Lab #5: Error Detection and Correction

Goal: Using Python, decode a sample bit stream as it might have come from a digital receiver. Provides exposure to the handling of packet framing, burst errors, and single-bit error correction.

Instructions:

1. Complete the pre-lab (see first section below). There are questions to be answered; please write your responses on a separate sheet of paper and turn them in at the beginning of lab on Wednesday.

2. Complete the activities for Wednesday’s lab (see second section below).

3. Prepare the requested material and think about the questions posed on the Checkoff Sheet, then find a staff member to complete your post-lab interview.

Pre-lab (due in lab on Wed, 3/19/08, at 9am or 2pm)

For the next few labs we’ll be using Python, an easy-to-learn interpreted programming language that has a nice selection of data types including support for object-oriented programming, interesting control structures, and a huge (and growing) set of libraries that make it simple to implement almost any processing task.

If you’d like to try out the 6.02 code on your own machine you’ll need to install:

Python -- http://www.python.org/download/
wxPython (ansi version) -- http://www.wxpython.org/download.php

There are pre-built binaries for Windows, Mac OS X and Linux.

If you’d like to learn or brush up on Python, start with the 6.01 Resources webpage

http://courses.csail.mit.edu/6.01/spring08/resource.html

which has links to several quick tutorials. For a more comprehensive introduction, we recommend the Python Tutorial written by Python’s architect Guido von Rossum:
http://docs.python.org/tut/tut.html

The following sections of the Python Tutorial are particularly useful for what we’ll be doing in 6.02:

- Section 3: An Informal Introduction to Python
- Section 4.1: if Statements
- Section 4.2: for Statements
- Section 4.6: Defining Functions
- Section 5.1: More on Lists
- Section 5.5: Dictionaries
- Section 9.3: A First Look at Classes

For the pre-lab, we’ll be reviewing the channel coding process that produced the bit stream you will decode during the lab on Wednesday. You’ll need the information presented here in order to correctly implement the decoding steps needed to recover the transmitted message.

The purpose of the channel coding process is to add redundant information to the transmitted bit stream so that the receiver will be able to detect and correct errors that occurred along the way. As mentioned in lecture, there may be errors that can’t be corrected with even the additional information, a circumstance that will be handled by higher levels of the communication protocol, e.g., by checking for uncorrectable errors using, say, a CRC to confirm the correct receipt of the entire message and requesting a retransmission if the message is corrupted.

Lab5.py is a Python module that defines several functions useful in channel coding tasks:

- `even_parity(binmsg)`
  takes a binary message (a list of 0’s and 1’s) and returns 0 if the list contains an even number of 1’s, otherwise it returns 1.

- `codeword(data,nrows,ncols)`
  `data` is a binary message (a list of 0’s and 1’s) with `nrows*ncols` elements. The first `ncol` elements belong to the first row, the second `ncol` elements to the second row, and so on. This routine returns a new binary message that starts with `data` and then appends (in the following order): even parity bits for each row, starting with the first row; even parity bits for each column, starting with the left column (i.e., the first element in each row); and finally an even parity bit for all the `data` and row/col parity bits. So the length of returned message is `nrows*ncols + nrows + ncols + 1`. The resulting codeword has a Hamming distance of at least 4 from codewords constructed from different `data`.

- `char2bin(c)`
  converts the character `c` into an 8-element binary list of 0’s and 1’s, where the first
element of the list is the least significant bit of the binary representation of \( c \), and so on.

bin2char(binmsg)
convert a binary list 0’s and 1’s (lsb first) into a printable characters. Returns ‘?’ if character is not printable.

printmsg(msg)
prints out a message represented as a list of characters.

transpose(binmsg,ncols)
takes a binary message (a list of 0’s and 1’s) and returns a new binary message with the rows and columns interchanged. \( ncols \) is the number of columns and \( binmsg \) is taken to have multiple rows, each containing \( ncol \) binary bits. The length of \( binmsg \) must be an integer multiple of \( ncols \).

Here the sequence of steps that were followed to prepare the message you’ll decode during lab on Wednesday:

Step 1: split message bit stream into 8-bit blocks and add the necessary row/column/overall parity bits to enable error correction.

For this lab, we want to communicate a text message encoded in ASCII, so we chose an 8-bit block.

We use a (15,8,4) block code that is a simple extension of the (8,4,3) code described in lecture: the eight data bits are organized into 2 rows of 4 columns, where each row and column has an associated parity bit. In order for the generated codewords to have a minimum Hamming distance of 4, we added an overall parity bit, i.e., a bit that ensures that the 15-bit block has even parity. The 8 data bits (D1, …, D4 from row 1 and D5, …, D8 from row 2) are combined with the 7 parity bits (R1, R2, C1, C2, C3, C4, and P) to form the 15-bit codeword. The order of bits in the codeword is shown below:

\[ D1, D2, D3, D4, D5, D6, D7, D8, R1, R2, C1, C2, C3, C4, P \]

Here’s the Python code that does the job:

\[ m = [\text{codeword(char2bin(c),2,4)} \text{ for c in message}] \]

**Question 1:** Describe the error detecting and correcting capabilities of the (15,8,4) code.

**Question 2:** What’s the code rate of a (15,8,4) code?

**Question 3:** 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 and P

Step 2: interleave the bits from B codewords to handle up to B-bit burst errors

First, we start by filling out the message so that its length is an integer multiple of B. To do the interleaving, take groups of B 15-bit codewords and think of them as a matrix with 15 columns (one for each bit of a codeword) and B rows (one for each of the B codewords). We use the transpose function from lab5.py to do the interleaving:

```python
for i in xrange(0,len(m),15*16):
    m[i:i+15*16] = transpose(m[i:i+15*16],15)
```

**Question 4:** Looking at the code above, what's the longest error burst that we'll be able to correct (i.e., what's B)?

Normally one would proceed directly to step 3, but at this point we corrupted each interleaved codeword block with an error burst: a random sequence of between 1 and B bits starting at a random position in the block.

Step 3: insert sync sequence before each bit-stuffed interleaved codeword block

To make it possible for the receiver to find the start of each block of interleaved codewords, we’ll add a sync sequence in front of each block. The sequence we use is defined by the `sync` variable defined in lab5.py: 0111111110. To ensure that the interleaved codeword block doesn’t accidentally contains the sequence of eight 1’s found in the sync sequence, we modify each block with a technique called bit stuffing. The block is examined for sequences of 1 bits of length seven (seven is one less than the sequence of eight 1’s in the sync) – if such a sequence is found, a 0 is inserted into the message at that point, thus guaranteeing that there will be no runs of eight 1’s.
The bit stuffing operation is easily reversed at the receiver: it just has to look for sequences of seven 1’s and then remove the following 0 that was inserted by the transmitter. Note that in the preparation step we stuff the 0 into the sequence whether or not the seven 1’s are followed by an eighth 1 – otherwise the receiver won’t be able to successfully unstuff the received message.

```python
transmit = []  # final binary message
# process each block of interleaved codewords
for i in xrange(0, len(m), 15*16):
    # add sync sequence to transmit message
    transmit.extend(sync)
    # copy block data, bit stuffing as we go
    count = 0
    for bit in m[i:i+15*16]:
        # copy block bit into transmit stream
        transmit.extend(bit)
        if bit == 1:
            # keep track of number of 1’s in a row
            count += 1
            # if we’ve seen seven, insert a 0 at this point
            if count == 7:
                count = 0
                transmit.extend(0)
            else:
                count = 0
```

When the entire message has been converted the accumulated bit stream is written out into a file so that you can have the fun of decoding it. Actually we added some random bits at the front so that one has to search for the sync pattern to find the first interleaved data block.

**Question 5:** We added a lot of bits to the original message. Ignoring the bits added by the bit-stuffing operation, what’s the effective code rate for this example channel coding scenario?

This completes the pre-lab. Please turn in your answers at the start of lab on Wednesday.

---

**In the lab (Wed, 3/19/08, 9am-12n or 2pm – 5pm)**

During this week’s lab your job is write a Python program that processes the channel coded bit stream (available as the value of the `message` variable in `lab5.py`) and recovers the original message. The bit stream has been corrupted by error bursts, so your program will need to implement the appropriate error correction.

To get started, set up the 6.02 environment and copy over the files you’ll need:

```
prompt% setup 6.02
```
Now in the new 6.02 window execute:

```
6.02% cd your_6.02_directory_name_here
6.02% cp -r /mit/6.02/Labs/Lab5 .  (note the period at the end of the command)
6.02% cd Lab5
```

To run the version of Python we use in 6.02, use the following command:

```
6.02% python-602
```

Feel free to develop and run your Python program using whatever development environment you’re used to. You can bring up IDLE (the integrated development environment that is distributed with Python) using the following command:

```
6.02% idle-602 &
```

Your job is to write a Python program that will process the received message and print out the corrected text. The received message is in the form of a binary list (a list of 0’s and 1’s) that’s the value of the `message` variable defined in lab5.py.

Your Python program should import the function and variable definitions from lab5.py, i.e., the first line of your program should be

```
from lab5 import *
```

Step 1: Write and test a program that prints out the message without error correction

Your program needs to reverse the steps described in the prelab:

- Use the `find_sync` function in lab5.py (look in that file to see how it’s called) to split message into a list of bit-stuffed blocks each containing B interleaved codewords.

- For each block of interleaved codewords:
  - Undo the bit-stuffing described in the pre-lab. If you’ve done this correctly, each block should now contain exactly 15*16 bits.
  - De-interleave the block using the `transpose` function defined in lab5.py.
  - For each codeword, i.e., for each 15-element segment of the block, extract the uncorrected data bits and use the `bin2char` function to convert them into an ASCII character, which you can then append to the accumulated message text.

When all the blocks have been processed use `printmsg` to print out the message text you’ve accumulated. It’ll look a bit weird since it’ll have many uncorrected errors.
Call over a staff member to get signed off for this part.

Step 2: Define a Python function `correct(binmsg)` that takes a 15-element binary list of 0’s and 1’s which it should interpret as 8-bit data encoded with the (15,8,4) code described in the pre-lab. Your function should return an 8-element binary list with the correct data OR it should raise a `DecodingError` exception if the codeword contains an uncorrectable error.

To detect and correct errors in the codeword, you’ll need to redo the row, column, and overall parity calculations to determine if a single-bit error has occurred and, if so, make the appropriate corrections to the data bits before returning the result. Your answer to Question 3 in the pre-lab should help you think about how to write your code. Think about the following:

- For what combination of parity errors in the P, Ri and Ci are no modifications necessary to the 8 data bits, i.e., when are there no errors at all, or only errors in the parity bits?

- For what combinations of parity errors has an error been detected in one of the data bits? How can you figure out which data bit got the error? What correction do you have to make to the erroneous data bit?

- All other combinations of parity errors indicate an uncorrectable error and you should raise a `DecodingError`.

Test your function using the `test_decoder` function defined in `lab5.py`:

```python
from lab5 import *

def correct(binmsg):
    # your function definition here

test_decoder(correct)
```

The `test_decoder` function will try a variety of test codewords and check for the correct results. If it finds an error, it’ll tell you which codeword failed; if your code is working, it’ll print out “Tests completed successfully.”

Call over a staff member to get signed off for this part.

Step 3: Modify your program from Step 1 to use your `correct` function to extract corrected data from each codeword before using `bin2char` to convert it into a text character.

When you have recovered the original message, call over a member of the course staff to be checked off.
This completes the lab.

Before finding a staff member for the post-lab interview spend a few minutes thinking about the interview questions listed on the Checkoff Sheet.
Checkoff Sheet for 6.02 Lab #5

Names of team members: _____________________________________________

Checkoff for Step 1 (staff initials): _________________________________

Checkoff for Step 2 (staff initials): _________________________________

Checkoff for Step 3 (staff initials): _________________________________

Post-lab interview questions:

Consider the implementation of error correction that you completed during the lab.

a. If an error burst has corrupted a particular instance of the sync pattern (i.e., it’s no longer recognizable as the sync pattern), briefly describe the effect on the recovered message.

b. In order to ensure the sync pattern wasn’t accidentally found in a block of interleaved codewords, the channel coding algorithm bit-stuffed 0’s after sequences of seven 1’s. If a transmission error changes a 1 in such a sequence to a 0, how will that affect the receiver?

c. Briefly describe the changes that would have to be made to the channel coding and error correction procedures if we wanted to correct errors bursts that were twice as long (bursts of length 2B instead of B).

d. Briefly explain why it would be useful to include a checksum or CRC at the end of the message before passing it through the channel coding process.

Interview signoff (staff initials, score): _______________________________