Digital Communication using GNU Radio

Prof. Kumar Appaiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

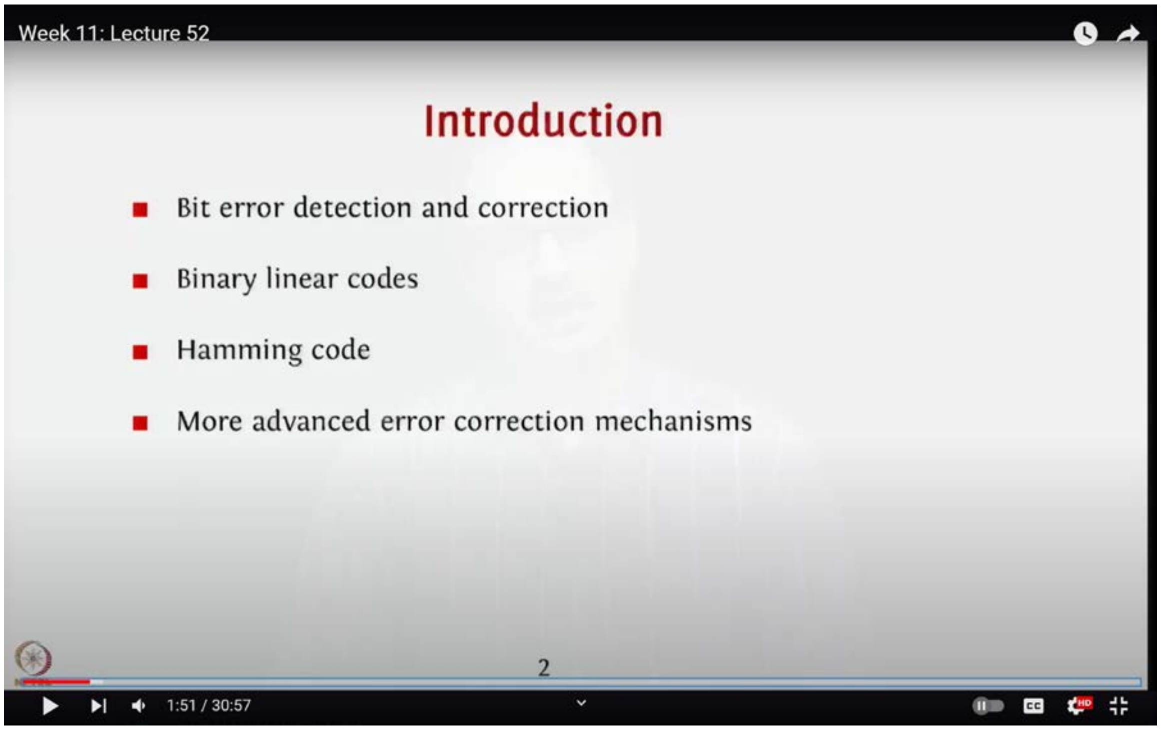
Week-11

Lecture-52

Error Control Coding: Parity Check Codes

Hello, and welcome to this lecture on Digital Communication using GNU Radio. My name is Kumar Appiah, and I am a part of the Department of Electrical Engineering at IIT Bombay. Today, we will be discussing another important concept called the error control coding. Up until now, we've discussed various practical challenges that arise in digital communication systems, challenges like noise and the impact of the channel on your transmitted signal.

(Refer Slide Time: 01:51)



Introduction

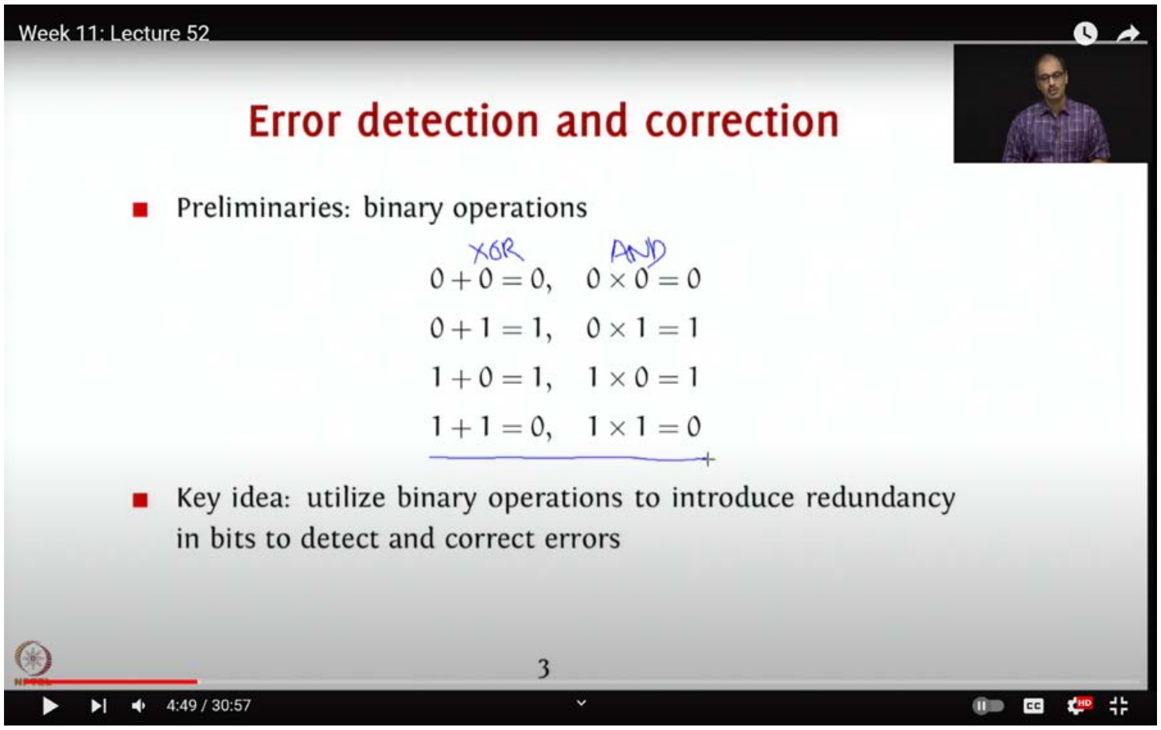
- Bit error detection and correction
- Binary linear codes
- Hamming code
- More advanced error correction mechanisms

2

We've explored several techniques to mitigate these issues, such as equalization. However, as we've seen, these methods can only go so far, especially when dealing with noise and other transmission impairments. There isn't a way to completely eliminate these effects.

So, how do we further reduce their impact? This is where error control coding comes into play. It's a powerful technique that allows us not only to detect the presence of errors but, to some extent, to correct them as well. Over the next few lectures, we will dive into a specific category of error control coding tools, focusing on linear block codes. Our discussions will center around bit error detection and correction.

(Refer Slide Time: 04:49)



Week 11: Lecture 52

Error detection and correction

- Preliminaries: binary operations

$$\begin{array}{ll} \overset{\text{XOR}}{0+0=0}, & \overset{\text{AND}}{0\times 0=0} \\ 0+1=1, & 0\times 1=1 \\ 1+0=1, & 1\times 0=1 \\ 1+1=0, & 1\times 1=0 \end{array}$$

- Key idea: utilize binary operations to introduce redundancy in bits to detect and correct errors

3

4:49 / 30:57

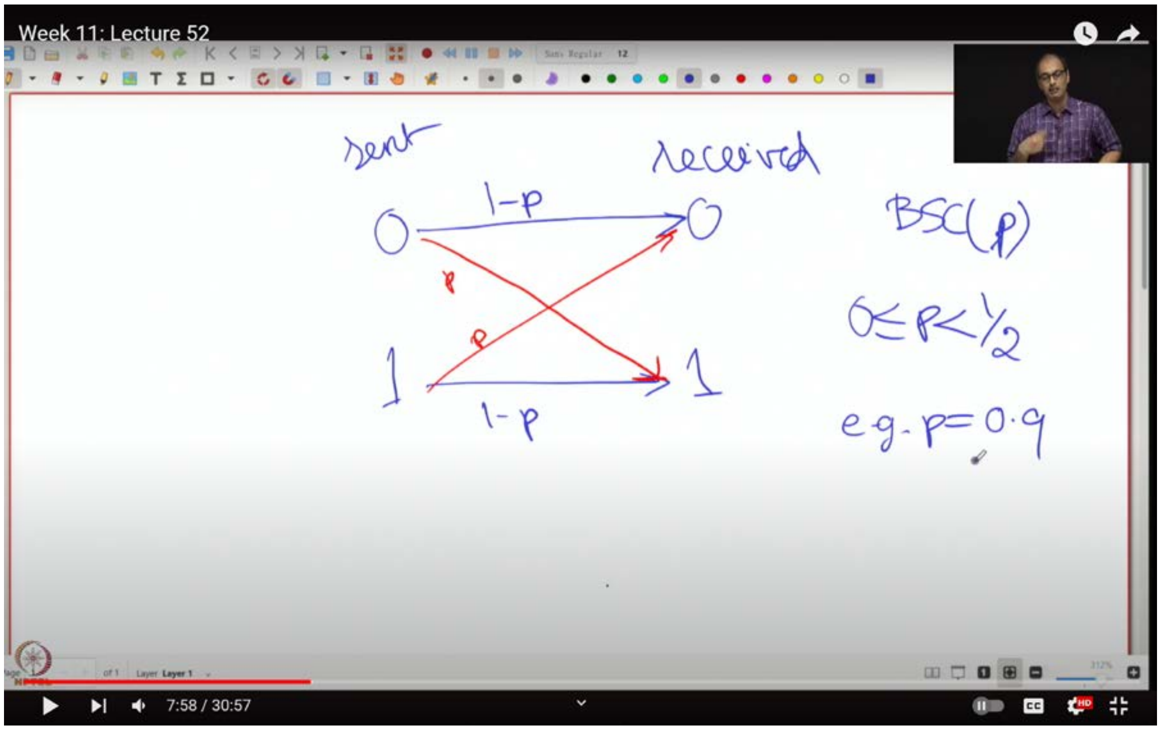
It's important to note that our discussions will operate at the bit level, specifically before modulation occurs, that is, before the bits are converted into symbols or constellation points. We'll concentrate on binary linear codes, which are a type of error control code that is both straightforward to implement and incredibly effective. While binary linear codes are not the only type available, they are certainly among the most commonly used and practical in many applications. Therefore, we will focus our attention on binary linear codes in this lecture series.

In particular, we will examine a specific code known as the Hamming code. This code is a cornerstone in the field of error control coding, and understanding it will provide a solid foundation for grasping more advanced error correction mechanisms. We will also briefly touch on where these advanced techniques are applicable and how they can be leveraged in different scenarios.

To start, we'll cover some preliminary concepts and focus on a set of fundamental binary operations. These basics will be essential as we move forward in our exploration of error control coding.

In binary operations, we work with addition and multiplication, but both are performed modulo 2. To clarify, in this context, the addition operation functions similarly to an XOR (exclusive OR), and the multiplication operation functions like an AND (logical AND). Let's break this down with truth tables for a clearer understanding.

(Refer Slide Time: 07:58)

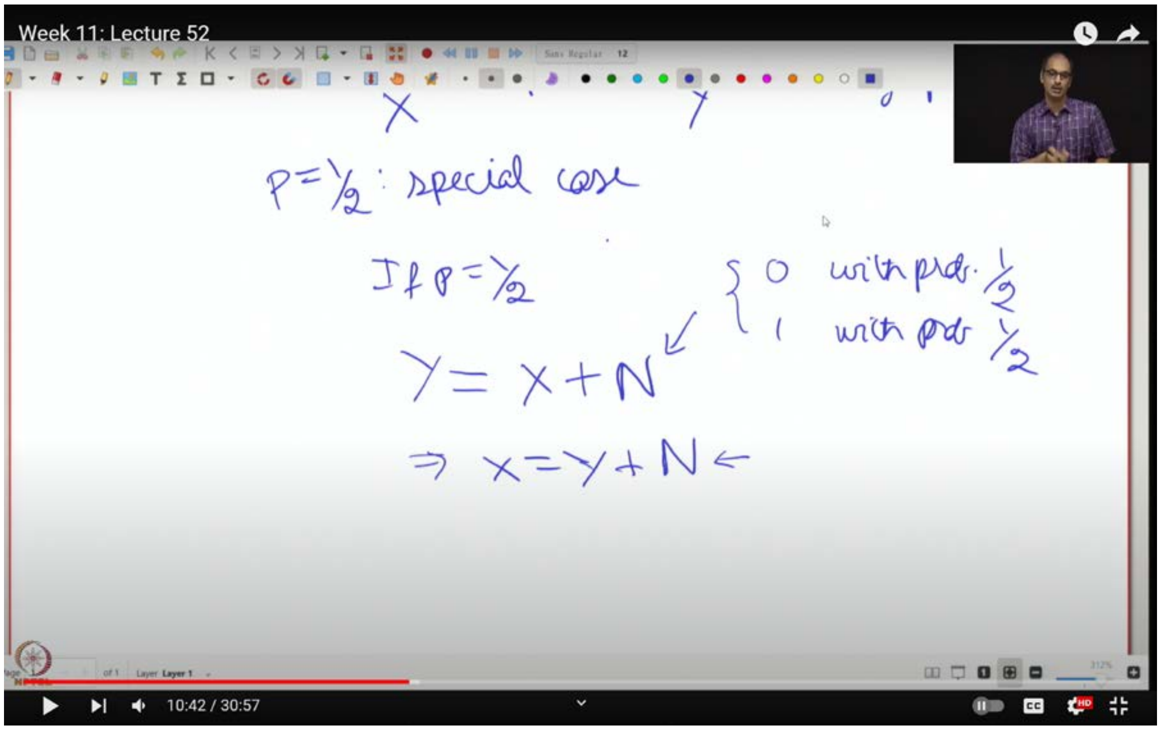


For the addition operation (modulo 2), think of it as an XOR operation. The truth table for XOR is as follows: XOR 0 with 0 results in 0, XOR 0 with 1 results in 1, XOR 1 with 0 results in 1, and XOR 1 with 1 results in 0. This table matches the truth table of an XOR gate.

On the other hand, the multiplication operation (modulo 2) resembles an AND operation. The truth table for AND is: AND 0 with 0 results in 0, AND 0 with 1 results in 0, AND 1 with 0 results in 0, and AND 1 with 1 results in 1. This table aligns with the truth table of an AND gate.

The crucial concept here is to use these binary operations to introduce redundancy into the bits we generate. This redundancy helps us detect and correct errors. For example, if you start with a sequence of 5 bits, you might expand it to 7 bits by adding 2 extra bits. These additional bits are used to detect and correct errors that might occur in the original 5 bits.

(Refer Slide Time: 10:42)



$X \qquad Y$

$p = 1/2$: special case

If $p = 1/2$

$$N = \begin{cases} 0 & \text{with prob. } 1/2 \\ 1 & \text{with prob } 1/2 \end{cases}$$

$Y = X + N$

$\Rightarrow X = Y + N$

While these binary operations can be viewed as part of a broader algebraic framework called field theory, which encompasses more general concepts in error control coding, our focus will remain on modulo 2 operations. In this framework, addition, subtraction, and multiplication have specific, consistent meanings that are fundamental to error control coding.

Now, let’s consider the adversary model, which addresses the problem we are dealing with. The main issue is that bits can be flipped due to errors. For instance, if you are using Binary Phase Shift Keying (BPSK) or any other modulation scheme, noise can cause a symbol to be detected incorrectly. This incorrect detection of a symbol results in an erroneous bit being detected.

To effectively model the errors that occur at the bit level, we use a channel model known as the binary symmetric channel. The binary symmetric channel essentially describes a scenario where if you transmit a 0, it is ideally received as a 0, although occasionally it might be received as a 1. Similarly, if you transmit a 1, it is ideally received as a 1, but sometimes it might be received as a 0. The key characteristic of this channel is that the probability of a 0 being flipped to a 1 is equal to the probability of a 1 being flipped to a 0. This equal probability is what defines the symmetry of the channel.

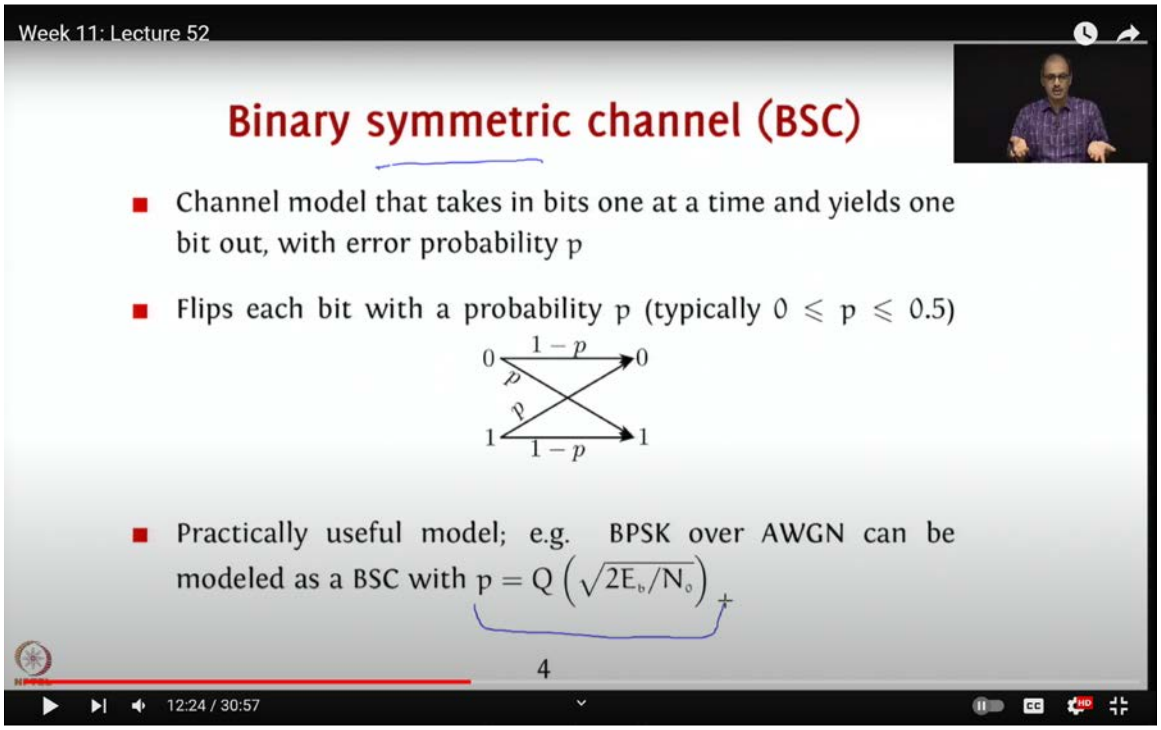
In technical terms, the binary symmetric channel takes in bits one at a time and outputs a bit with an error probability p. This means that each bit has a probability p of being flipped. Typically, p is constrained such that $0 \leq p \leq 0.5$.

To visualize this, imagine the following: if you send a 0, it might be received as a 0 with probability 1 - p, or it might be flipped to a 1 with probability p. Likewise, if you send a 1, it might be received as a 1 with probability 1 - p, or it might be flipped to a 0 with probability p. You can think of this process as flipping a biased coin for each bit. If the coin lands on the more probable outcome, you send the bit as-is; if it lands on the less probable outcome, you flip the bit before sending it.

Now, why is p typically less than or equal to 0.5? If the probability of a bit flip exceeds 0.5, say p = 0.9, sending a 0 would result in a 1 with a high probability, and sending a 1 would result in a 0 with a high probability. In this scenario, it would be more efficient to simply swap the 0s and 1s at the receiver and use a channel model with a bit flip probability of 1 - p, which would be less than 0.5. Thus, for practical purposes, we focus on binary symmetric channels where $0 \leq p < 0.5$.

The case where p = 0.5 is special because, in this case, the channel behaves like a noisy channel where each bit is equally likely to be flipped or not. If you denote the sent value by x and the received value by y, the channel behaves in a manner where each bit is effectively randomized, and the model needs to account for this symmetry.

(Refer Slide Time: 12:24)


Week 11: Lecture 52

Binary symmetric channel (BSC)

- Channel model that takes in bits one at a time and yields one bit out, with error probability p
- Flips each bit with a probability p (typically $0 \leqslant p \leqslant 0.5$)

$$\begin{tikzcd} 0 \arrow[r, "1-p"] \arrow[dr, "p"'] & 0 \\ 1 \arrow[r, "1-p"'] \arrow[ur, "p"] & 1 \end{tikzcd}$$

- Practically useful model; e.g. BPSK over AWGN can be modeled as a BSC with $p = Q\left(\sqrt{2E_b/N_o}\right)$

4

12:24 / 30:57

In the binary symmetric channel (BSC) model, the relationship between the transmitted bit x and the received bit y can be expressed as $y = x \oplus n$ or equivalently y = x + n (where addition is performed modulo 2). Here, n is a noise term that is equally likely to be 0 or 1, with each occurring with a probability of 0.5. In this context, if you analyze the relationship between y and x, you will find that y and x are uncorrelated, and indeed, they can be considered independent.

Why is this the case? When n is known, x can be directly determined. However, when n is unknown, the operation is akin to flipping a fair coin to decide whether to flip the bit or not. This randomness means you have no additional information about x based on y. Consequently, if you are given y, the probability of x being 0 or 1 is effectively 50%. This lack of dependence can be mathematically expressed by the relationship $x = y \oplus n$, which simplifies to x = y + n due to the properties of modulo 2 arithmetic where subtraction is equivalent to addition.

In this scenario, regardless of the value of y, x remains independent of y. I encourage you to prove this formally as an exercise. The intuition is that, with no knowledge of n, x is not dependent on y because n simply introduces random flips, leading to no useful information about x being gleaned from y. Thus, we will consider the binary symmetric channel with $0 \leq p < 0.5$.

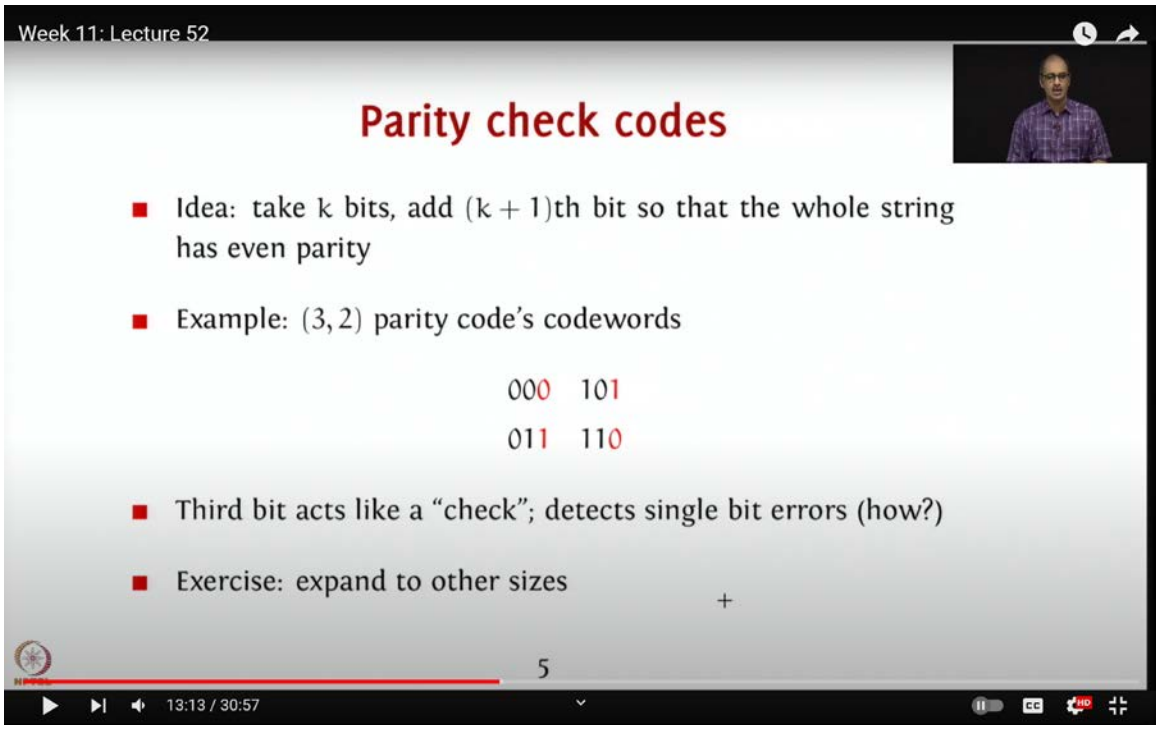
A binary symmetric channel with p = 0.5 results in a situation where no useful information can be extracted because the noise introduces a complete randomization of the bits.

In practical applications, we often model BPSK (Binary Phase Shift Keying) over an AWGN (Additive White Gaussian Noise) channel as a binary symmetric channel. In this model, the bit-flip probability p is given by $p = \frac{q}{\sqrt{2E_b/N_0}}$, where E_b is the energy per bit and N_0 is the noise power spectral density. For BPSK, if you transmit -1, there is a probability $\frac{q}{\sqrt{2E_b/N_0}}$ that it will be incorrectly detected as 1 due to noise. Similarly, if you transmit 1, there is the same probability that it will be detected as -1. As N_0 approaches infinity, the bit-flip probability p approaches 0.5, reflecting the fact that with extremely high noise, it becomes impossible to reliably detect whether the symbol was a 1 or a -1.

There are, of course, various simplifications applicable to other modulation schemes as well, but the approach we are discussing provides a clear and intuitive understanding. A BPSK (Binary Phase Shift Keying) system over an AWGN (Additive White Gaussian Noise) channel essentially simplifies to a binary symmetric channel under these assumptions.

Let us now turn to our first practical example of an error control code. Note that I specifically used the term "error control code" rather than "error correction code." Error control codes encompass both error detection and error correction. While some codes are designed solely for detecting errors, others can detect and correct errors as well. We will begin with a code primarily used for error detection, known as the parity check code.

(Refer Slide Time: 13:13)



Parity check codes

- Idea: take k bits, add $(k+1)$th bit so that the whole string has even parity
- Example: $(3, 2)$ parity code's codewords

  000 101
  011 110

- Third bit acts like a "check"; detects single bit errors (how?)
- Exercise: expand to other sizes

5

The key idea behind parity check codes is to add an extra bit to a set of k bits, so that the entire sequence has even parity. For example, consider the 3-2 parity check code. To illustrate, let's detail this example.

In a 3-2 parity check code, the notation '3-2' indicates that for every 2 information bits, we add 1 parity bit to make a total of 3 bits. Here's how it works: you take 2 bits at a time and output 3 bits, which are then sent to the modulator for further processing.

Let's say we have the bit sequence: 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, and so on. According to the parity check code rule, for each group of 2 bits, we append 1 parity bit to ensure that the total number of 1s in the group of 3 bits is even.

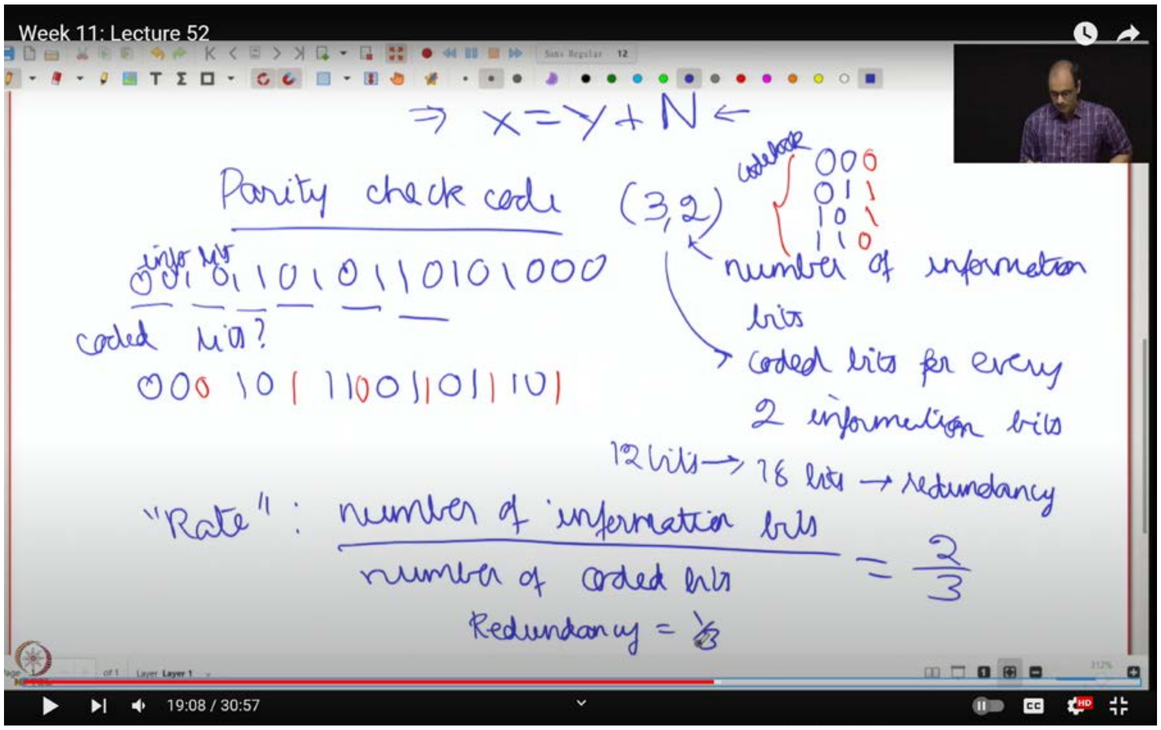
Here's the breakdown: In our modulo-2 system, addition and multiplication are performed as follows:

- For the bit group 0, 0, the parity bit added is 0 because the sum (0 + 0 + 0) is 0, which is even.

- For the bit group 0, 1, the parity bit added is 1 because the sum (0 + 1 + 1) is 0, and we need an extra 1 to make the sum even.
- For the bit group 1, 0, the parity bit added is 1 because the sum (1 + 0 + 1) is 0.
- For the bit group 1, 1, the parity bit added is 0 because the sum (1 + 1 + 0) is 0.

Remember, in our modulo-2 system, 1 + 1 equals 0. This is an important detail when working with binary operations where the arithmetic is performed modulo 2.

(Refer Slide Time: 19:08)


Week 11: Lecture 52

$\Rightarrow x = y + N \Leftarrow$

Parity check code $(3,2)$

codebook: 000, 011, 101, 110

number of information bits

coded bits for every 2 information bits

info bits: 0 0 1 0 1 1 0 1 0 1 1 0 1 0 1 0 0 0

coded bits?

000 101 1100 1101 1101

12 bits → 18 bits → redundancy

"Rate": $\dfrac{\text{number of information bits}}{\text{number of coded bits}} = \dfrac{2}{3}$

Redundancy $= \frac{1}{3}$

19:08 / 30:57

This is the codebook we are working with. Given this codebook, let’s examine how we handle the information bits and their corresponding coded bits.

For this example, we’ll add parity bits to our information bits to form the coded bits. Let’s consider the following sequence of information bits: 0, 0, 1, 0, 1, 1, 0, 1, and so on. For each of these bit pairs, we will append a parity bit to ensure even parity.

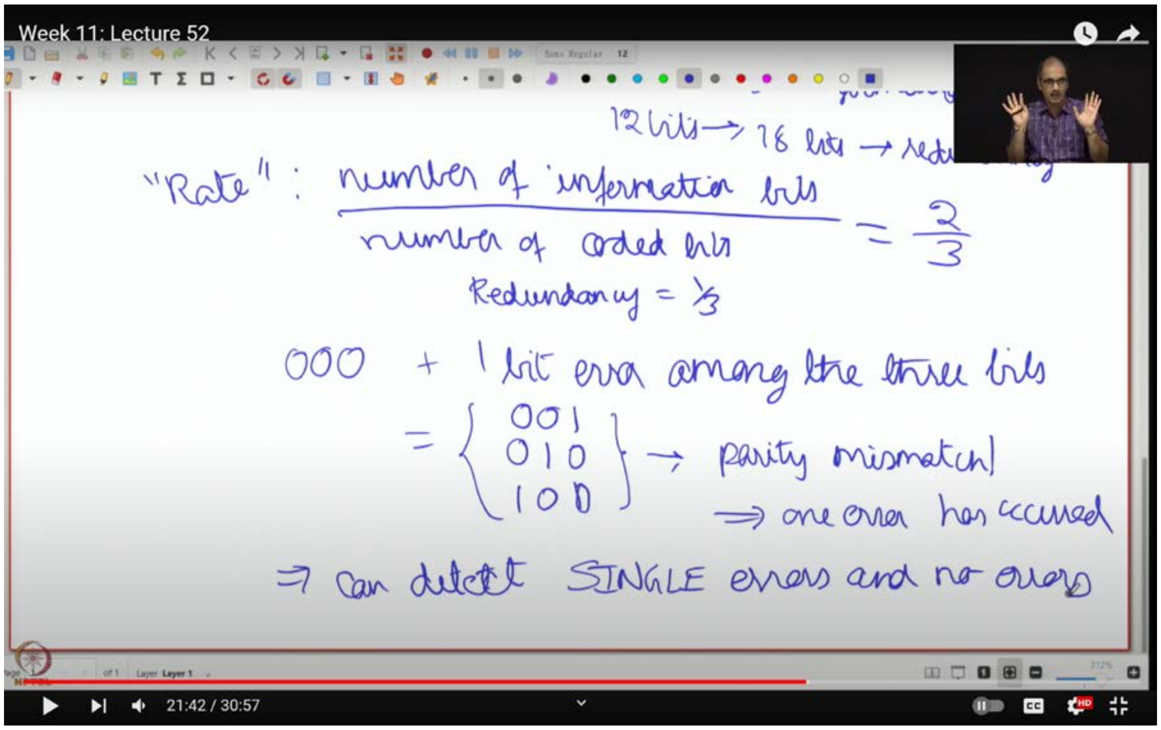
Let’s determine the parity bit for each pair:

- For 0, 0, the parity bit is 0, because 0 + 0 + 0 = 0 (even).
- For 1, 0, the parity bit is 1, because 1 + 0 + 1 = 0 (even).
- For 1, 1, the parity bit is 0, because 1 + 1 + 0 = 0 (even).
- For 0, 1, the parity bit is 1, because 0 + 1 + 1 = 0 (even).

Thus, instead of sending 12 bits, you will now send 16 bits. Specifically, you're sending 3 coded bits for every 2 information bits.

This process introduces redundancy, as we're adding extra bits to our transmission. The code rate, which is the ratio of information bits to coded bits, is $\frac{2}{3}$. Thus, out of the total bits sent, approximately 66.66% are information bits and 33.33% are redundant.

(Refer Slide Time: 21:42)



Week 11: Lecture 52

12 bits → 18 bits → Redu

"Rate" : $\frac{\text{number of information bits}}{\text{number of coded bits}} = \frac{2}{3}$

Redundancy = $\frac{1}{3}$

000 + 1 bit error among the three bits

$= \left\{ \begin{matrix} 001 \\ 010 \\ 100 \end{matrix} \right\} \rightarrow$ parity mismatch

$\Rightarrow$ one error has occurred

$\Rightarrow$ can detect SINGLE errors and no errors

21:42 / 30:57

Technically, the redundancy can be viewed as overhead. In this context, one-third of the transmitted bits are for redundancy. Alternatively, you might describe this as 50% overhead if you consider the additional bits as a fraction of the original information bits.

Thus, for an (n, k) code, where n is the total number of bits transmitted and k is the number of information bits, the rate of the code is $\frac{k}{n}$. In this example, the rate is $\frac{2}{3}$, reflecting the redundancy introduced by the parity bits.

Now let’s analyze this particular code in more detail.

Consider the case where the codeword 0, 0, 0 is sent, and suppose exactly one bit error occurs in this 3-bit block. This means one of the bits is flipped, resulting in possible error patterns such as 0, 0, 1, 0, 1, 0, or 1, 0, 0.

For each of these patterns, we can observe that there is a parity mismatch, indicating that an error has occurred. The reason is that the parity rule requires the total number of 1s in each block of three bits to be even. Let’s verify this:

- For the pattern 0, 0, 1 : The sum is 0 + 0 + 1 = 1 (odd).
- For the pattern 0, 1, 0 : The sum is 0 + 1 + 0 = 1 (odd).
- For the pattern 1, 0, 0 : The sum is 1 + 0 + 0 = 1 (odd).

Since each of these patterns results in an odd number of 1s, they do not satisfy the parity check rule, which requires an even number of 1s. Thus, the parity check code can effectively detect single errors. However, it cannot correct them, nor can it handle multiple errors within the same block.

Now, let’s see how this works with an example. Suppose you have a bit pattern like 00100010000110001. If we use this pattern with a binary symmetric channel, you will need to check the received bits for parity.

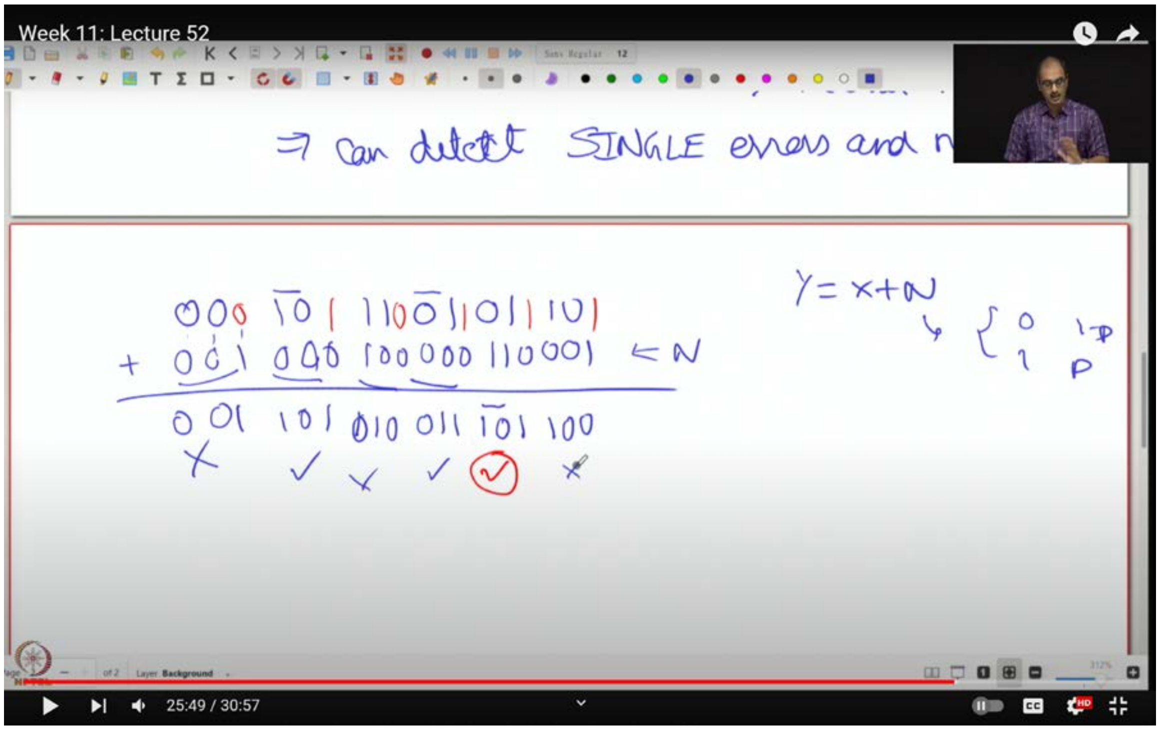
For instance:

- If you receive 001, the parity is odd (since there is one 1), so you would reject it.
- If you receive 101, the parity is even, so you would accept it.
- If you receive 010, the parity is odd, so you would reject it.
- If you receive 011, the parity is even, so you would accept it.

Here’s an interesting point: Up to this point, we have only seen cases where there is a single bit error in the block of 3. However, if there are 2 bit errors in the block, the parity will still appear correct. For example, if 011 was sent, but due to 2 errors it was received as 101, the resulting block still has even parity, misleading the detection process. Thus, you would incorrectly conclude no error occurred, even though 2 bit errors were introduced.

Therefore, the single parity check code can effectively detect single-bit errors but fails to detect or correct multiple-bit errors. If there are more than one bit error, the code may fail to detect the error or provide incorrect results, as seen in the case of 011 being mistakenly read as 101. This demonstrates the limitations of the parity check code, where the detection capability is compromised with more than one bit error.

(Refer Slide Time: 25:49)



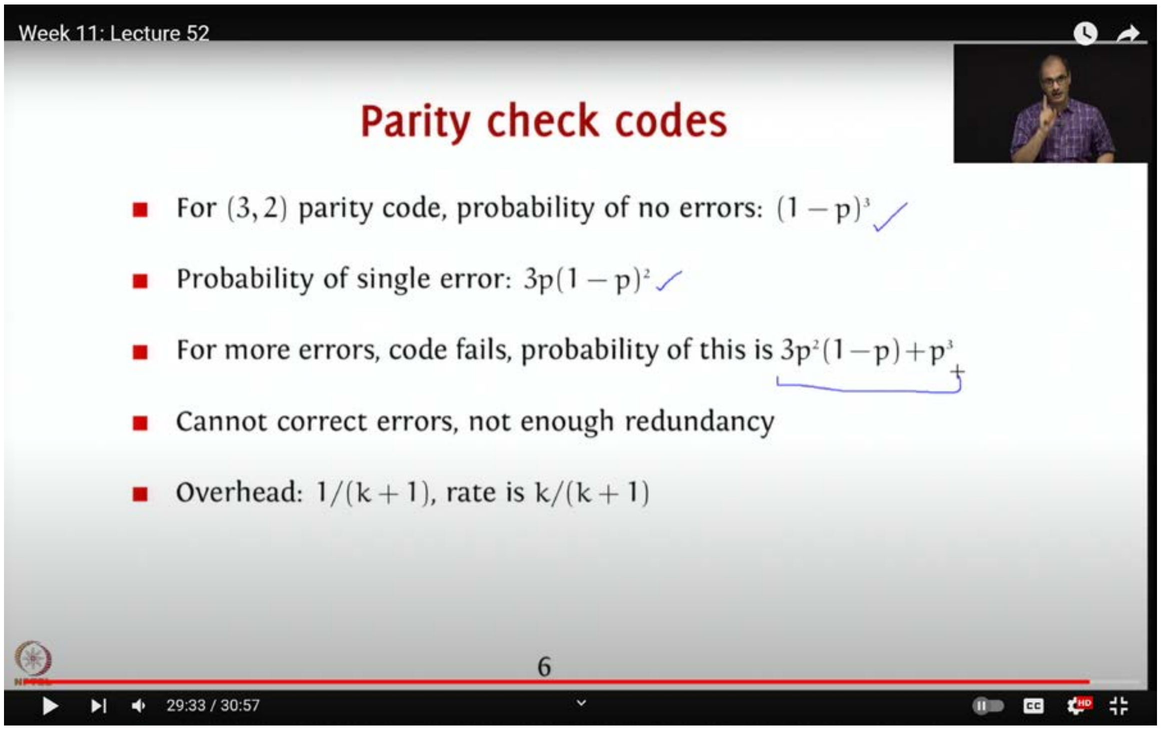
This is an important aspect to consider: when there are 2-bit errors, the parity check code can fail. The failure occurs because with 2-bit errors, the code might incorrectly conclude that no errors have occurred. The way the parity check code is designed, it involves appending a parity bit to the data bits. After transmitting, the receiver drops this parity bit and checks the remaining bits for correctness. If the remaining bits satisfy the parity condition, they are considered correct. However, if there are 2-bit errors, the parity check may not detect these errors, leading to incorrect conclusions.

For instance, in a simple parity check code, a single additional bit is used to detect errors. This bit serves as a check and is effective for detecting single-bit errors. If you want to extend this idea to codes of different sizes, it is straightforward. For example, you can use a 5-4 parity code, where you take 4 bits of data and add a 5th bit for parity. In this case, you append a parity bit to make sure that the total number of 1s in the 5-bit block is even.

Consider the following example:

- For the bit sequence `1010`, which already contains an even number of 1s, you add a `0` as the parity bit.
- For the bit sequence `1011`, which has an odd number of 1s, you add a `1` to make the total number of 1s even.

(Refer Slide Time: 29:33)


Week 11: Lecture 52

Parity check codes

- For $(3,2)$ parity code, probability of no errors: $(1-p)^3$ ✓
- Probability of single error: $3p(1-p)^2$ ✓
- For more errors, code fails, probability of this is $\underbrace{3p^2(1-p)+p^3}$
- Cannot correct errors, not enough redundancy
- Overhead: $1/(k+1)$, rate is $k/(k+1)$

6

29:33 / 30:57

In general, for a k+1 parity check code, the rate of the code is given by $\frac{k}{k+1}$. The redundancy of the code is $\frac{1}{k+1}$. As the parity check code gets larger, it can detect more single-bit errors but may struggle with multiple-bit errors, leading to an increased likelihood of making mistakes. Therefore, it's crucial to balance the code's size and error detection capability.

Now, let’s examine the probability of making errors with a 3-2 parity code.

The probability of encountering no errors in a block of 3 bits is given by $(1 - p)^3$. On the other hand, the probability of a single error occurring is $3p(1 - p)^2$. This calculation is based on the fact that the error could occur in any one of the three positions. For each position, the probability of having an error in that spot while the other two bits are correct is $p(1 - p)^2$. Since these scenarios are mutually exclusive, you sum them up to get $3p(1 - p)^2$.

For more than one error, if there are two errors, the probability of this happening is $3p^2(1 - p) + p^3$. If there are more than two errors, the code fails to detect them correctly. In other words, the code will accurately detect no errors or a single error and then accept the bit sequence after removing the parity bit. However, if there are two or three errors, the code cannot reliably detect them, leading to incorrect conclusions.

Despite these limitations, using a parity check code does offer some advantages. For an uncoded system, the probability of a single bit error is p. In comparison, for a block of 2 bits, the probability of at least one error is p^2 (if both bits are erroneous) plus $2p(1 - p)$ (if one bit is erroneous). The parity check code, with a probability of $3p(1 - p)^2$ for detecting single-bit errors, generally provides better performance. This can be verified through calculations or exercises.

It’s important to note that while a parity check code can detect errors, it cannot correct them. The redundancy of the code is $\frac{1}{k+1}$, and the rate is $\frac{k}{k+1}$, as we have discussed. This parity check code is effective for detecting certain errors but not for correcting them.

In the next lecture, we will explore another class of simple error control codes known as block codes. These block codes not only detect errors but can also correct them, meaning that they can recover the original information bits even if some of the coded bits are flipped. I look forward to discussing this with you next time. Thank you.