Digital Communication using GNU Radio

Prof. Kumar Appaiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

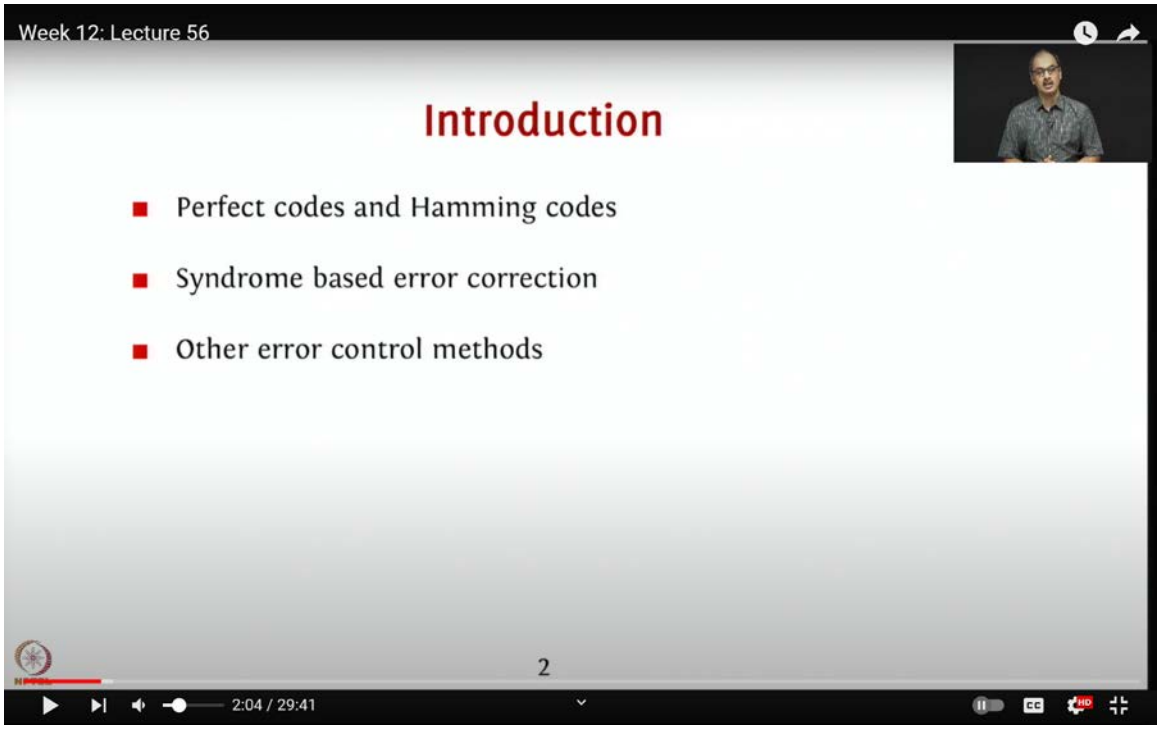
Week-12

Lecture-56

Error Control Coding: Perfect Codes

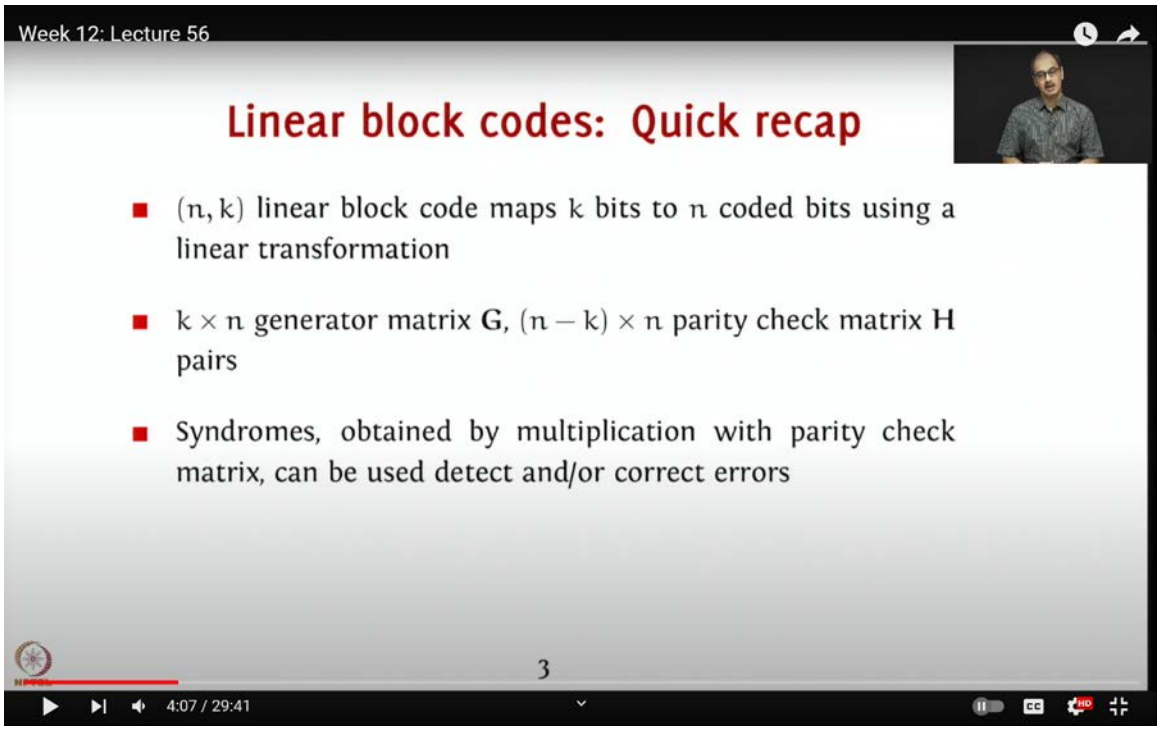
Hello and welcome to this lecture on Digital Communication using GNU Radio. My name is Kumar Appiah, and I am with the Department of Electrical Engineering at IIT Bombay. Today's lecture continues our exploration of error control codes. In our previous session, we implemented the repetition code in GNU Radio and observed its effectiveness in correcting errors. To refresh your memory, with an n-bit repetition code, where n is an odd number, we could reliably correct up to $\frac{n-1}{2}$ bit errors and accurately determine the sent bit. This approach significantly reduced the bit error rate.

(Refer Slide Time: 02:04)



However, a major drawback is the code rate, which is $\frac{1}{n}$. As n increases, the rate deteriorates markedly, making it impractical for large block lengths. In this lecture, we will explore a class of linear block codes known as Hamming codes. Hamming codes are notable for their ability to correct a single-bit error while maintaining a higher rate than repetition codes. They also offer intriguing geometric and structural properties that are worth understanding.

(Refer Slide Time: 04:07)



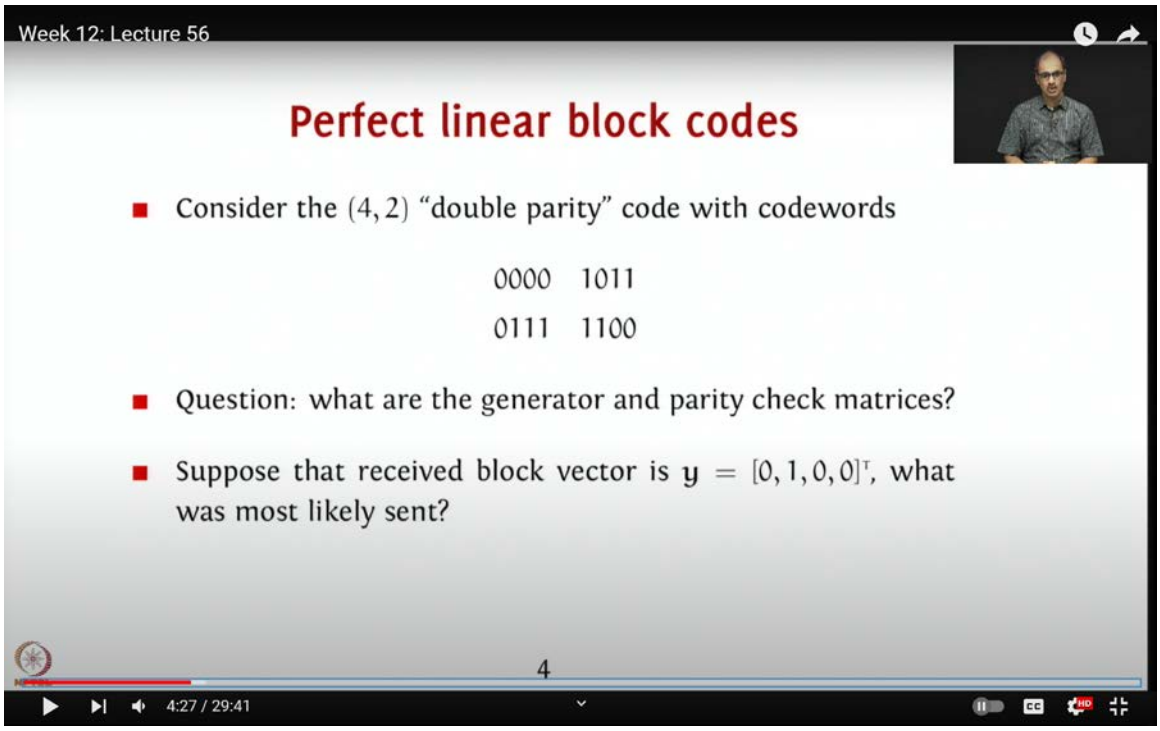
We will first discuss perfect codes and then delve into Hamming codes, which are a specific type of perfect code. Following that, we will cover syndrome-based error correction. As we've mentioned in previous lectures, by computing the syndrome, obtained by multiplying the received bit vector by the parity-check matrix, and comparing it with various error pattern syndromes, we can identify and correct errors.

Finally, we will explore additional error control methods and discuss commonly used techniques in the field. To recap, linear block codes are defined by two parameters, n and k. An (n, k) linear block code maps k bits to n coded bits using a linear transformation.

If you recall, we previously implemented the linear transformation using a matrix G. By

convention, $\boldsymbol{G}$ is a matrix with k rows and n columns. This means the message, which is a k-bit vector, is pre-multiplied by $\boldsymbol{G}$, resulting in an n-bit vector, known as the codeword. Correspondingly, there is an $(n-k) \times n$ parity check matrix, denoted H. The matrix $\boldsymbol{H}$ is designed so that when any codeword is multiplied by $\boldsymbol{H}$, the result is zero. Essentially, the parity check matrix $\boldsymbol{H}$ is used to verify that no error has occurred.

(Refer Slide Time: 04:27)

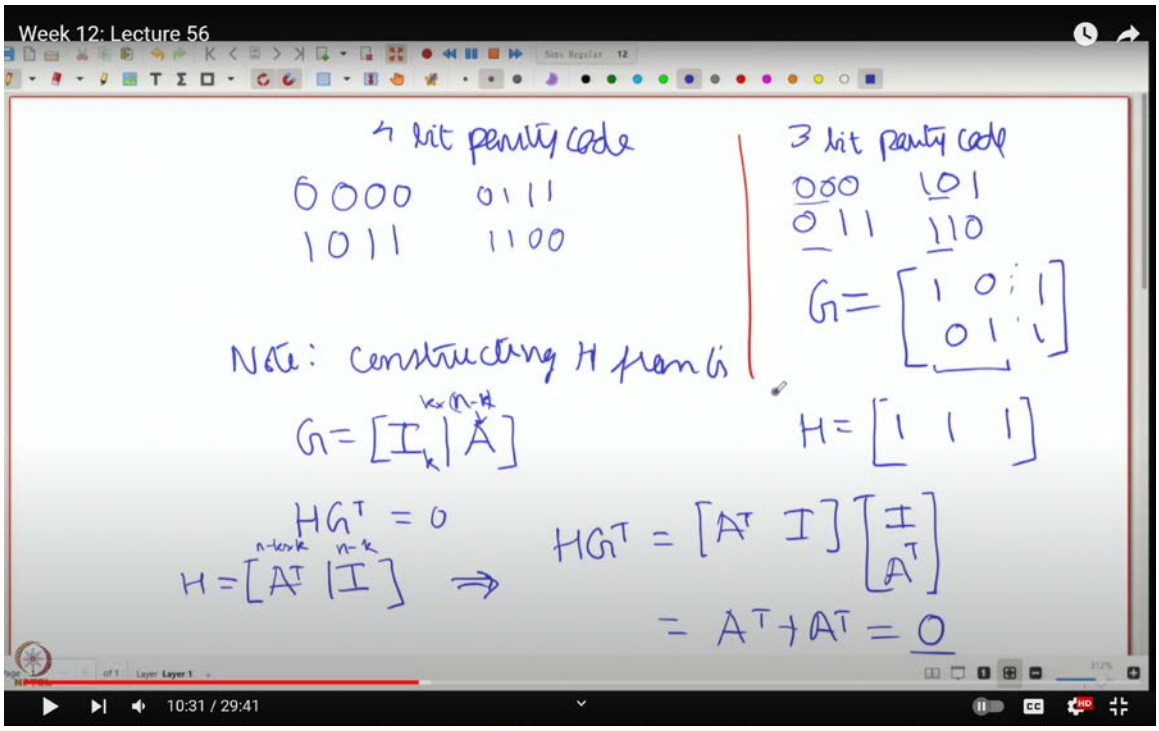


Syndromes are obtained by multiplying the received bit vector by the parity check matrix $\boldsymbol{H}$. If the result, which is the syndrome, is zero, it indicates that no errors have occurred, though there could be many errors, this usually suggests no errors. If the syndrome is non-zero, it not only signals the presence of an error but can also potentially identify where the error occurred. This was demonstrated with parity check matrices, such as those used in parity codes, where they indicated the occurrence of errors and sometimes even their location.

Now, let's briefly discuss perfect codes or perfect linear block codes. Consider the (4,2) double parity code with codewords:

0000, 0101, 1011, 1110

(Refer Slide Time: 10:31)



To find the generator and parity check matrices for this code, let's do a quick analysis. For example, with a 4-bit parity code:

0000, 0101, 1011, 1110

We generate the code by taking the first two bits, finding their parity, and repeating it. For instance, 10 has parity 1, so 11 is repeated twice. Similarly, 01 has a parity of 1, so you get 11. This code is essentially a variant of the 3-bit parity code, which involves appending bits to form the codewords.

To find the generator matrix for this code, we can use the standard approach by selecting two linearly independent codewords. For this example, let's choose:

1010, 1110

To find the parity check matrix, here's a useful trick: if you write your generator matrix in

the form $[\, I \mid A \,]$, where I is the $k \times k$ identity matrix and A is the remaining matrix, then A must be an $(n-k) \times k$ matrix.

Let's construct the generator matrix for the 3-bit parity code:

- Choose two linearly independent codewords.
- Use them to form the generator matrix.

Finally, for the parity check matrix, if you write the generator matrix as $[\, I \mid A \,]$, where I is the identity matrix and A is the matrix, then A should be an $(n-k) \times k$ matrix.

To construct the parity check matrix $\boldsymbol{H}$, you need to ensure that $H \cdot G^T = 0$. Here's an efficient way to achieve this without much hassle:

Consider the matrix $\boldsymbol{H}$ in the form $[\, A^T \mid I \,]$, where A is an $(n-k) \times k$ matrix and $\boldsymbol{I}$ is an $(n-k) \times (n-k)$ identity matrix. To determine the size of $\boldsymbol{I}$, note that since $\boldsymbol{A}$ is $(n-k) \times k$, the transpose A^T will be $k \times (n-k)$, and I must be $(n-k) \times (n-k)$.

Let's use this to construct $\boldsymbol{H}$. Suppose we have:

$$G = [\, I \mid A \,]$$

Then:

$$H = [\, A^T \mid I \,]$$

To verify this, compute:

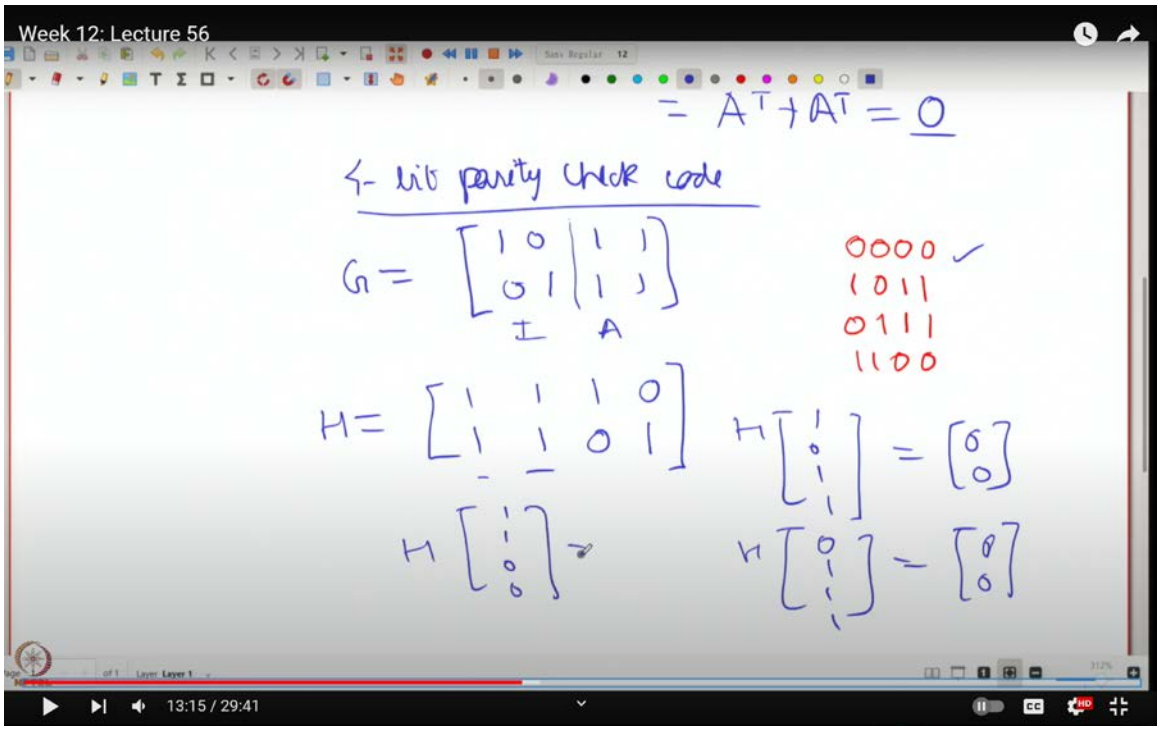
$$H \cdot G^T = [\, A^T \mid I \,] \cdot [\, I^T \mid A^T \,]$$

Expanding this:

$$H \cdot G^T = [\, A^T \cdot I^T \mid A^T \cdot A^T \,] = [\, A^T \mid A^T \cdot A^T \,]$$

Since $A^T \cdot A^T$ results in the zero matrix under modulo 2 arithmetic, this confirms that $\boldsymbol{H}$ is correct.

(Refer Slide Time: 13:15)



For instance, in a systematic linear block code where the codeword appears directly, you can see an identity matrix at the front. For example, if we have a (4,2) code with the following codewords:

0000, 0101, 1011, 1110

The generator matrix $\boldsymbol{G}$ can be:

$$G = [\, I \mid A \,]$$

where $\boldsymbol{A}$ consists of the parity check components. Construct $\boldsymbol{H}$ as:

$$H = [\, A^T \mid I \,]$$

For the (3-bit) parity check matrix code, the generator matrix might be:

$$G = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

To find the parity check matrix $\boldsymbol{H}$, use the approach:

$$H = [\, A^T \mid I \,]$$

For the 3-bit parity code, $\boldsymbol{A}$ is:

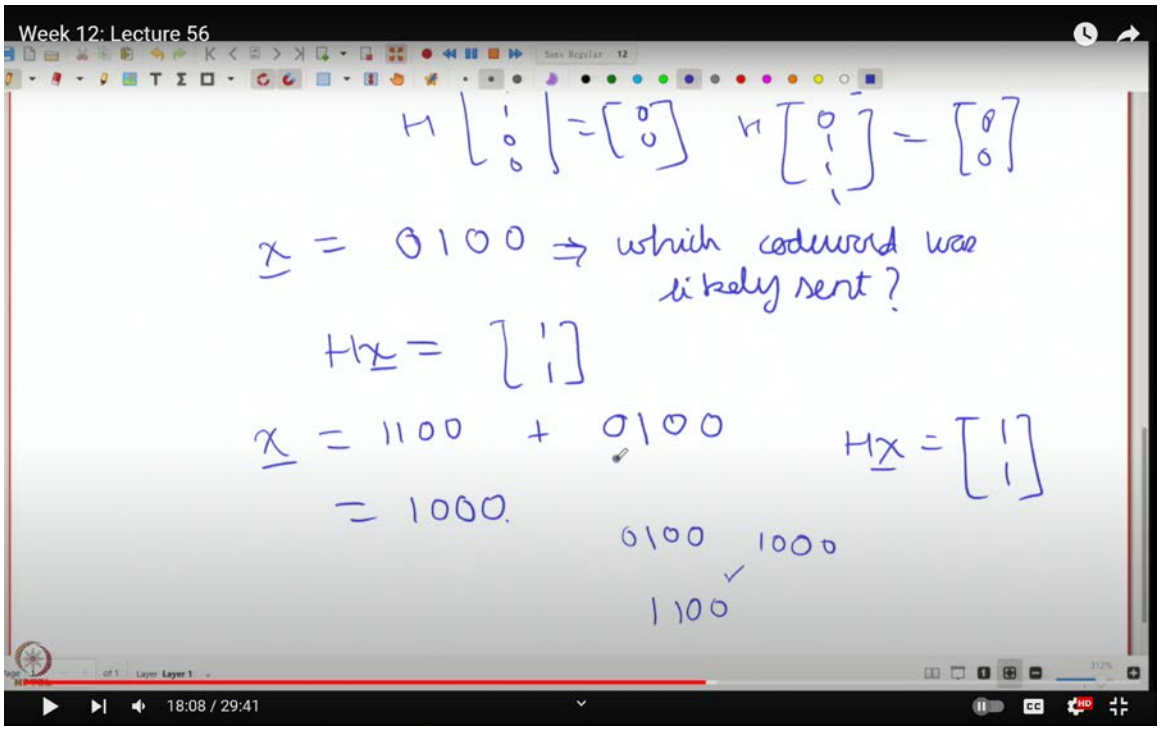
$$A = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

Thus:

$$H = \begin{pmatrix} 1 & 1 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Verify that $\boldsymbol{H}$ is correct by checking that the syndrome is zero for codewords, confirming the code's correctness. If the parity check matrix results in non-unique columns or other issues, this may reflect linear dependencies in $\boldsymbol{H}$ or $\boldsymbol{G}$, but for a well-constructed code, these will be minimal.

(Refer Slide Time: 18:08)

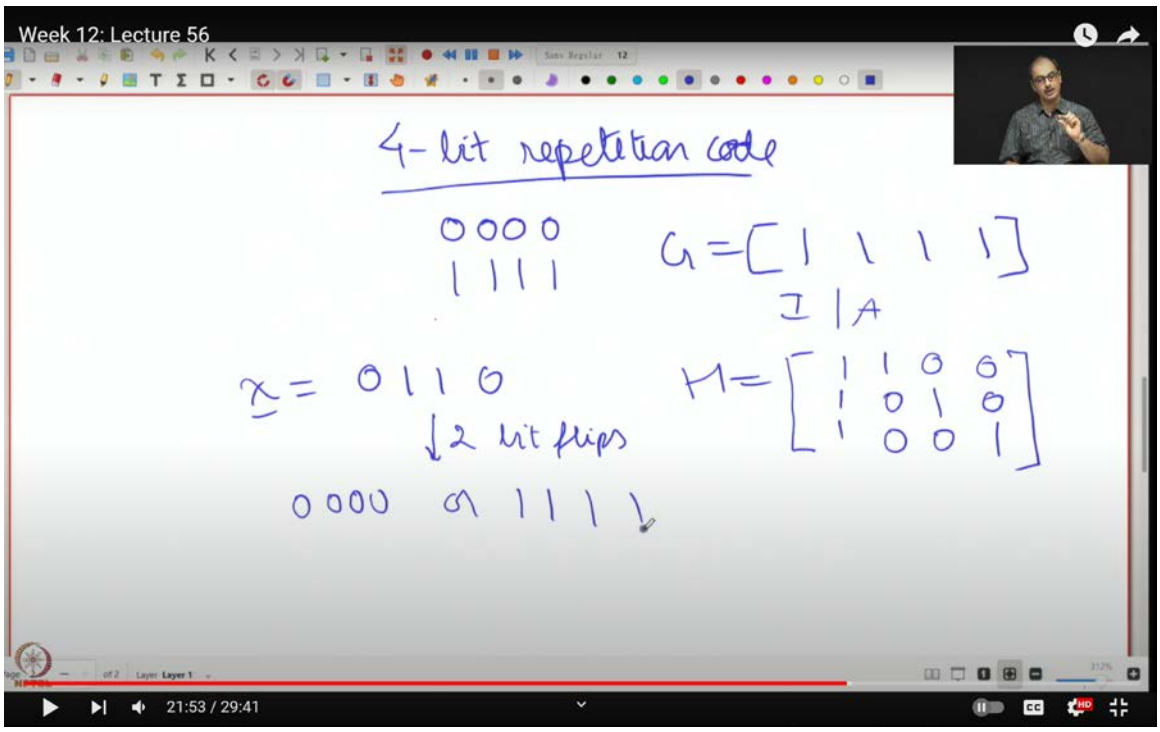


This method of constructing $\boldsymbol{H}$ from $\boldsymbol{G}$ and verifying correctness ensures accurate error detection and correction capabilities.

Let's examine an intriguing error event by considering a specific error pattern. Suppose we have an error event E with some bit flips that result in a non-unique outcome. For illustration, let's assume the received sequence is 0 1 0 0. Although I've presented this sequence as a string, you can envision it as a column vector. For clarity, let's refer to this sequence as X rather than calling it an error.

The question now is: which code word was most likely transmitted? Given 0 1 0 0, we need to determine its syndrome. In this case, the syndrome is 1 1. The issue with this syndrome is that it does not uniquely identify the exact error. The 1 1 syndrome could result from an error occurring in the first bit or the second bit of the sequence.

(Refer Slide Time: 21:53)



To better understand this, consider the decodability of the code. Suppose we have a set of code words and the received sequence is 0 1 0 0. A single-bit flip could have led to this received sequence. For instance, flipping one bit in 0 1 0 0 might result in 1 1 0 0, or

flipping a different bit could result in 0 0 0 0. Therefore, multiple single-bit flips can produce valid code words, creating ambiguity in identifying the exact transmitted code word.

To illustrate this further, let's consider another example. Suppose we have 0 1 1 1 and we add it to 0 1 0 0. The result of this addition is 0 0 1 1. Applying the syndrome calculation $H \cdot X$ to this result yields 1 1 again. This outcome is consistent with our previous understanding that $H \cdot X$ (where X is a code word with an added error) is equivalent to $H \cdot E$, because $H \cdot \text{code word} = 0$, thus yielding 1 1.

The presence of 1 1 in the syndrome indicates that the error could have occurred in the first or second bit, or potentially in both. However, for this example, let's disregard the possibility of two-bit errors as they are less likely. Thus, the error could have occurred either in the second bit of 0 1 1 1, which would result in flipping it to 0 0, or in the second bit of the received sequence.

Let me clarify this by correcting the explanation, focusing on the case of the sequence 1 1 0 0 instead of 0 1 1 1, which makes the analysis a bit easier.

Consider the sequence 1 1 0 0. To analyze this, we calculate its syndrome using the matrix H. If $H \cdot X = 1\ 1$, this indicates that if a single-bit error occurred, it must have been in either the first or second position. Specifically, flipping the first bit of 1 1 0 0 results in 0 1 0 0, and flipping the second bit results in 1 0 0 0. Neither of these outcomes is a valid code word.

Thus, there is no way to correct this error because neither 0 1 0 0 nor 1 0 0 0 corresponds to a valid code word. This demonstrates a fundamental problem with this code: not all n-bit vectors can be uniquely mapped to a single valid code word with the minimum number of bit flips. This issue arises due to inherent ambiguities in the generator matrix G and parity check matrix H.

Now, let's analyze a different example, specifically a 4-bit repetition code where n is chosen as an odd number. In practice, an odd number simplifies error correction because majority logic becomes unambiguous. For example, with 3 bits, choosing the majority is

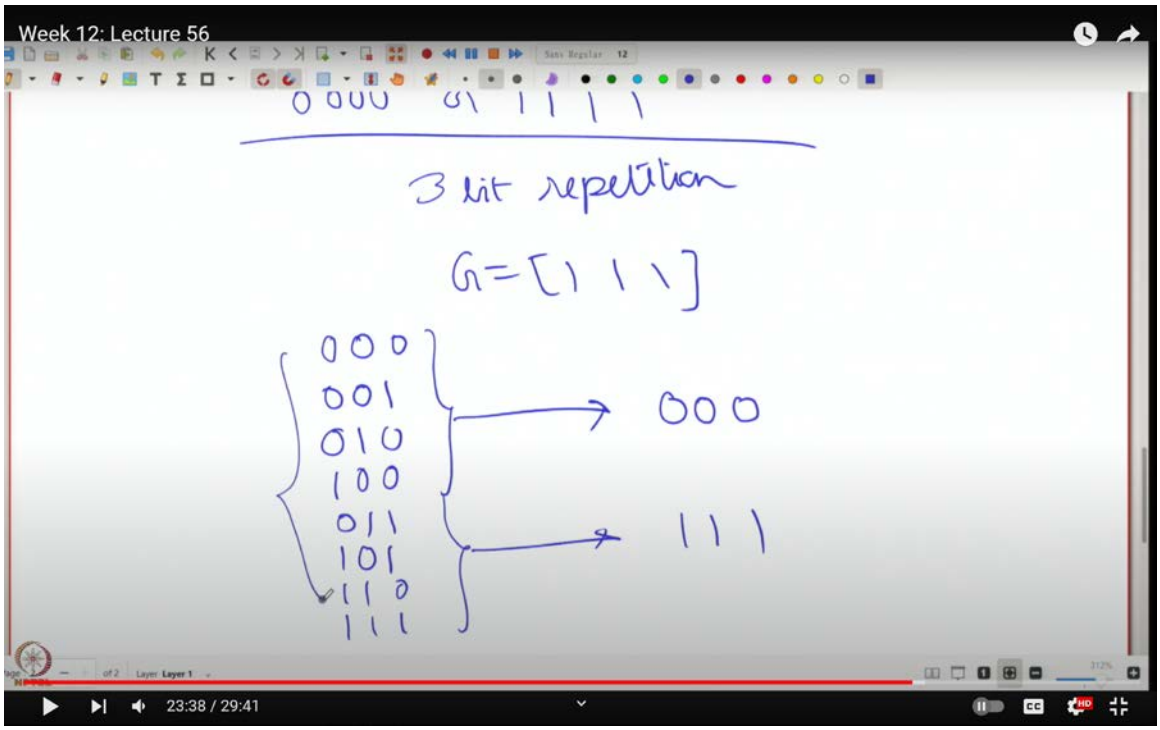
straightforward. However, with 4 bits, if you encounter an equal number of 0s and 1s, the decision becomes trickier.

For the 4-bit repetition code, the generator matrix G is straightforward, and the parity check matrix H is the transpose of the identity matrix. This setup ensures that multiplying 0 yields 0 and multiplying 1 1 1 1 yields 0, validating the parity check matrix.

However, there are issues with this code as well. For instance, consider the sequence 0 1 1 0. Similar to the previous case, flipping any single bit in this sequence does not produce a valid code word. For example, flipping the first bit results in 1 1 1 0, and flipping the second bit results in 0 0 1 0 or 0 1 0 0. None of these sequences is a valid code word.

Now, examining 2-bit flips, although each sequence is equally likely, performing two-bit flips can result in different code words. This discrepancy shows that a 4-bit repetition code cannot uniquely map all n-bit sequences to a single code word with minimum bit flips. Therefore, this code does not satisfy unique decodability for every possible n-bit sequence.

(Refer Slide Time: 23:38)



As we move towards more advanced codes like Hamming codes, one crucial requirement will be to ensure that for any received n-bit sequence, it is possible to perform the minimum number of bit flips to obtain a unique code word.

Let's discuss an example of a well-structured code word using a 3-bit repetition code. In this code, the generator matrix G is [1 1 1]. The code words generated are 0 0 0 and 1 1 1. This example will demonstrate how the minimum number of bit flips can be uniquely decoded.

Consider the following sequences and their decoding outcomes:

- 0 0 0
- 0 0 1
- 0 1 0
- 0 1 1
- 1 0 0
- 1 0 1
- 1 1 0
- 1 1 1

In this 3-bit repetition code, the decoding is straightforward:

For sequences 0 0 0 and 1 1 1, the code word is clear and unambiguous. This is because the most probable event, given a low probability p of errors (where p is less than 0.5), is that there are no errors. Therefore, sequences with no bit flips or a single bit flip can be decoded uniquely.

For example, if you start with 0 0 1 and perform a one-bit flip:

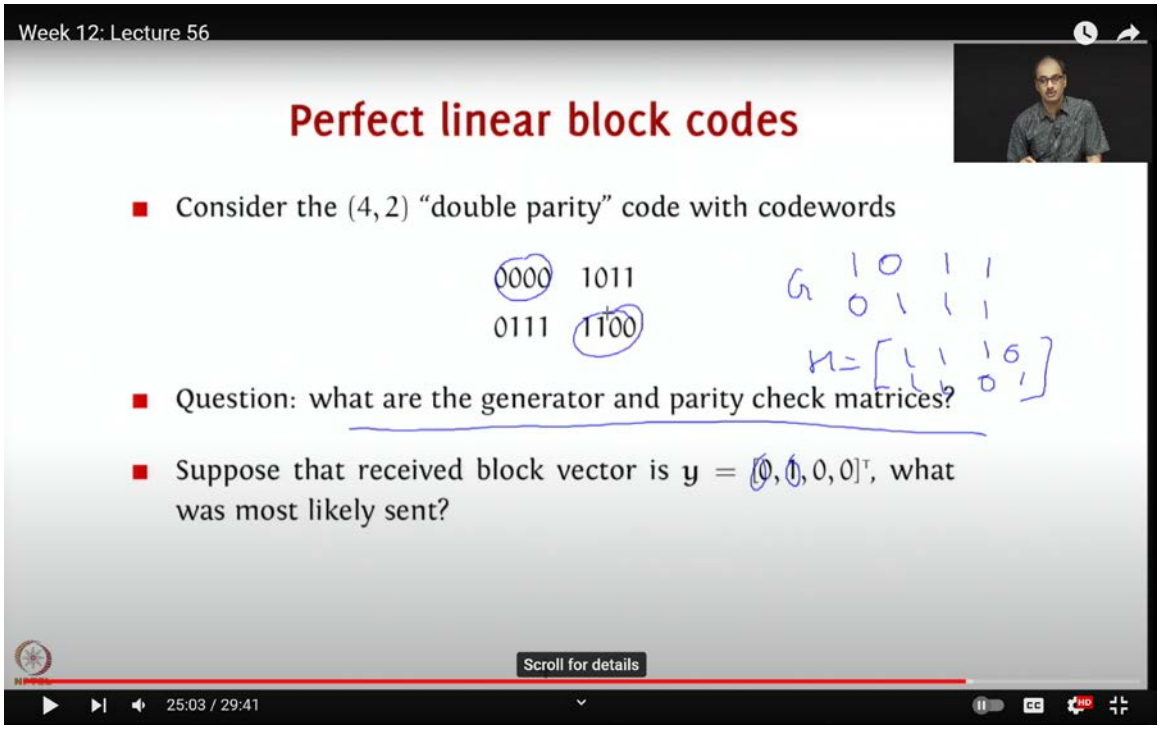
- Flipping the first bit yields 1 0 1, which is not a code word.
- Flipping the second bit results in 0 1 1, which is not a code word.
- Flipping the third bit yields 0 0 0, which is a valid code word.

Similarly, for 1 1 0:

- Flipping the first bit results in 0 1 0, which is not a code word.
- Flipping the second bit results in 1 0 0, which is not a code word.
- Flipping the third bit yields 1 1 1, which is a valid code word.

In this manner, each of the $2^3 = 8$ possible sequences is mapped to a single, unique code word, avoiding ambiguity in decoding. This is in contrast to other codes, such as repetition codes or parity check codes, where single-bit flips or multiple-bit flips can lead to ambiguous results.

(Refer Slide Time: 25:03)



Now, let's discuss the generator and parity check matrices for this code. The generator matrix for the 3-bit repetition code is straightforward and is essentially an identity matrix. The parity check matrix is also derived from the generator matrix and is used to ensure error detection and correction.

We should also introduce some key concepts: Hamming weight and Hamming distance.

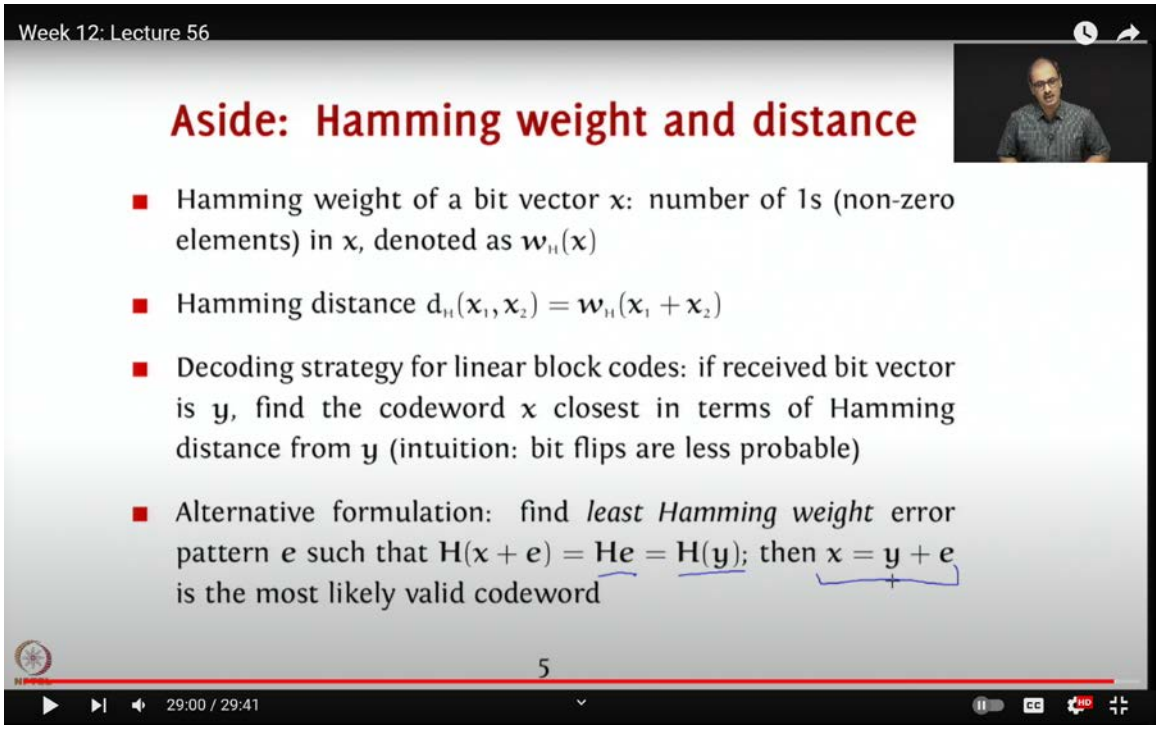
- Hamming Weight: The Hamming weight of a bit vector X is the number of nonzero

elements or ones in X. For instance, if X = 0 1 0, its weight is 1 because there is one '1'. If X = 1 1 1, its weight is 3, and for X = 0 0 0, the weight is 0.

- Hamming Distance: The Hamming distance between two n-bit sequences X_1 and X_2 is the number of bits in which they differ. To find this, perform a bitwise XOR operation between X_1 and X_2, then count the number of ones in the result. For example, for X_1 = 0 1 0 and X_2 = 0 1 1, XORing these sequences gives 0 0 1. Counting the ones, we get a Hamming distance of 1, indicating that the sequences differ in one bit.

Here's a refined explanation of decoding strategies for linear block codes, which can also simplify computer implementation:

(Refer Slide Time: 29:00)



To begin with, let’s consider the Hamming distance, denoted as $D_H(X_1, X_2)$. This distance between two bit vectors X_1 and X_2 can be computed as the Hamming weight of their XOR, $X_1 \oplus X_2$.

Now, let’s outline the decoding strategy for linear block codes. Suppose you receive a bit

vector Y, and X is a valid code word. The vector Y represents X with added noise. The goal is to find the code word X that is closest to Y in terms of Hamming distance. In other words, we want to determine the minimum number of bit flips required to transform Y into a valid code word X.

Why is this strategy optimal? In a binary symmetric channel with a low probability p of errors (where $p < 0.5$), the most likely event is that no errors have occurred, meaning no bit flips are necessary. The next most probable scenario is that a single-bit error has occurred, so we would flip one bit and check whether the result is a valid code word. If a single-bit flip doesn't yield a code word, we then consider two-bit errors, and so on, to find the modification that results in the code word closest to X in terms of Hamming distance.

Alternatively, we can formulate this problem as finding the least Hamming weight error pattern e such that $H(X + e) = H(Y)$, where H is the parity check matrix. In this case, X is the most likely valid code word if $X = Y + e$. Essentially, we are looking for the smallest number of bit flips (or smallest Hamming weight vector e) that corrects the received sequence Y to match a valid code word X.

If no bit errors occurred, e would be the all-zeros vector. For a single-bit error, we would test vectors like 1 0 0, 0 1 0, 0 0 1, and determine which results in a valid code word by checking if $H(X + e) = H(Y)$.

If the solution is unambiguous, as it is with the 3-bit repetition code, then error correction is straightforward. We will delve deeper into these concepts and discuss single-error correcting codes, such as Hamming codes, in the next lecture. Thank you.