Code Generation

Daniel H. Luecking
MASC

November 10, 2023

Producing a group code

One way to look at the set of messages $\mathbb{Z}_2^m$ is to think of it as being all possible sums of certain basic strings.

Producing a group code

One way to look at the set of messages $\mathbb{Z}_2^m$ is to think of it as being all possible sums of certain basic strings. For a small example, you can take $\mathbb{Z}_2^3$ and get the 7 nonzero strings by summing combinations of the 3 strings $(100), (010), (001)$.

Producing a group code

One way to look at the set of messages $\mathbb{Z}_2^m$ is to think of it as being all possible sums of certain basic strings. For a small example, you can take $\mathbb{Z}_2^3$ and get the 7 nonzero strings by summing combinations of the 3 strings $(100), (010), (001)$. We get (000) if we add any of these to itself.

Producing a group code

One way to look at the set of messages $\mathbb{Z}_2^m$ is to think of it as being all possible sums of certain basic strings. For a small example, you can take $\mathbb{Z}_2^3$ and get the 7 nonzero strings by summing combinations of the 3 strings $(100), (010), (001)$. We get (000) if we add any of these to itself. If we perform an encoding by appending bits, we can ensure we get a group by appending bits to just these 3 and then taking all possible sums of the results.

Producing a group code

One way to look at the set of messages $\mathbb{Z}_2^m$ is to think of it as being all possible sums of certain basic strings. For a small example, you can take $\mathbb{Z}_2^3$ and get the 7 nonzero strings by summing combinations of the 3 strings $(100), (010), (001)$. We get (000) if we add any of these to itself. If we perform an encoding by appending bits, we can ensure we get a group by appending bits to just these 3 and then taking all possible sums of the results. For example if I append as follows:

$$(100) \to (100\,101) \quad (010) \to (010\,110) \quad (001) \to (001\,011)$$

I get the example $\mathscr{C}$ we introduced last lecture.

Producing a group code

One way to look at the set of messages $\mathbb{Z}_2^m$ is to think of it as being all possible sums of certain basic strings. For a small example, you can take $\mathbb{Z}_2^3$ and get the 7 nonzero strings by summing combinations of the 3 strings $(100), (010), (001)$. We get (000) if we add any of these to itself. If we perform an encoding by appending bits, we can ensure we get a group by appending bits to just these 3 and then taking all possible sums of the results. For example if I append as follows:

$$(100) \to (100\,101) \quad (010) \to (010\,110) \quad (001) \to (001\,011)$$

I get the example $\mathscr{C}$ we introduced last lecture. For example

$$(110\,011) = (100\,101) + (010\,110) \text{ and}$$
$$(111\,000) = (100\,101) + (010\,110) + (001\,011).$$

Producing a group code with error detection

Of course we can't just randomly append any bits, they have to be chosen to fulfill the function of a code: error detection and correction.

Producing a group code with error detection

Of course we can't just randomly append any bits, they have to be chosen to fulfill the function of a code: error detection and correction.

To be able to detect one error we need a code where the minimum distance between code words is at least 2.

Producing a group code with error detection

Of course we can't just randomly append any bits, they have to be chosen to fulfill the function of a code: error detection and correction.

To be able to detect one error we need a code where the minimum distance between code words is at least 2. The ASCII even-parity code does this because a string with an even number of 1s will have an odd number if only a single bit is changed.

Producing a group code with error detection

Of course we can't just randomly append any bits, they have to be chosen to fulfill the function of a code: error detection and correction.

To be able to detect one error we need a code where the minimum distance between code words is at least 2. The ASCII even-parity code does this because a string with an even number of 1s will have an odd number if only a single bit is changed. The ASCII even-parity code is a group code that can be obtained by the process discussed on the previous slide.

Producing a group code with error detection

Of course we can't just randomly append any bits, they have to be chosen to fulfill the function of a code: error detection and correction.

To be able to detect one error we need a code where the minimum distance between code words is at least 2. The ASCII even-parity code does this because a string with an even number of 1s will have an odd number if only a single bit is changed. The ASCII even-parity code is a group code that can be obtained by the process discussed on the previous slide.

To get one-bit error correction we need a minimum distance between code words of at least 3.

Producing a group code with error detection

Of course we can't just randomly append any bits, they have to be chosen to fulfill the function of a code: error detection and correction.

To be able to detect one error we need a code where the minimum distance between code words is at least 2. The ASCII even-parity code does this because a string with an even number of 1s will have an odd number if only a single bit is changed. The ASCII even-parity code is a group code that can be obtained by the process discussed on the previous slide.

To get one-bit error correction we need a minimum distance between code words of at least 3. For a group code, the minimum distance is the minimum weight of the nonzero code words,

Producing a group code with error detection

Of course we can't just randomly append any bits, they have to be chosen to fulfill the function of a code: error detection and correction.

To be able to detect one error we need a code where the minimum distance between code words is at least 2. The ASCII even-parity code does this because a string with an even number of 1s will have an odd number if only a single bit is changed. The ASCII even-parity code is a group code that can be obtained by the process discussed on the previous slide.

To get one-bit error correction we need a minimum distance between code words of at least 3. For a group code, the minimum distance is the minimum weight of the nonzero code words, so we want to append bits with enough 1s in them to give us the minimum weight we want

Producing a group code with error detection

Of course we can't just randomly append any bits, they have to be chosen to fulfill the function of a code: error detection and correction.

To be able to detect one error we need a code where the minimum distance between code words is at least 2. The ASCII even-parity code does this because a string with an even number of 1s will have an odd number if only a single bit is changed. The ASCII even-parity code is a group code that can be obtained by the process discussed on the previous slide.

To get one-bit error correction we need a minimum distance between code words of at least 3. For a group code, the minimum distance is the minimum weight of the nonzero code words, so we want to append bits with enough 1s in them to give us the minimum weight we want

[To be able to discuss general cases we need a notation for the strings with a single 1 in them. So we let e_j stand for the string of all 0s except for a 1 in position j.]

A way to do this efficiently: Generator matrices

For our example $\mathbb{Z}_2^3$, the 3 basic strings were e_1, e_2, e_3.

A way to do this efficiently: Generator matrices

For our example $\mathbb{Z}_2^3$, the 3 basic strings were e_1, e_2, e_3. These all have weight 1 and so if we append bits, then the string we append has to have enough weight to bring the total weight up to the level we want.

A way to do this efficiently: Generator matrices

For our example $\mathbb{Z}_2^3$, the 3 basic strings were e_1, e_2, e_3. These all have weight 1 and so if we append bits, then the string we append has to have enough weight to bring the total weight up to the level we want. For example if we want the minimum weight to be 3 we need at least two more 1s.

A way to do this efficiently: Generator matrices

For our example $\mathbb{Z}_2^3$, the 3 basic strings were e_1, e_2, e_3. These all have weight 1 and so if we append bits, then the string we append has to have enough weight to bring the total weight up to the level we want. For example if we want the minimum weight to be 3 we need at least two more 1s. Consider the matrix

$$G = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array} \right)$$

Its rows are exactly the elements e_1, e_2, e_3 from $\mathbb{Z}_2^3$, each with the bits from before appended.

A way to do this efficiently: Generator matrices

For our example $\mathbb{Z}_2^3$, the 3 basic strings were e_1, e_2, e_3. These all have weight 1 and so if we append bits, then the string we append has to have enough weight to bring the total weight up to the level we want. For example if we want the minimum weight to be 3 we need at least two more 1s. Consider the matrix

$$G = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right)$$

Its rows are exactly the elements e_1, e_2, e_3 from $\mathbb{Z}_2^3$, each with the bits from before appended. To get all possible sums of these rows (plus the zero string) we can do matrix multiplication.

A way to do this efficiently: Generator matrices

For our example $\mathbb{Z}_2^3$, the 3 basic strings were e_1, e_2, e_3. These all have weight 1 and so if we append bits, then the string we append has to have enough weight to bring the total weight up to the level we want. For example if we want the minimum weight to be 3 we need at least two more 1s. Consider the matrix

$$G = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right)$$

Its rows are exactly the elements e_1, e_2, e_3 from $\mathbb{Z}_2^3$, each with the bits from before appended. To get all possible sums of these rows (plus the zero string) we can do matrix multiplication. That is, to get the sum of the first two rows, multiply by (110):

$$(110)G = (100101) + (010110) = (110011)$$

You should know matrix multiplication for real numbers; this is just like that except the operations are performed in $\mathbb{Z}_2$: that is, mod 2.

You should know matrix multiplication for real numbers; this is just like that except the operations are performed in $\mathbb{Z}_2$: that is, mod 2. If you have trouble with matrix multiplication, no worries: to find the product wG, just add the rows in G that correspond to the position of 1s in w (adding mod 2).

You should know matrix multiplication for real numbers; this is just like that except the operations are performed in $\mathbb{Z}_2$: that is, mod 2. If you have trouble with matrix multiplication, no worries: to find the product wG, just add the rows in G that correspond to the position of 1s in w (adding mod 2). For example $(011)G$ is the sum (mod 2) of the 2nd and 3rd rows of G.

You should know matrix multiplication for real numbers; this is just like that except the operations are performed in $\mathbb{Z}_2$: that is, mod 2. If you have trouble with matrix multiplication, no worries: to find the product wG, just add the rows in G that correspond to the position of 1s in w (adding mod 2). For example $(011)G$ is the sum (mod 2) of the 2nd and 3rd rows of G. The case $(000)G$ produces a string of just 0s.

You should know matrix multiplication for real numbers; this is just like that except the operations are performed in $\mathbb{Z}_2$: that is, mod 2. If you have trouble with matrix multiplication, no worries: to find the product wG, just add the rows in G that correspond to the position of 1s in w (adding mod 2). For example $(011)G$ is the sum (mod 2) of the 2nd and 3rd rows of G. The case $(000)G$ produces a string of just 0s. The set of all possible wG is the entire code.

You should know matrix multiplication for real numbers; this is just like that except the operations are performed in $\mathbb{Z}_2$: that is, mod 2. If you have trouble with matrix multiplication, no worries: to find the product wG, just add the rows in G that correspond to the position of 1s in w (adding mod 2). For example $(011)G$ is the sum (mod 2) of the 2nd and 3rd rows of G. The case $(000)G$ produces a string of just 0s. The set of all possible wG is the entire code.

The vertical bar in our example G is just to visually separate the two distinctive parts of our matrix: the basic elements of $\mathbb{Z}_2^3$ on the left, the added bits on the right.

You should know matrix multiplication for real numbers; this is just like that except the operations are performed in $\mathbb{Z}_2$: that is, mod 2. If you have trouble with matrix multiplication, no worries: to find the product wG, just add the rows in G that correspond to the position of 1s in w (adding mod 2). For example $(011)G$ is the sum (mod 2) of the 2nd and 3rd rows of G. The case $(000)G$ produces a string of just 0s. The set of all possible wG is the entire code.

The vertical bar in our example G is just to visually separate the two distinctive parts of our matrix: the basic elements of $\mathbb{Z}_2^3$ on the left, the added bits on the right.

Here's another example, which generates a code in $\mathbb{Z}_2^7$ for message words from $\mathbb{Z}_2^4$:

You should know matrix multiplication for real numbers; this is just like that except the operations are performed in $\mathbb{Z}_2$: that is, mod 2. If you have trouble with matrix multiplication, no worries: to find the product wG, just add the rows in G that correspond to the position of 1s in w (adding mod 2). For example $(011)G$ is the sum (mod 2) of the 2nd and 3rd rows of G. The case $(000)G$ produces a string of just 0s. The set of all possible wG is the entire code.

The vertical bar in our example G is just to visually separate the two distinctive parts of our matrix: the basic elements of $\mathbb{Z}_2^3$ on the left, the added bits on the right.

Here's another example, which generates a code in $\mathbb{Z}_2^7$ for message words from $\mathbb{Z}_2^4$:

$$G = \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array} \right)$$

Structure of the Generating matrix

Structure of the Generating matrix

Both of our examples have the following form $G = (I \mid A)$, where I is an *identity matrix*.

Structure of the Generating matrix

Both of our examples have the following form $G = (I \mid A)$, where I is an *identity matrix*. The identity matrix has m rows and m columns and consist mostly of 0s except for a diagonal of 1s from upper left to lower right.

Structure of the Generating matrix

Both of our examples have the following form $G = (I \mid A)$, where I is an *identity matrix*. The identity matrix has m rows and m columns and consist mostly of 0s except for a diagonal of 1s from upper left to lower right. Its purpose is to take a w in $\mathbb{Z}_2^m$ and reproduce it exactly.

Structure of the Generating matrix

Both of our examples have the following form $G = (I \mid A)$, where I is an *identity matrix*. The identity matrix has m rows and m columns and consist mostly of 0s except for a diagonal of 1s from upper left to lower right. Its purpose is to take a w in $\mathbb{Z}_2^m$ and reproduce it exactly. That is, wG will be a string whose length is the same as the length of the rows of G, but the first m bits will be a copy of w.

Structure of the Generating matrix

Both of our examples have the following form $G = (I \mid A)$, where I is an *identity matrix*. The identity matrix has m rows and m columns and consist mostly of 0s except for a diagonal of 1s from upper left to lower right. Its purpose is to take a w in $\mathbb{Z}_2^m$ and reproduce it exactly. That is, wG will be a string whose length is the same as the length of the rows of G, but the first m bits will be a copy of w.

The matrix A on the right of G is the part that appends parity bits, (i.e., it serves to produce the rest of wG, after the copy of w).

Structure of the Generating matrix

Both of our examples have the following form $G = (I \mid A)$, where I is an *identity matrix*. The identity matrix has m rows and m columns and consist mostly of 0s except for a diagonal of 1s from upper left to lower right. Its purpose is to take a w in $\mathbb{Z}_2^m$ and reproduce it exactly. That is, wG will be a string whose length is the same as the length of the rows of G, but the first m bits will be a copy of w.

The matrix A on the right of G is the part that appends parity bits, (i.e., it serves to produce the rest of wG, after the copy of w). Let's write a formula for wG using the example G from the previous slide and writing $w = (w_1w_2w_3w_4)$ where each w_j are either a 0 or a 1.

Structure of the Generating matrix

Both of our examples have the following form $G = (I \mid A)$, where I is an *identity matrix*. The identity matrix has m rows and m columns and consist mostly of 0s except for a diagonal of 1s from upper left to lower right. Its purpose is to take a w in $\mathbb{Z}_2^m$ and reproduce it exactly. That is, wG will be a string whose length is the same as the length of the rows of G, but the first m bits will be a copy of w.

The matrix A on the right of G is the part that appends parity bits, (i.e., it serves to produce the rest of wG, after the copy of w). Let's write a formula for wG using the example G from the previous slide and writing $w = (w_1 w_2 w_3 w_4)$ where each w_j are either a 0 or a 1.

$$(w_1\, w_2\, w_3\, w_4) \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right)$$
$$= (w_1\, w_2\, w_3\, w_4 (w_1 + w_2 + w_4)(w_2 + w_3 + w_4)(w_1 + w_3 + w_4))$$

The extra bits, for example $w_1 + w_2 + w_4$ in the 5th position, are called parity bits because the effect they have is that the bits of wG in certain positions must have an even number of 1s.

The extra bits, for example $w_1 + w_2 + w_4$ in the 5th position, are called parity bits because the effect they have is that the bits of wG in certain positions must have an even number of 1s. For example the 1st, 2nd, 4th and 5th position will have an even number of 1s.

The extra bits, for example $w_1 + w_2 + w_4$ in the 5th position, are called parity bits because the effect they have is that the bits of wG in certain positions must have an even number of 1s. For example the 1st, 2nd, 4th and 5th position will have an even number of 1s.

This formula allows us to take a received word $r = (r_1\, r_2\, r_3\, r_4\, r_5\, r_6\, r_7)$ and test whether it is a code word.

The extra bits, for example $w_1 + w_2 + w_4$ in the 5th position, are called parity bits because the effect they have is that the bits of wG in certain positions must have an even number of 1s. For example the 1st, 2nd, 4th and 5th position will have an even number of 1s.

This formula allows us to take a received word $r = (r_1\, r_2\, r_3\, r_4\, r_5\, r_6\, r_7)$ and test whether it is a code word. If all 7 bits have survived unchanged, then $(r_1\, r_2\, r_3\, r_4)G$ must equal r. That means that r_5 must be $r_1 + r_2 + r_4$ and so on.

The extra bits, for example $w_1 + w_2 + w_4$ in the 5th position, are called parity bits because the effect they have is that the bits of wG in certain positions must have an even number of 1s. For example the 1st, 2nd, 4th and 5th position will have an even number of 1s.

This formula allows us to take a received word $r = (r_1\, r_2\, r_3\, r_4\, r_5\, r_6\, r_7)$ and test whether it is a code word. If all 7 bits have survived unchanged, then $(r_1\, r_2\, r_3\, r_4)G$ must equal r. That means that r_5 must be $r_1 + r_2 + r_4$ and so on. Thus the receiver's test for correctness is to check the following equations (parity check):

$$r_1 + r_2 + r_4 = r_5$$
$$r_2 + r_3 + r_4 = r_6$$
$$r_1 + r_3 + r_4 = r_7$$

The extra bits, for example $w_1 + w_2 + w_4$ in the 5th position, are called parity bits because the effect they have is that the bits of wG in certain positions must have an even number of 1s. For example the 1st, 2nd, 4th and 5th position will have an even number of 1s.

This formula allows us to take a received word $r = (r_1\, r_2\, r_3\, r_4\, r_5\, r_6\, r_7)$ and test whether it is a code word. If all 7 bits have survived unchanged, then $(r_1\, r_2\, r_3\, r_4)G$ must equal r. That means that r_5 must be $r_1 + r_2 + r_4$ and so on. Thus the receiver's test for correctness is to check the following equations (parity check):

$$
\begin{aligned}
r_1 + r_2 + r_4 &= r_5 \\
r_2 + r_3 + r_4 &= r_6 \\
r_1 + r_3 + r_4 &= r_7
\end{aligned}
$$

This is easier to coordinate if we rearrange it, using the fact that in $\mathbb{Z}_2$, $x + x$ is always zero.

Thus if we add r_5 to both sides of the first equation, the right side becomes 0, etc.

Thus if we add r_5 to both sides of the first equation, the right side becomes 0, etc. These parity check equation become

$$\begin{aligned}
r_1 + r_2 \quad\;\; + r_4 + r_5 \qquad\qquad &= 0 \\
r_2 + r_3 + r_4 \quad\;\; + r_6 \qquad &= 0 \\
r_1 \quad\;\; + r_3 + r_4 \qquad\qquad + r_7 &= 0
\end{aligned}$$

Thus if we add r_5 to both sides of the first equation, the right side becomes 0, etc. These parity check equation become

$$\begin{aligned}
r_1 + r_2 \quad\quad + r_4 + r_5 \quad\quad\quad\quad &= 0 \\
r_2 + r_3 + r_4 \quad\quad + r_6 \quad\quad &= 0 \\
r_1 \quad\quad + r_3 + r_4 \quad\quad\quad\quad + r_7 &= 0
\end{aligned}$$

Each of these sums says that an even number of the variables have to be 1s

Thus if we add r_5 to both sides of the first equation, the right side becomes 0, etc. These parity check equation become

$$\begin{aligned} r_1 + r_2 \quad\quad + r_4 + r_5 \quad\quad\quad &= 0 \\ r_2 + r_3 + r_4 \quad\quad + r_6 \quad\quad &= 0 \\ r_1 \quad\quad + r_3 + r_4 \quad\quad\quad\quad + r_7 &= 0 \end{aligned}$$

Each of these sums says that an even number of the variables have to be 1s

These equations can be written in matrix form as

$$\left(\begin{array}{cccc|ccc} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right) \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

The equation on the previous slide could also be rewritten by taking the transpose:

$$(r_1\ r_2\ r_3\ r_4\ r_5\ r_6\ r_7)\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$$

But our book has chosen to use the other way.

The leftmost matrix in the previous equation is denoted H and is called the *parity-check matrix* for G.

The leftmost matrix in the previous equation is denoted H and is called the *parity-check matrix* for G. It can easily be written down just by examining G. If $G = (I \mid A)$ then $H = (A^{\mathrm{tr}} \mid I)$.

The leftmost matrix in the previous equation is denoted H and is called the *parity-check matrix* for G. It can easily be written down just by examining G. If $G = (I \mid A)$ then $H = (A^{\mathrm{tr}} \mid I)$. The notation A^{tr} means the *transpose* of A and it is obtained by taking each row of A, starting at the top, and writing its entries in a column.

The leftmost matrix in the previous equation is denoted H and is called the *parity-check matrix* for G. It can easily be written down just by examining G. If $G = (I \mid A)$ then $H = (A^{\mathrm{tr}} \mid I)$. The notation A^{tr} means the *transpose* of A and it is obtained by taking each row of A, starting at the top, and writing its entries in a column. The first (top) row of A becomes the first (leftmost) column of A^{tr}.

The leftmost matrix in the previous equation is denoted H and is called the *parity-check matrix* for G. It can easily be written down just by examining G. If $G = (I \mid A)$ then $H = (A^{\mathrm{tr}} \mid I)$. The notation A^{tr} means the *transpose* of A and it is obtained by taking each row of A, starting at the top, and writing its entries in a column. The first (top) row of A becomes the first (leftmost) column of A^{tr}. For example

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}^{\mathrm{tr}} = \begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix}.$$

The leftmost matrix in the previous equation is denoted H and is called the *parity-check matrix* for G. It can easily be written down just by examining G. If $G = (I \mid A)$ then $H = (A^{\mathrm{tr}} \mid I)$. The notation A^{tr} means the *transpose* of A and it is obtained by taking each row of A, starting at the top, and writing its entries in a column. The first (top) row of A becomes the first (leftmost) column of A^{tr}. For example

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}^{\mathrm{tr}} = \begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix}.$$

To save space, we often write a single column matrix, like the one containing r_1 through r_7 on the previous slide, as the transpose of a row: $(r_1 r_2 r_3 r_4 r_5 r_6 r_7)^{\mathrm{tr}}$.

The leftmost matrix in the previous equation is denoted H and is called the *parity-check matrix* for G. It can easily be written down just by examining G. If $G = (I \mid A)$ then $H = (A^{\text{tr}} \mid I)$. The notation A^{tr} means the *transpose* of A and it is obtained by taking each row of A, starting at the top, and writing its entries in a column. The first (top) row of A becomes the first (leftmost) column of A^{tr}. For example

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}^{\text{tr}} = \begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix}.$$

To save space, we often write a single column matrix, like the one containing r_1 through r_7 on the previous slide, as the transpose of a row: $(r_1 r_2 r_3 r_4 r_5 r_6 r_7)^{\text{tr}}$. Thus, our parity-check procedure is to compute Hr^{tr} and check if it is a column of 0s.

Note that multiplying a matrix by a column (in that order) is different than multiplying a row times a matrix. To compute Hr^{tr} you must add up the columns in H that correspond to the position of 1s in r.

Note that multiplying a matrix by a column (in that order) is different than multiplying a row times a matrix. To compute Hr^{tr} you must add up the columns in H that correspond to the position of 1s in r.

Here is a new example where all these ideas are worked out in some detail.

Note that multiplying a matrix by a column (in that order) is different than multiplying a row times a matrix. To compute Hr^{tr} you must add up the columns in H that correspond to the position of 1s in r.

Here is a new example where all these ideas are worked out in some detail. If our code generator matrix G is

$$G = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{array} \right) \quad \text{then} \quad A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

Note that multiplying a matrix by a column (in that order) is different than multiplying a row times a matrix. To compute Hr^{tr} you must add up the columns in H that correspond to the position of 1s in r.

Here is a new example where all these ideas are worked out in some detail. If our code generator matrix G is

$$G = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{array} \right) \quad \text{then} \quad A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

And so,

$$A^{\mathrm{tr}} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \quad \text{and} \quad H = \left(\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{array} \right)$$

Now suppose we and a remote site have arranged to use this code to send 3-bit messages, and I want to send $w = (101)$.

Now suppose we and a remote site have arranged to use this code to send 3-bit messages, and I want to send $w = (101)$. The first step is to encode it:

$$wG = (101011)$$

So the code word is $c = (101011)$.

Now suppose we and a remote site have arranged to use this code to send 3-bit messages, and I want to send $w = (101)$. The first step is to encode it:

$$wG = (101011)$$

So the code word is $c = (101011)$. Suppose, when we transmit this c to our remote site it arrives as $r = (100011)$ with an error in the 3rd position.

Now suppose we and a remote site have arranged to use this code to send 3-bit messages, and I want to send $w = (101)$. The first step is to encode it:

$$wG = (101011)$$

So the code word is $c = (101011)$. Suppose, when we transmit this c to our remote site it arrives as $r = (100011)$ with an error in the 3rd position. Our colleagues there will check it by multiplication:

$$Hr^{\mathrm{tr}} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \quad \text{(or } (101)^{\mathrm{tr}}\text{)}.$$

Now suppose we and a remote site have arranged to use this code to send 3-bit messages, and I want to send $w = (101)$. The first step is to encode it:

$$wG = (101011)$$

So the code word is $c = (101011)$. Suppose, when we transmit this c to our remote site it arrives as $r = (100011)$ with an error in the 3rd position. Our colleagues there will check it by multiplication:

$$Hr^{\mathrm{tr}} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \quad \text{(or } (101)^{\mathrm{tr}}\text{)}.$$

Since the result is not $(000)^{\mathrm{tr}}$ they know there is an error.

This is a correctable error.

This is a correctable error. They know this because of the following: if the rows of A all have weight at least 2 and if they are all different, then $G = (I \mid A)$ produces a code with minimum nonzero weight at least 3.

This is a correctable error. They know this because of the following: if the rows of A all have weight at least 2 and if they are all different, then $G = (I \mid A)$ produces a code with minimum nonzero weight at least 3. Our example G satisfies all that, so our colleagues only have to find the closest code word to r to correct it.

This is a correctable error. They know this because of the following: if the rows of A all have weight at least 2 and if they are all different, then $G = (I \mid A)$ produces a code with minimum nonzero weight at least 3. Our example G satisfies all that, so our colleagues only have to find the closest code word to r to correct it. This is pretty simple here, because there are only eight words in the code.

This is a correctable error. They know this because of the following: if the rows of A all have weight at least 2 and if they are all different, then $G = (I \mid A)$ produces a code with minimum nonzero weight at least 3. Our example G satisfies all that, so our colleagues only have to find the closest code word to r to correct it. This is pretty simple here, because there are only eight words in the code. If our messages had been 64 bits long then there would be 2^{64} code words, too many to check.

This is a correctable error. They know this because of the following: if the rows of A all have weight at least 2 and if they are all different, then $G = (I \mid A)$ produces a code with minimum nonzero weight at least 3. Our example G satisfies all that, so our colleagues only have to find the closest code word to r to correct it. This is pretty simple here, because there are only eight words in the code. If our messages had been 64 bits long then there would be 2^{64} code words, too many to check.

They are saved from having to do those comparisons by the following fact: if r differs from c in a single bit then $r = c + e$ where e has only a single 1.

This is a correctable error. They know this because of the following: if the rows of A all have weight at least 2 and if they are all different, then $G = (I \mid A)$ produces a code with minimum nonzero weight at least 3. Our example G satisfies all that, so our colleagues only have to find the closest code word to r to correct it. This is pretty simple here, because there are only eight words in the code. If our messages had been 64 bits long then there would be 2^{64} code words, too many to check.

They are saved from having to do those comparisons by the following fact: if r differs from c in a single bit then $r = c + e$ where e has only a single 1. In this example $r = c + e_3$ where recall e_3 means (001000).

This is a correctable error. They know this because of the following: if the rows of A all have weight at least 2 and if they are all different, then $G = (I \mid A)$ produces a code with minimum nonzero weight at least 3. Our example G satisfies all that, so our colleagues only have to find the closest code word to r to correct it. This is pretty simple here, because there are only eight words in the code. If our messages had been 64 bits long then there would be 2^{64} code words, too many to check.

They are saved from having to do those comparisons by the following fact: if r differs from c in a single bit then $r = c + e$ where e has only a single 1. In this example $r = c + e_3$ where recall e_3 means (001000). Our colleagues don't know what c is, nor that the error is e_3, but they do know that if $r = c + e$ then $Hr^{\text{tr}} = Hc^{\text{tr}} + He^{\text{tr}} = He^{\text{tr}}$, because Hc^{tr} is a column of zeros for code words.

But if they know (or assume) that e has only a single 1, then by our rules for matrix multiplication He^{tr} will be the column of H corresponding to the position of the 1 in e.

But if they know (or assume) that e has only a single 1, then by our rules for matrix multiplication He^{tr} will be the column of H corresponding to the position of the 1 in e. In our actual example $Hr^{\mathrm{tr}} = He^{\mathrm{tr}} = (101)^{\mathrm{tr}}$ is the 3rd column of H. So that pinpoints the error.

But if they know (or assume) that e has only a single 1, then by our rules for matrix multiplication He^{tr} will be the column of H corresponding to the position of the 1 in e. In our actual example $Hr^{\mathrm{tr}} = He^{\mathrm{tr}} = (101)^{\mathrm{tr}}$ is the 3rd column of H. So that pinpoints the error. Our colleagues simply change the 3rd bit of r from 0 to 1 and get the correct $c = (101011)$.

But if they know (or assume) that e has only a single 1, then by our rules for matrix multiplication He^{tr} will be the column of H corresponding to the position of the 1 in e. In our actual example $Hr^{\mathrm{tr}} = He^{\mathrm{tr}} = (101)^{\mathrm{tr}}$ is the 3rd column of H. So that pinpoints the error. Our colleagues simply change the 3rd bit of r from 0 to 1 and get the correct $c = (101011)$.

Finally, the correct word has to be decoded: remove the parity bits.

But if they know (or assume) that e has only a single 1, then by our rules for matrix multiplication He^{tr} will be the column of H corresponding to the position of the 1 in e. In our actual example $Hr^{\mathrm{tr}} = He^{\mathrm{tr}} = (101)^{\mathrm{tr}}$ is the 3rd column of H. So that pinpoints the error. Our colleagues simply change the 3rd bit of r from 0 to 1 and get the correct $c = (101011)$.

Finally, the correct word has to be decoded: remove the parity bits. Since G adds 3 bits they remove the last 3, which have done their job, to get the message I wanted them to have: (101).

But if they know (or assume) that e has only a single 1, then by our rules for matrix multiplication He^{tr} will be the column of H corresponding to the position of the 1 in e. In our actual example $Hr^{\mathrm{tr}} = He^{\mathrm{tr}} = (101)^{\mathrm{tr}}$ is the 3rd column of H. So that pinpoints the error. Our colleagues simply change the 3rd bit of r from 0 to 1 and get the correct $c = (101011)$.

Finally, the correct word has to be decoded: remove the parity bits. Since G adds 3 bits they remove the last 3, which have done their job, to get the message I wanted them to have: (101).

So the rule at the receiving end is: multiply Hr^{tr} then

1. If the result is all zeros, accept r as correct.

But if they know (or assume) that e has only a single 1, then by our rules for matrix multiplication He^{tr} will be the column of H corresponding to the position of the 1 in e. In our actual example $Hr^{\mathrm{tr}} = He^{\mathrm{tr}} = (101)^{\mathrm{tr}}$ is the 3rd column of H. So that pinpoints the error. Our colleagues simply change the 3rd bit of r from 0 to 1 and get the correct $c = (101011)$.

Finally, the correct word has to be decoded: remove the parity bits. Since G adds 3 bits they remove the last 3, which have done their job, to get the message I wanted them to have: (101).

So the rule at the receiving end is: multiply Hr^{tr} then

1. If the result is all zeros, accept r as correct.
2. If the result is one of the columns of H, correct r by changing the bit in the corresponding position of r.

But if they know (or assume) that e has only a single 1, then by our rules for matrix multiplication He^{tr} will be the column of H corresponding to the position of the 1 in e. In our actual example $Hr^{\mathrm{tr}} = He^{\mathrm{tr}} = (101)^{\mathrm{tr}}$ is the 3rd column of H. So that pinpoints the error. Our colleagues simply change the 3rd bit of r from 0 to 1 and get the correct $c = (101011)$.

Finally, the correct word has to be decoded: remove the parity bits. Since G adds 3 bits they remove the last 3, which have done their job, to get the message I wanted them to have: (101).

So the rule at the receiving end is: multiply Hr^{tr} then

1. If the result is all zeros, accept r as correct.
2. If the result is one of the columns of H, correct r by changing the bit in the corresponding position of r.
3. If the result is anything else, then r cannot be corrected (and maybe ask the senders to try again).

But if they know (or assume) that e has only a single 1, then by our rules for matrix multiplication He^{tr} will be the column of H corresponding to the position of the 1 in e. In our actual example $Hr^{\mathrm{tr}} = He^{\mathrm{tr}} = (101)^{\mathrm{tr}}$ is the 3rd column of H. So that pinpoints the error. Our colleagues simply change the 3rd bit of r from 0 to 1 and get the correct $c = (101011)$.

Finally, the correct word has to be decoded: remove the parity bits. Since G adds 3 bits they remove the last 3, which have done their job, to get the message I wanted them to have: (101).

So the rule at the receiving end is: multiply Hr^{tr} then

1. If the result is all zeros, accept r as correct.
2. If the result is one of the columns of H, correct r by changing the bit in the corresponding position of r.
3. If the result is anything else, then r cannot be corrected (and maybe ask the senders to try again).
4. In case 1 or 2, they have the correct code word; the original message is found by removing the parity bits that were added by the encoding.

If we need only one-bit error correction it can be shown that the number of bits to add to m-bit messages is a little more than $\log_2 m$.

If we need only one-bit error correction it can be shown that the number of bits to add to m-bit messages is a little more than $\log_2 m$. For example, 64-bit messages need add only 7 bits. Then $G = (I \mid A)$ has 64 rows and 71 columns where I has 64 rows and columns and A has 64 rows and 7 columns.

If we need only one-bit error correction it can be shown that the number of bits to add to m-bit messages is a little more than $\log_2 m$. For example, 64-bit messages need add only 7 bits. Then $G = (I \mid A)$ has 64 rows and 71 columns where I has 64 rows and columns and A has 64 rows and 7 columns. Then $H = (A^{\mathrm{tr}} \mid I)$ is 7-by-71 with a 7-row by 64-column A^{tr} and 7 by 7 identity I. [Note that the two identity matrices are rarely the same size.]

If we need only one-bit error correction it can be shown that the number of bits to add to m-bit messages is a little more than $\log_2 m$. For example, 64-bit messages need add only 7 bits. Then $G = (I \mid A)$ has 64 rows and 71 columns where I has 64 rows and columns and A has 64 rows and 7 columns. Then $H = (A^{\mathrm{tr}} \mid I)$ is 7-by-71 with a 7-row by 64-column A^{tr} and 7 by 7 identity I. [Note that the two identity matrices are rarely the same size.] The correction step requires that one compare the result Hr^{tr} to the columns of H, which is clearly doable.

If we need only one-bit error correction it can be shown that the number of bits to add to m-bit messages is a little more than $\log_2 m$. For example, 64-bit messages need add only 7 bits. Then $G = (I \mid A)$ has 64 rows and 71 columns where I has 64 rows and columns and A has 64 rows and 7 columns. Then $H = (A^{\mathrm{tr}} \mid I)$ is 7-by-71 with a 7-row by 64-column A^{tr} and 7 by 7 identity I. [Note that the two identity matrices are rarely the same size.] The correction step requires that one compare the result Hr^{tr} to the columns of H, which is clearly doable.

The book talks about "decoding with coset leaders".

If we need only one-bit error correction it can be shown that the number of bits to add to m-bit messages is a little more than $\log_2 m$. For example, 64-bit messages need add only 7 bits. Then $G = (I \mid A)$ has 64 rows and 71 columns where I has 64 rows and columns and A has 64 rows and 7 columns. Then $H = (A^{\mathrm{tr}} \mid I)$ is 7-by-71 with a 7-row by 64-column A^{tr} and 7 by 7 identity I. [Note that the two identity matrices are rarely the same size.] The correction step requires that one compare the result Hr^{tr} to the columns of H, which is clearly doable.

The book talks about "decoding with coset leaders". This just means that instead of comparing to the columns of H, we prepare a table where you can look up Hr^{tr} and find the error position.

If we need only one-bit error correction it can be shown that the number of bits to add to m-bit messages is a little more than $\log_2 m$. For example, 64-bit messages need add only 7 bits. Then $G = (I \mid A)$ has 64 rows and 71 columns where I has 64 rows and columns and A has 64 rows and 7 columns. Then $H = (A^{\mathrm{tr}} \mid I)$ is 7-by-71 with a 7-row by 64-column A^{tr} and 7 by 7 identity I. [Note that the two identity matrices are rarely the same size.] The correction step requires that one compare the result Hr^{tr} to the columns of H, which is clearly doable.

The book talks about "decoding with coset leaders". This just means that instead of comparing to the columns of H, we prepare a table where you can look up Hr^{tr} and find the error position. The reason for the term "coset" is that errors correspond to cosets of the code. If $\mathscr{C}$ is the group code, received words with 1-bit errors are in one of the cosets $e_j + \mathscr{C}$.

If we need only one-bit error correction it can be shown that the number of bits to add to m-bit messages is a little more than $\log_2 m$. For example, 64-bit messages need add only 7 bits. Then $G = (I \mid A)$ has 64 rows and 71 columns where I has 64 rows and columns and A has 64 rows and 7 columns. Then $H = (A^{\mathrm{tr}} \mid I)$ is 7-by-71 with a 7-row by 64-column A^{tr} and 7 by 7 identity I. [Note that the two identity matrices are rarely the same size.] The correction step requires that one compare the result Hr^{tr} to the columns of H, which is clearly doable.

The book talks about "decoding with coset leaders". This just means that instead of comparing to the columns of H, we prepare a table where you can look up Hr^{tr} and find the error position. The reason for the term "coset" is that errors correspond to cosets of the code. If $\mathscr{C}$ is the group code, received words with 1-bit errors are in one of the cosets $e_j + \mathscr{C}$.

Of course, this scheme corrects only one-bit errors, probably not enough for reliable transmission of 64 bits at a time.

If we need only one-bit error correction it can be shown that the number of bits to add to m-bit messages is a little more than $\log_2 m$. For example, 64-bit messages need add only 7 bits. Then $G = (I \mid A)$ has 64 rows and 71 columns where I has 64 rows and columns and A has 64 rows and 7 columns. Then $H = (A^{\mathrm{tr}} \mid I)$ is 7-by-71 with a 7-row by 64-column A^{tr} and 7 by 7 identity I. [Note that the two identity matrices are rarely the same size.] The correction step requires that one compare the result Hr^{tr} to the columns of H, which is clearly doable.

The book talks about "decoding with coset leaders". This just means that instead of comparing to the columns of H, we prepare a table where you can look up Hr^{tr} and find the error position. The reason for the term "coset" is that errors correspond to cosets of the code. If $\mathscr{C}$ is the group code, received words with 1-bit errors are in one of the cosets $e_j + \mathscr{C}$.

Of course, this scheme corrects only one-bit errors, probably not enough for reliable transmission of 64 bits at a time. Setting up one that corrects more errors is beyond the scope of this course.