HW8: Error Correction
CSE1010 Fall 2012
Jeffrey A. Meunier
University of Connecticut

1. Introduction
In this project you will use arrays of numbers to create groups of bits (binary digits) that would be sent over a simulated communication channel from one computer to another. The sender uses parity bits to encode extra information in the data that allows the receiver to detect and correct errors in the transmitted bits. The simulated communications channel will be a function that with low probability flips bits randomly.

2. Value
This program is worth a maximum of 20 points. See the grading rubric at the end for a breakdown of the values of different parts of this project.

3. Due Date
This project is due by 11:59PM on Sunday, Dec 2, 2012.

4. Objectives
In this project you will learn about:
- Using and manipulating strings.
- More vectors and matrices, including 3-dimensional matrices.
- More binary numbers.
- Functions, functions, and more functions.

Keywords: error detection, error correction, 3-d matrices

5. Background
- Read the first section on this web page: Basic Concepts Behind the Binary System.
- Read this link from Wikipedia about what a parity bit is.
- Read this web page for a brief definition of ASCII.
- Read this web page about using multi-dimensional arrays.

5.1. Binary transmission

Modern wired networks and telephone lines often have very low error rates, but wireless networks are subject to much higher error rates from radio interference. Interference can come from a wide range of sources, including other nearby radio networks, airplanes, clouds (clouds with lots of ionization will reflect radio signals), lightning, heavy machinery, sunspot activity, and even cosmic rays emitted by stars that are millions of light years away.

An error in data communication is any bit that is flipped from its original value. In a received data stream, it is impossible just to look at the bits and determine which, if any, are in error, so the sender and receiver agree on a specific data format that allows the use of extra bits in the data stream that are used for error detection or correction.

One error detection scheme calls for placing one extra bit in the data stream at regular intervals. Say you have groups of eight bits to be sent over a network. A ninth bit is added after each 8 bits, and its value is set so that it makes the total number of ones in the nine bits either even if even parity is being used or odd if odd parity is being used.

```
0 1 1 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0 0 1 1 1 0 0 1 0 1 0 1 ...
      data      parity      data      parity      data      parity
```

When a receiver receives the group of nine bits, the parity bit of the original eight bits is recalculated, and if it matches the received parity bit, then it is assumed that the transmission contains no errors (even though it is possible that it contains an even number of errors, or the parity bit itself is in error). If the parity bit does not match, then the transmission contains an error and the receiver asks the sender to re-send the data. More advanced parity schemes allow the receiver to detect any small number of errors, and in some cases to correct one or more errors.

In this assignment you will employ parity generation and analysis that could be used to detect ***and correct*** errors in bit streams. The bit stream will in fact just be a two-dimensional vector of ones and zeroes (rather, a 2-D matrix sent as a 1-D vector, then re-constructed into a 2-D matrix). Applying parity to each row and column of this array will enable you to write a function that can actually *correct* a certain small number of errors.

Bytes and Octets

Because all values in a computer are stored internally using binary values (ones and zeros), it is only natural that a computer would transmit data to other computers using this format. When a byte leaves a computer and is sent out into some transmission medium

(like a network), the byte is usually called an octet, which means 'group of eight'. The reason for this is that in some computers (usually old ones), a byte can be more or fewer than 8 bits, but in a network, we need to know exactly how many bits we're dealing with, so a group of 8 bits is called an octet. If there are more or fewer than 8 bits in a grouping, it's not called an octet.

Parity

When an octet is sent from one computer to another, the sender appends additional information to the byte in order to allow the receiver to detect if a transmission error has occurred. A single bit called a parity bit is often appended to each octet. This bit is used to add redundant information to each octet, making the total number of bits in each 9-bit group either even or odd. The sender and receiver must agree ahead of time to use either even or odd parity. If the receiver receives an odd number of ones in a 9-bit group, but the sender and receiver are using even parity, then the receiver knows that one of the bits was flipped from 0 to 1 or from 1 to 0 during transmission.

For example, say that the sender wishes to send the octet 11010011 over the network. The sender and receiver have agreed to use even parity, which means that an additional bit will be appended to the octet that makes the number of bits even. The octet has 5 ones, so a 1 is appended to the octet in the 9^{th} bit position, making it 110100111. This entire group (it can no longer be called an octet) now has an even number of bits. Note that if the original octet already had an even number of bits, then the sender would have appended a 0, keeping the number of bits even. Finally, after appending the parity bit to the octet, the 9-bit group is sent to the receiver.

Now assume that the receiver receives this nine bit group: 110000111 (notice that it's different from the one that the sender sent). The receiver calculates the parity (*even* parity, in this case) for the first 8 bits, which is 0 (since 4 of the 8 bits are 1). However, the sender sent a 1 for the parity bit (the 9^{th} bit). The received and calculated parity bits disagree, which tells the receiver that this octet contains one flipped bit. Normally when an error is detected the receiver would ask the sender to re-send the group of bits.

Now let's assume that we have the following 8 octets to send:

```
0 1 0 1 0 1 0 1
1 1 1 0 1 1 1 0
1 1 0 0 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 0 1 0 1 0
```

```
1 0 0 0 0 1 1 0
1 0 1 1 0 0 0 0
0 0 1 1 1 1 0 1
```

What we would normally do is compute a parity bit for each octet, and send each 9-bit group in succession. However, by adding more parity information we can allow the receiver to both detect and correct a single bit error. We can do this by calculating parity bits not only across the rows of bits, but also down the columns of bits. This generates one parity bit for each row, plus an extra row of parity bits, one for each column. Here the even parity bits are calculated for the rows and columns. The row and column numbers are shown in light blue, and the parity bits are in column 9 and row 9.

Add a 9th column of parity bits, one bit for each row:

	1	2	3	4	5	6	7	8	9
1	0	1	0	1	0	1	0	1	0
2	1	1	1	0	1	1	1	0	0
3	1	1	0	0	0	1	0	1	0
4	1	0	1	0	1	0	1	0	0
5	0	1	0	0	1	0	1	0	1
6	1	0	0	0	0	1	1	0	1
7	1	0	1	1	0	0	0	0	1
8	0	0	1	1	1	1	0	1	1

Add a 9th row of parity bits, one bit for each column including the 9th:

	1	2	3	4	5	6	7	8	9
1	0	1	0	1	0	1	0	1	0
2	1	1	1	0	1	1	1	0	0
3	1	1	0	0	0	1	0	1	0
4	1	0	1	0	1	0	1	0	0
5	0	1	0	0	1	0	1	0	1
6	1	0	0	0	0	1	1	0	1
7	1	0	1	1	0	0	0	0	1
8	0	0	1	1	1	1	0	1	1
9	1	0	0	1	0	1	0	1	0

Note that here we are using even parity, and each row of 9 bits now contains an even number of bits, and each column of 9 bits contains an even number of bits. It turns out that it is a coincidence that the 9th row contains an even number of bits: it may not be this way for every example.

After transmission, the receiver receives the entire 81-bit group and places it into nine rows and nine columns. However, during transmission there was some noise interference

and one of the bits was flipped from 1 to 0, shown highlighted in the middle:

```
0 1 0 1 0 1 0 1 0
1 1 1 0 1 1 1 0 0
1 1 0 0 0 1 0 1 0
1 0 1 0 1 0 1 0 0
0 1 0 0 0 0 1 0 1
1 0 0 0 0 1 1 0 1
1 0 1 1 0 0 0 0 1
0 0 1 1 1 1 0 1 1
1 0 0 1 0 1 0 1 0
```

The receiver does not yet know that there was a transmission error. In order to determine this, the receiver calculates its own parity bits for each row and column of data bits (not including the parity bits that the sender sent) and compares them to the parity bits that the sender sent. If the parity columns differ in any location or the parity rows differ in any location, then that row and column indicate where the flipped bit is.

Below, the upper left 8x8 square of bits is called the payload. The received parity bits are shown in bold, making up the whole 9x9 frame. The calculated parity bits are shown in blue separated from the frame by dashed lines. The receiver determines that one of the rows and one of the columns contains an error. The intersection of this row and column indicates where the error is:

```
0 1 0 1 0 1 0 1   0 0
1 1 1 0 1 1 1 0   0 0
1 1 0 0 0 1 0 1   0 0
1 0 1 0 1 0 1 0   0 0
0 1 0 0 0 0 1 0   1 0  ← parity mismatch in this row
1 0 0 0 0 1 1 0   1 1
1 0 1 1 0 0 0 0   1 1
0 0 1 1 1 1 0 1   1 1

1 0 0 1 0 1 0 1   0
1 0 0 1 1 1 0 1   0
        ↑
parity mismatch in this column
```

The receiver knows that the 0 is incorrect, so the correct value must be 1, and it simply flips the bit to correct it, and then discards all the parity information, leaving the original 8x8 matrix of bits.

```
0 1 0 1 0 1 0 1
```

```
1 1 1 0 1 1 1 0
1 1 0 0 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 0 1 0 1 0
1 0 0 0 0 1 1 0
1 0 1 1 0 0 0 0
0 0 1 1 1 1 0 1
```

6. Assignment

Read through all of the subsections in this section before you start to make sure you understand how you must proceed with the assignment.

The program consists of three phases, neatly divided into three parts:

1. Encoding: a string is encoded as bits with parity
2. Transmission: noise is added to the signal to simulate transmission over a noisy channel
3. Decoding: the bits with parity are decoded back into a string

6.1. Encoding Phase

In your CSE1010 folder in Matlab create a new folder for HW8.

Section 6.1 is the encoding phase. It extends what you did in the previous assignment.

6.1.1. Function string2bin

This function takes a string and converts it into a matrix of bits. Each row of the matrix contains the bits that encode the ASCII value for a single character. Use the **char2bin** function that you already wrote. This ensures that all rows have 8 columns.

This function iterates over the string, converting each character into a row vector of bits, and appends each row vector to the bottom of the matrix of bits.

Testing

```
>> string2bin('a')
ans =
  0  1  1  0  0  0  0  1 (binary representation of 'a')
>> string2bin('uvwxyz')
ans =
  0  1  1  1  0  1  0  1 (binary representation of 'u')
  0  1  1  1  0  1  1  0 ('v')
  0  1  1  1  0  1  1  1 ('w')
  0  1  1  1  1  0  0  0 ('x')
  0  1  1  1  1  0  0  1 ('y')
```

```
0  1  1  1  1  0  1  0 ('z')
```

6.1.2. Function segmentString

This function takes any string and returns an *n* x 8 array of characters. Here's an example:

```
>> segmentString('Hello, world!')
ans =
Hello, w
orld!
```

What you don't see is that there are 3 invisible null characters (character number 0) at the end of the second row, making the second row 8 characters long (`o r l d ! ␣ ␣ ␣`).

The simplest way to add those null characters is to store a null character in the last location of the string that makes the string a multiple of 8 characters in length, and Matlab will fill in all the locations in between with null characters. For example, if the string is 'Hello', which is 5 characters long, then storing a 0 in location 8 will fill locations 6, 7, and 8 with 0s. See how it works:

```
>> s='Hello';
>> double(s)
ans =
  72  101  108  108  111
>> s(8)=0% char(0) will also work; both are correct
ans = Hello
>> double(s)
ans =
  72  101  108  108  111  0  0  0
```

If the string is from 9 to 15 characters long, then store the 0 in location 16. If it's from 17 to 23 characters long, store the 0 in location 24. And so on. This does not mean you need to use a series of if statements. I'm simply describing how the function behaves. The function should work no matter how long the string is.

Make sure you don't overwrite the 8th character (or 16th or 24th...) if the length of the string is already a multiple of 8.

Make sure you don't append an additional row of all 0 characters.

After you make the string the correct length, you can use the reshape function to chop it into segments. If you type **help reshape** in Matlab, the first paragraph of the description is what you need to do.

You do not need to use a *for* loop in this function, but use one if you want to. You do need

to use some math.

6.1.3. Function string2blocks

This function takes a string of any length and cuts it into 8-character segments using the **segmentString** function, then calls the **string2bin** function on each segment. The matrix of bits returned by the string2bin function is appended to the end of a 3-dimensional matrix of bits. The 3-d matrix is returned by this function.

Testing

```
>> string2blocks('Hello')
ans =
  0 1 0 0 1 0 0 0  (H)
  0 1 1 0 0 1 0 1  (e)
  0 1 1 0 1 1 0 0  (l)
  0 1 1 0 1 1 0 0  (l)
  0 1 1 0 1 1 1 1  (o)
  0 0 0 0 0 0 0 0  (0)
  0 0 0 0 0 0 0 0  (0)
  0 0 0 0 0 0 0 0  (0)
```

The string 'Hello' fits into a single frame, but notice that the last three characters are null characters: all bits are 0.

```
>> string2blocks('Hello, World!')
ans(:,:,1) =
  0  1  0  0  1  0  0  0
  0  1  1  0  0  1  0  1
  0  1  1  0  1  1  0  0
  0  1  1  0  1  1  0  0
  0  1  1  0  1  1  1  1
  0  0  1  0  1  1  0  0
  0  0  1  0  0  0  0  0
  0  1  0  1  0  1  1  1
ans(:,:,2) =
  0  1  1  0  1  1  1  1
  0  1  1  1  0  0  1  0
  0  1  1  0  1  1  0  0
  0  1  1  0  0  1  0  0
  0  0  1  0  0  0  0  1
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
```

The string 'Hello, World!' needs two frames.

6.1.4. Modify the appendParity function

Modify the **appendParity** function so that it places the parity bit in location 9 of the bit vector, regardless of how many bits the vector has. Thus:

```
>> appendParity([1 0 1], 1)
ans =
  1  0  1  0  0  0  0  0  1
>> size(ans)
ans =
  1  9
```

6.1.5. Function appendParityColumn

This function accepts any number of rows of bits, and a desired parity (in other words it has 2 parameters). This means you will need to append the parity bit to each row of the matrix passed as an argument to this function. Use the **appendParity** function to append a parity bit to a single row.

Testing

```
>> B=[1 0 1; 1 0 0; 0 1 1]
B =
  1  0  1
  1  0  0
  0  1  1
>> appendParityColumn(B, 1)
ans =
  1  0  1  0  0  0  0  0  1
  1  0  0  0  0  0  0  0  0
  0  1  1  0  0  0  0  0  1
```

Just for fun, at the command prompt call the **appendParityColumn** function on the result of the **string2bin** function:

```
>> appendParityColumn(string2bin('abc'),0)
ans =
  0  1  1  0  0  0  0  1  1
  0  1  1  0  0  0  1  0  1
  0  1  1  0  0  0  1  1  0
>> size(ans)
ans =
  3  9
```

6.1.6. Function appendParityRow

This function takes a matrix of bits and a desired parity, and adds a 9th row of bits that are the parity bits of the columns of the matrix.

It turns out that you can (and should) call the **appendParityColumn** function inside this function. You need to append parity bits into the 9th *row*, but **appendParityColumn** places parity bits in the 9th *column*. What you must do is transpose the matrix of bits (turning rows into columns), append parity bits to the rows (they will appear in the 9th column), then transpose the matrix again to make the 9th column the 9th row.

Testing

```
>> appendParityColumn(string2bin('abc'),0)
ans =
  0  1  1  0  0  0  0  1  1
  0  1  1  0  0  0  1  0  1
  0  1  1  0  0  0  1  1  0
>> appendParityRow(ans,0)
ans =
  0  1  1  0  0  0  0  1  1
  0  1  1  0  0  0  1  0  1
  0  1  1  0  0  0  1  1  0
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  0  1  1  0  0  0  0  0  0
>> size(ans)
ans =
  9  9
```

6.1.7. Function appendParityToBlock

This function takes a block of bits and a desired parity, and returns a 9 x 9 matrix that has a column and a row of parity bits appended to it. Inside this function call the **appendParityColumn** function first, then call the **appendParityRow** function on that result.

Testing

```
>> appendParityToBlock([1 0 1; 1 0 0; 0 1 1], 1)
ans =
  1  0  1  0  0  0  0  0  1
  1  0  0  0  0  0  0  0  0
  0  1  1  0  0  0  0  0  1
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  1  0  1  1  1  1  1  1  1
```

After calling this function, all rows (except occasionally the bottom row) will have the correct number of bits (even or odd), and all columns will have the correct number of bits (even or odd).

6.1.8. Function appendParityToBlocks

This function takes a 3-d matrix of bits and a desired parity, and it appends a parity row and column to each 2-d block in the matrix. The new 3-d matrix is returned. Each 2-d matrix (each 9x9 block) in this 3-d matrix is called a *frame*.

Testing

```
>> appendParityToBlocks(string2blocks('Hello, World!'), 0)
ans(:,:,1) =
  0  1  0  0  1  0  0  0  0
  0  1  1  0  0  1  0  1  0
  0  1  1  0  1  1  0  0  0
  0  1  1  0  1  1  0  0  0
  0  1  1  0  1  1  1  1  0
  0  0  1  0  1  1  0  0  1
  0  0  1  0  0  0  0  0  1
  0  1  0  1  0  1  1  1  1
  0  0  0  1  1  0  0  1  1
ans(:,:,2) =
  0  1  1  0  1  1  1  1  0
  0  1  1  1  0  0  1  0  0
  0  1  1  0  1  1  0  0  0
  0  1  1  0  0  1  0  0  1
  0  0  1  0  0  0  0  1  0
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  0  0  1  1  0  1  0  0  1
>> size(ans)
ans =
  9  9  2
```

6.1.9. Summary

You have now written functions that encode any string into blocks of bits that have two-dimensional parity bits appended to them. The parity bits will be used during the decoding phase to determine if any bits were erroneously flipped during transmission.

6.2. Transmission Phase

This is where errors will be introduced into the blocks of bits, simulating the transmission of the bits over a noisy communications channel. Examples of communications channels are: Ethernet network cables, Wi-fi wireless networks, phone lines, fiber optic cables, cable TV cables, even storage media like hard disks and CD or DVD ROM disks can have errors. In fact, no communications channel is perfectly error-free, it's just that some are much more reliable than others.

6.2.1. Function transmitFrames

This function takes a 3-d matrix of bits (the *frames*) and an error probability. Call the **addNoise** function on the frames, giving it the entire 3-d matrix of frames and the error probability. Display the number of bits that were flipped, and return only the new frames. Use the fprintf function to display the message so it appears on one line.

Testing

```
>> frames = appendParityToBlocks(string2blocks('Hello, World!'), 0);
>> transmitFrames(frames, 0.01)
number of bits flipped: 1
ans(:,:,1) =
  0  1  0  0  1  0  0  0  0
  0  1  1  0  0  1  0  1  0
  0  1  1  0  1  1  0  0  0
  0  1  1  0  1  1  0  0  0
  0  1  1  0  1  1  1  1  0
  0  0  1  0  1  1  0  0  1
  0  0  1  0  0  0  0  0  1
  0  1  0  1  0  1  1  1  1
  0  0  0  1  1  0  0  1  1
ans(:,:,2) =
  0  1  1  0  1  0  1  1  0
  0  1  1  1  0  0  1  0  0
  0  1  1  0  1  1  0  0  0
  0  1  1  0  0  1  0  0  1
  0  0  1  0  0  0  0  1  0
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  0  0  1  1  0  1  0  0  1
```

6.3. Decoing Phase

Make sure you read and understand the material in the Introduction section, otherwise this may not make much sense.

6.3.1. Function checkParityOfFrame

This function takes a frame (9x9 matrix of bits) and a desired parity. Cut the frame into 3 pieces: the payload (the upper left 8x8 matrix of data bits), the parity column (but only rows 1 - 8), and the parity row (all 9 columns). I will refer to these as the *received parity column* and *received parity row*.

This function must re-calculate the parity of the payload (*only the payload*) by using the **appendParityToBlock** function (remember, now we are on the receiving end of the transmission and the receiver does not yet know if there are any errors). Now take the first 8 bits from the parity column and the 9 bits of the parity row. I will call these the

calculated parity column and *calculated parity row*.

Compare the received parity column to the calculated parity column. The rows where the columns do not match (are not equal) indicate the row or rows in the payload where there is an error. Also compare the received parity row to the calculated parity row. The columns where the rows do not match indicate the column or columns in the payload where there is an error.

In this function you can use the **checkParity** function from homework project 5, but you don't have to.

Return two values from this function: (1) the rows where there are errors, and (2) the columns where there are errors. That means you need to use two return variables.

Testing

```
>> blocks = string2blocks('Hello, World!');
>> txFrames = appendParityToBlocks(blocks, 0);
>> rxFrames = transmitFrames(txFrames, 0.01);
number of bits flipped: 2
```

You may see a different number of bits flipped.

```
>> frame = rxFrames(:, :, 1) % get the first frame
>> [r c] = checkParityOfFrame(frame, 0)
r =
  5
c =
  4  9
```

You will certainly see different values for r and c, but if there is only one bit error in that frame, then c will probably be the vector `[__ 9]`, i.e., *something* and 9. The reason you often see an error in column 9 is because there is an error in one of the rows, and since that parity bit is stored in column 9, the column parity for column 9 is in error.

6.3.2. Function repairFrame

This function takes a frame (9x9), a vector of error rows, and a vector of error columns, and tries to repair the frame. The error rows and error columns are the values returned by the **checkParityOfFrame** function above.

This function returns two things: the repaired frame, and the repair status. The repair status is a number: 1 means that the frame contains no errors, 2 means that the frame was corrected, and 0 means that the frame could not be corrected.

If there is a single bit error in the payload then there will be a single error row indicated, and two error columns indicated: the actual error column, and column 9 (because the parity column will now be incorrect). If there are two payload errors, then it is not possible to correct the error. Draw a matrix of bits (4x4 is sufficient), add a parity column and parity row, and mark two error rows and two error columns. Understand why you can't determine exactly where the errors are.

The function must work like this:

- If either there are no error rows or no error columns, then there are no errors in the payload portion of the frame (the 8x8 section of data bits), so simply return the frame as it is and the repair status of 1.
    - Consider: If there no error rows but there is an error column (or vice versa), then the error is in the parity bits themselves and they can safely be ignored. This case already handles this because I wrote *either/or* above.
- If there is exactly 1 error row *and* exactly 2 error columns *and* one of those columns is column 9, then the error row number and column number indicate which bit in the payload is in error. Flip that bit from a 0 to a 1, or from a 1 to a 0. Return the frame and the repair status of 2.
- Otherwise return the frame as-is, and a repair status of 0.

Testing

It may be useful to enter all the previous testing statements into a script file to make it easier to re-run each test.

```
>> [repairedFrame status] = repairFrame(frame, r, c)
repairedFrame =
  0  1  0  0  1  0  0  0  0
  0  1  1  0  0  1  0  1  0
  0  1  1  0  1  1  0  0  0
  0  1  1  0  1  1  0  0  0
  0  1  1  0  1  1  1  1  0
  0  0  1  0  1  1  0  0  1
  0  0  1  0  0  0  0  0  1
  0  1  0  1  0  1  1  1  1
  0  0  0  1  1  0  0  1  1
status =
  2
```

6.3.3. Function repairFrames

This function takes a 3-d matrix of 9x9 frames and a desired parity. It returns two values: the repaired frames, and a vector of numbers indicating which frames could not be repaired.

To do this, iterate over all the frames one at a time. For each frame, call the **checkParityOfFrame** function on it and save the return values. Then take those return

values and use them to call the **repairFrame** function on the same frame, and save those return values.

Check the repair status:

- If it is 0, display '**Frame** *frameNumber* **could not be repaired**'.
- If it is 1, display '**Frame** *frameNumber* **is correct**'.
- If it is 2, display '**Frame** *frameNumber* **has been repaired**'.

In each case, *frameNumber* is the number of the frame in the 3-d matrix (it's the index of the 3rd dimension).

Testing

Using the same 3-d matrix **rxFrames** from the previous example, if the number of bits flipped is low enough (not more than 1 error per frame), then you should see something like this:

```
>> [repairedFrames errorFrames] = repairFrames(rxFrames, 0);
Frame 1 has been repaired
Frame 2 has been repaired
>> repairedFrames
repairedFrames(:,:,1) =
  0  1  0  0  1  0  0  0  0
  0  1  1  0  0  1  0  1  0
  0  1  1  0  1  1  0  0  0
  0  1  1  0  1  1  0  0  0
  0  1  1  0  1  1  1  1  0
  0  0  1  0  1  1  0  0  1
  0  0  1  0  0  0  0  0  1
  0  1  0  1  0  1  1  1  1
  0  0  0  1  1  0  0  1  1
repairedFrames(:,:,2) =
  0  1  1  0  1  1  1  1  0
  0  1  1  1  0  0  1  0  0
  0  1  1  0  1  1  0  0  0
  0  1  1  0  0  1  0  0  1
  0  0  1  0  0  0  0  1  0
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0
  0  0  1  1  0  1  0  0  1
```

This is an ideal example where there were just two errors, one in each frame, and each error could easily be repaired.

If any frame could not be repaired, it does not mean your program is wrong, it just means that there are too many errors in the transmission stream. Check the value of the **errorFrames** vector. The frame numbers that could not be corrected should be in that variable. Try the example over again with a shorter string (like just a single letter instead of

'Hello, World!'). Run the test several times until you get (1) no errors, (2) a single error that is repaired, and (3) multiple errors that are not repaired. Verify that everything works correctly.

6.3.4. Function stripFrames

This function takes a 3-d matrix of 9x9 frames and removes the parity row and column, returning a 3-d matrix of 8x8 payload bits.

Iterate over each frame in the matrix of frames. Copy only the upper left 8x8 bits of the frame into a new matrix. Return that 3-d matrix.

Testing

```
>> blocks = stripFrames(repairedFrames)
blocks(:,:,1) =
  0  1  0  0  1  0  0  0
  0  1  1  0  0  1  0  1
  0  1  1  0  1  1  0  0
  0  1  1  0  1  1  0  0
  0  1  1  0  1  1  1  1
  0  0  1  0  1  1  0  0
  0  0  1  0  0  0  0  0
  0  1  0  1  0  1  1  1
blocks(:,:,2) =
  0  1  1  0  1  1  1  1
  0  1  1  1  0  0  1  0
  0  1  1  0  1  1  0  0
  0  1  1  0  0  1  0  0
  0  0  1  0  0  0  0  1
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
```

6.3.5. Function bin2string

This function does the opposite of the **string2bin** function. It takes a 2-d matrix of bits having 8 columns and any number of rows. It iterates over each row of the matrix, converts each row into a character (use your **bin2char** function), and appends the character to the end of the string. The string is returned.

A special case is if the character is equal to 0. When you find this character, you may end processing the bits immediately, since all subsequent characters will also be 0.

Testing

```
>> string2bin('wxyz')
ans =
  0  1  1  1  0  1  1  1
```

```
  0  1  1  1  1  0  0  0
  0  1  1  1  1  0  0  1
  0  1  1  1  1  0  1  0
>> [ans ; zeros(2,8)]
ans =
  0  1  1  1  0  1  1  1
  0  1  1  1  1  0  0  0
  0  1  1  1  1  0  0  1
  0  1  1  1  1  0  1  0
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
>> bin2string(ans)
ans =
wxyz
>> size(ans)
ans =
 1  4
```

6.3.6. Function blocks2string

This function takes a 3-d matrix of 8x8 blocks of bits. Every 8x8 block is a segment of 8 characters. Each segment must be converted from its binary form into a string of 8 characters.

Iterate over the blocks in the 3-d array, and build a string 8 characters at a time. For each block, call the **bin2string** function on that block and store the returned characters to the end of the string you are building. After the first block the string will have 8 characters. After the second block the string will have 16 characters, then 24, and so on until there are no more blocks.

Testing

```
>> blocks2string(blocks)
ans =
Hello, World!
```

6.4. Script p8

This script ties it all together. Do this:

1. Define two variables: **ERROR_PROB** to be 0.01, and **DESIRED_PARITY** to be 0.
2. Re-seed the random number generator.
3. Get a string from the user.
4. Convert that string to blocks.
5. Append parity to those blocks. Use **DESIRED_PARITY**. I call the result the *frames* (or *transmitted frames*).
6. Transmit the frames, use **ERROR_PROB**, and store the result in a variable. I call the result the *received frames*.
7. Repair the received frames using **DESIRED_PARITY**. I call the result the *repaired*

frames.

8. Strip the repaired frames. I call the result the *received blocks*.
9. Convert the received blocks to a string.
10. Display the string.
11. Display the vector containing the list of un-repairable frames.

Testing

```
>> p8
enter a string: Hello, World!
number of bits flipped: 0
frame 1 is correct
frame 2 is correct
received string = "Hello, World!"
frames containing errors:
>> p8
enter a string: Hello, World!
number of bits flipped: 1
frame 1 has been repaired
frame 2 is correct
received string = "Hello, World!"
frames containing errors:
>> p8
enter a string: Hello, World!
number of bits flipped: 4
frame 1 is correct
frame 2 could not be repaired
received string = "Hello, Wmrlda"
frames containing errors:
  2
```

Note that this means that frame 2 contains errors, not that there are 2 frames with errors.

6.5. Program comment lines

Place the normal comment block at the beginning of your **p8.m** file, and two internal comment lines in each function. The first comment line should show a brief description of the function, and the second line should show how to call the function.

7. Report

Create a Microsoft Word or OpenOffice Word file (or some other format that your TA agrees on – ask him or her if you are not sure). Save the file with a .doc or .docx format.

At the beginning of the document include this information:

Error Correction
CSE1010 Project ___, Fall 2012

Your name goes here
The current date goes here
TA: *Your TA's name goes here*
Section: *Your section number goes here*
Instructor: Jeffrey A. Meunier

Be sure to replace the parts that are underlined above.

Now create the following five sections in your document.

1. Introduction

In this section copy & paste the text from the introduction section of this assignment. (It's not plagiarism if you have permission to copy something. I give you permission.)

2. Inputs & outputs

Describe what kind of values the user is expected to supply, and what kind of values the program displays. Don't give examples, just describe the information: pretend you're telling your mom or dad or S.O. about it over the phone. "The user must enter a string, and the program displays the binary values of the characters in the string. Then the program displays the string after simulating transmission of that string over a noisy channel." Etc. Also describe any constants used in the program (this is the error probability and the parity used, even or odd).

3. Sample run

Copy & paste two sample runs. They will both probably have one or more transmission errors.

4. Source code

Copy & paste the contents of your .m files here. You do not need to write anything.

8. Submission

Submit the following things things on HuskyCT:

1. All the .m files for this project.
2. The MS Word document.

If for some reason you are not able to submit your files on HuskyCT, email your TA before the deadline. Attach your files to the email.

9. Notes

Here are some notes about working on this project:

- I can't think of anything right now.

10. Grading Rubric

Your TA will grade your assignment based on these criteria:

- (14 points) The program works correctly
- (3 points) The program is formatted neatly and correctly, and the comments are as they should be: in each function and in the script file.
- (3 points) The document contains all the correct information and is formatted neatly.

11. Getting help

Start your project early, because you will probably not be able to get help in the last few hours before the project is due.

- If you need help, send e-mail to your TA immediately.
- Go to the office hours for (preferably) your TA or any other TA. I suggest you seeing your own TA first because your TA will know you better. However, don't let that stop you from seeing any TA for help.
- Send e-mail to Jeff.
- Go to Jeff's office hours.