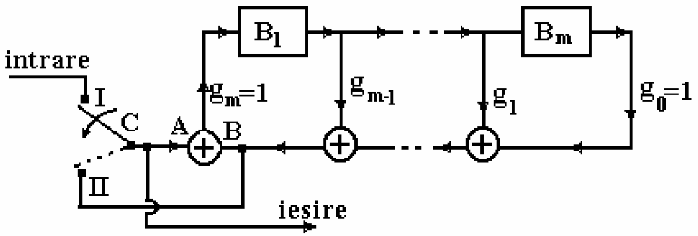

Figure 13: Systematic cyclic encoder circuit

- It contains inside a division circuit (upper right part)

Systematic cyclic encoder circuit

Operation of the cyclic encoder circuit:

- Initially all cells are 0
- Switch in position I:
  - information bits are applied to the output and to the division circuit
  - first bits of the output are the information bits => indeed systematic
  - the input bits are applied to the division circuit
- Switch in position II:
  - some output bits are put at the ouput
  - the same output bits are also applied to the input of the division circuit
- **In the end all cells end up with value 0**
  - because in phase II we add the input (A) with itself (B) at the input of the division circuit, so they cancel each other

Systematic cyclic encoder circuit

- Why is the output $c(x)$ the desired codeword? Because:
  1. has the information bits in the first part (systematic)
  2. is a multiple of $g(x)$
- Why is it a multiple of $g(x)$? Because:
  - the output $c(x)$ is always applied also to the input of the division circuit
    - in both phases of operation
  - after division, the cells end up in 0, which means there is no remainder of division
- Side note: we haven't really explained *why* the output $c(x)$ is a codeword, we just showed that it is so

The parity-check matrix for systematic cyclic codes

- Requires a more in-depth anaysis of Linear Feedback Shift Registers (LFSR)

Linear-Feedback Shift Registers (LFSR)

- A **flip-flop** = a cell holding a bit value (0 or 1)
  - called *"bistabil"* in Romanian
  - operates on the edges of a clock signal
- A **register** = a group of flip-flops, holding multiple bits
  - example: an 8-bit register
- A **shift register** = a register where the output of a flip-flop is connected to the input of the next one
  - the bit sequence is shifted to the right
  - has an input (for the first cell)
- A **linear feedback shift register** (LFSR) = a shift register for which the input is a computed as a linear combination of the flip-flops values
  - input = usually a XOR of some cells from the register
  - like a division circuit without any input
  - feedback = all flip-flops, with coefficients g_i in general
  - example at whiteboard

States and transitions of LFSR

- **State** of the LFSR = the sequence of bit values it holds at a certain moment (in order: right to left)
- The state at the next moment, $S(k+1)$, can be computed by multiplication of the current state $S(k)$ with the **companion matrix** (or **transition matrix**) $[T]$:

$$S(k+1) = [T] * S(k)$$

- The companion matrix is defined based on the feedback coefficients g_i:

$$T = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \cdot & \cdot & \cdot & \dots & \cdot \\ 0 & 0 & 0 & \dots & 1 \\ g_0 & g_1 & g_2 & \dots & g_{m-1} \end{bmatrix}$$

- Note: reversing the order of bits in the state => transposed matrix
- Starting at time 0, then the state at time k is:

Period of LFSR

- The number of states is finite => they must repeat at some moment
- The state equal to 0 must not be encountered (in this case the LFSR will remain 0 forever)
- The **period** of the LFSR = number of time moments until the state repeats
- If period is N, then state at time N is same as state at time 0:

  $$S(N) = [T]^N S(0) = S(0),$$

  which means:

  $$[T]^N = I_m$$

- Maximum period is $N_{max} = 2^m - 1$ (excluding state 0), in this case the polynomial $g(x)$ is called **primitive polynomial**

LFSR with inputs

- What if the LFSR has an input added to the feedback (XOR)?
  - exactly like a division circuit
  - assume the input is a sequence $a_{N-1}, ... a_0$
- Since a LFSR is a **linear circuit**, the effect is added:

$$S(1) = [T] \cdot S(0) \oplus \begin{bmatrix} 0 \\ 0 \\ ... \\ a_{N-1} \end{bmatrix}$$

- In general:

$$S(k_1) = [T] \cdot S(k) \oplus a_{N-k} \cdot [U],$$

where $[U]$ is:

$$[U] = \begin{bmatrix} 0 \\ 0 \\ ... \\ 1 \end{bmatrix}$$

The parity-check matrix for systematic cyclic codes

- Cyclic codes are linear block codes, so they have a parity-check and a generator matrix
  - but it is more efficient to implement them with polynomial multiplication / division circuits
- The parity-check matrix $[H]$ can be deduced by analyzing the states of the LFSR (divider) inside the encoder:
  - it is a LFSR with feedback and input
  - the input is the codeword $c(x)$
  - do computations at whiteboard ...
  - ... arrive at expression for matrix $[H]$

The parity-check matrix for systematic cyclic codes

- The parity check matrix $[H]$ has the form

$$[H] = [U, TU, T^2U, \dots T^{n-1}U]$$

- The cyclic codeword satisfies the usual relation

$$S(n) = 0 = [H]\mathbf{c}^{\mathbf{T}}$$

- In case of an error, the state at time n will be the syndrome (non-zero):

$$S(n) = [H]\mathbf{r}^{\mathbf{T}} \neq 0$$

Error detection and correction capability

Theorem:

Any (n,k) cyclic code with $g(x)$ being a primitive polynomial is capable of detecting 2 errors, or of correcting 1 error

- Proof:
  - $g(x)$ is primitive polynomial $=>$ the LSFR cycles through all possible states (non-zero)
  - therefore all the columns of [H] are distinct
  - Use the conditions based on the columns of [H] from first part of chapter
    - sum of any two columns is non-zero $=>$ can detect 2 errors
    - any two columns are distinct $=>$ can correct 1 error

Packets of errors

- Until now, we considered a single error (i.e errors appear independently)
- In real life, many times the errors appear in groups
- A **packet of errors** (*an error burst*) is a sequence of two or more **consecutive errors**
  - examples: *fading* in wireless channels
- The **length** of the packet = the number of consecutive errors

Condition on columns of [H] for packets of errors

- Conditions for packets of **e** errors are less restrictive than for **e** independent errors
- Error **detection** of e independent errors:
  - sum of **any** e or fewer columns is **non-zero**
- Error **detection** of a packet of e errors
  - sum of any **consecutive** e or fewer columns is **non-zero**
- Error **correction** of e independent errors
  - sum of **any** e or fewer columns is **unique**
- Error **correction** of a packet of e errors
  - sum of any **consecutive** e or fewer columns is **unique**

Detection of packets of errors

Theorem:

Any (n,k) cyclic code is capable of detecting any error packet of length $n - k$ or less

- A large fraction of longer bursts can also be detected (but not all)
- No proof (too complicated)

Cyclic decoder implemented with LFSR

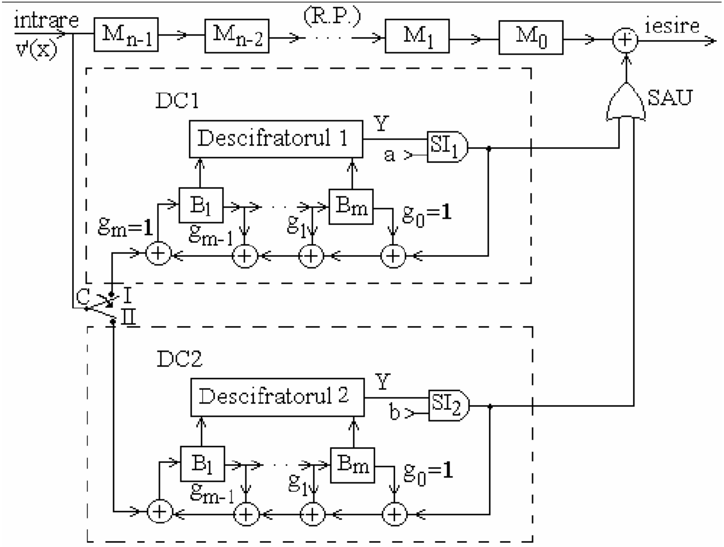

Figure 14: Cyclic decoder circuit

Cyclic decoder implemented with LFSR

- Consists of:
  - main shift register MSR
  - main switch SW
  - 2 LFSRs (divider circuits), built based on $g(x)$
  - 2 error locator blocks, one for each divider
  - 2 validation gates V1, V2, for each divider
  - output XOR gate for correcting errors

Cyclic decoder implemented with LFSR

- Operation phases:

1. Input phase: SW on position I, validation gate V1 blocked
    - The received codeword $r(x)$ is received one by one, starting with largest power of x^n
    - The received codeword enters the MSR and first LFSR (divider)
    - The first divider computes $r(x) : g(x)$
    - The validation gate V1 is blocked, no output

- Input phase ends after n moments, the switch SW goes into position II
- If the received word has no errors, all LFSR cells are 0 (no remainder), will remain 0, the error locator will always output 0,
    - the MSR will output the received bits unchanged

Cyclic decoder implemented with LFSR

2. Decoding phase: SW on position **II**, validation gate V1 open
   - LFSR keeps running with no input for n more moments
   - the MSR provides the received bits at the output, one by one
   - **exactly when the erroneous bit is at the main output of MSR, the error locator will output 1, and the output XOR gate will correct the bit (TO BE PROVEN)**
   - during this time the next codeword is loaded into MSR and into second LFSR (input phase for second LFSR)

- After n moments, the received word is fully decoded and corrected

- SW goes back into position **I**, the second LFSR starts decoding phase, while the first LFSR is loading the new receiver word, and so on

- **To prove:** error locator outputs 1 exactly when the erroneous bit is at the main output

Cyclic decoder implemented with LFSR

Theorem: if the k-th bit r_{n-k} from $r(x)$ has an error, the error locator will output 1 exactly after $k-1$ moments

- That's exactly when the erroneous k-th bit will be output from MSR => will be changed back to the good value
- **Proof:**
  1. assume error on position r_{n-k}
  2. the state of the LFSR at end of phase I = syndrome = column $(n-k)$ from $[H]$
  $$S(n) = [H]\mathbf{r}^T = [H]\mathbf{e}^T = T^{n-k}U$$
  3. after another $k-1$ moments, the state will be
  $$T^{k-1}T^{n-k}U = T^{n-1}U$$
  4. since $T^n = I_n$ -> $T^{n-1} = T^{-1}$
  5. $T^{-1}U$ is the state preceding state U, which is state
  $$\begin{bmatrix} 1 \\ 0 \\ \dots \end{bmatrix}$$

Cyclic decoder implemented with LFSR

- Step 5 above can be shown in two ways:
  - reasoning on the circuit
  - using the definition of T^{-1}

$$T = \begin{bmatrix} g_1 & g_2 & \dots g_{m-1} & 1 & \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ . & . & . & \dots & . \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix}$$

- The error locator is designed to detect this state $T^{-1}U$, i.e. it is designed as shown on blackboard
- Therefore, the error locator will correct an error
- This works only for 1 error, due to proof (1 column from [H])

Summary of cyclic codes

- Generated using a generator polynomial $g(x)$
- Non-systematic:

$$c(x) = i(x) \cdot g(x)$$

- Systematic:

$$c(x) = b(x) \oplus X^{n-k} i(x)$$

  - $b(x)$ is the remainder of dividing $X^{n-k} i(x)$ to $g(x)$
- A codeword is always a multiple of $g(x)$
- Error detection: divide by $g(x)$, look at remainder
- Schematics:
  - Cyclic encoder
  - Cyclic decoder with LFSR
  - Thresholding cyclic decoder
  - Encoder/decoder for packets of up to 2 errors