From the Homework...

- **Problem 4.** Ben Bitdiddle woke up in the middle of the night with the following great idea: in order to implement *double-bit error correction* he would use the (8,4,3) code described in lecture – which can correct single-bit errors – to encode a message twice. In other words, after the message was encoded for the first time with the (8,4,3) code, the resulting bit stream would be reencoded with the same code a second time.
  
a. If the original message had 80 bits, how many bits will be in the doubly-encoded message?

b. Will Ben’s scheme work, i.e., will he be able to correct double-bit errors? Briefly describe why or why not.
From the homework...

<table>
<thead>
<tr>
<th>Digit</th>
<th>Encoding</th>
<th>Printed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 0 0 1 1</td>
<td>n_1</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0 1</td>
<td>n_2</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1 0</td>
<td>n_3</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0 1</td>
<td>n_4</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1 0</td>
<td>n_5</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0 0</td>
<td>n_6</td>
</tr>
<tr>
<td>7</td>
<td>1 0 0 0 1</td>
<td>n_7</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 1 0</td>
<td>n_8</td>
</tr>
<tr>
<td>9</td>
<td>1 0 1 0 0</td>
<td>n_9</td>
</tr>
<tr>
<td>0</td>
<td>1 1 0 0 0</td>
<td>n_0</td>
</tr>
</tbody>
</table>

1. Describe the error detection and correction capabilities of the POSTNet code assuming that we want to safely both detect and correct errors. What’s the number of errors per symbol that can be detected? Corrected?

2. When adding a POSTNet bar code to a piece of mail, the Post Office encodes a 5-digit zip code using 6 digits, where the sixth digit is chosen so that the sum of all six digits is 0 modulo 10. Briefly describe what purpose is served by adding this extra digit.

From the lab...

The (15,8,4) code used in this lab has seven parity bits:

- R1 and R2: the even parity bits for the two 4-bit rows of data bits
- C1, C2, C3 and C4: the even parity bits for four 2-bit columns of data bits
- P: the even parity bit for D1, ..., D8, R1, R2, C1, ..., C4.

When processing a received codeword, we check each of the parity computations and call any mismatch a “parity error”. For each of the following combinations of parity errors assume that 0, 1 or 2 bit errors have occurred. Indicate how many errors have occurred and, if there has been at least one error, whether it is uncorrectable or describe what corrective action is needed to produce the corrected data:

a. no parity errors detected
b. only detect a parity error in the computation of P
c. only detect parity errors in the computation of C1 and C2
d. only detect parity errors in the computation of R1 and C4
e. only detect parity errors in the computation of R1, C4 and P
f. only detect a parity error in the computation of R2
In search of a better code

- Problem: information about a particular message unit (bit, byte, ...) is captured in just a few locations, i.e., the message unit and some number of parity units. So a small but unfortunate set of errors might wipe out all the locations where that info resides, causing us to lose the original message.

- Potential Solution: figure out a way to spread the info in each message unit throughout all the codewords in a block. Require only some fraction good codewords to recover the original message.

Spreading the wealth...

- Idea: oversampled polynomials. Let

\[ P(x) = m_0 + m_1x + m_2x^2 + \ldots + m_{k-1}x^{k-1} \]

where \( m_0, m_1, \ldots, m_{k-1} \) are the \( k \) message units to be encoded. Transmit value of polynomial at \( n \) different predetermined points \( v_0, v_1, \ldots, v_{n-1} \):

\[ P(v_0), P(v_1), P(v_2), \ldots, P(v_{n-1}) \]

Use any \( k \) of the received values to construct a linear system of \( k \) equations which can then be solved for \( k \) unknowns \( m_0, m_1, \ldots, m_{k-1} \). Each transmitted value contains info about all \( m_i \).

- Note that using integer arithmetic, the \( P(v) \) values are numerically greater than the \( m_i \) and so require more bits to represent than the \( m_i \). In general the encoded message would require a lot more bits to send than the original message!
Solving for the $m_i$

- Solving a set of linear equations using Gaussian Elimination (multiplying rows, switching rows, adding multiples of rows to other rows) requires add, subtract, multiply and divide operations.
- These operations (in particular division) are only well defined over fields, eg. rational numbers, real numbers, complex numbers -- not at all convenient to implement in hardware.
- Reed's & Solomon's idea: do all the arithmetic using a finite field (also called a Galois field). If the $m_i$ have $B$ bits, then use a finite field with order $2^B$ so that there will be a field element corresponding to each possible value for $m_i$.
- Note that in a Galois field there are at most $2^B$ unique values $v$ we can use to generate the $P(V)$ -- if we send more than $2^B$ values, some of the equations we might use when solving for the $m_i$ may not be linearly independent and we won't have enough information to find a unique solution for the $m_i$. So Reed-Solomon codes use $n = 2^B-1$ ($n$ is the number of $P(v)$ values we generate and send).

Use for error correction

- If one of the $P(v_i)$ is received incorrectly, if it's used to solve for the $m_i$, we'll get the wrong result.
- So try all possible ($n$ choose $k$) subsets of values and use each subset to solve for $m_i$. Choose solution set that gets the majority of votes.
  - No winner? Uncorrectable error... throw away block.
- If a particular received value is known to be erroneous (an "erasure"), don't use it all: ($n,k$) code can correct $n-k$ erasures since we only need $k$ equations to solve for the $k$ unknowns.
- ($n,k$) code can correct up to $(n-k)/2$ errors since we need enough good values to ensure that the correct solution set gets a majority of the votes.
Example: CD error correction

- **On a CD: two concatenated R-S codes**

  - 32-byte block 32-byte block ... 32-byte block
  - 32-byte block 32-byte block ... 32-byte block
  - Uncorrectable error
  - 28-byte block 28-byte block 28-byte block
  - 28-byte block 28-byte block 28-byte block
  - 24-byte block 24-byte block 24-byte block
  - 24-byte block 24-byte block 24-byte block
  - De-interleave
  - (32,28) code
  - Handles up to 2 byte errors
  - De-interleave
  - (28,24) code
  - Handles up to 4 byte erasures
  - De-interleave

**Result:** correct up to 3500-bit error bursts (2.4mm on CD surface)