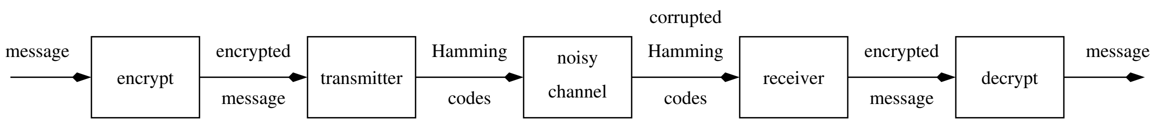
Honors EA1, Homework #7: *Encryption and error correction*

In this assignment, you will implement as MATLAB functions both the encryption and the error correction schemes presented in class. The combination of these schemes can be pictured as follows:



Begin by writing a MATLAB function called `crypt` which takes as inputs a message string and a square key matrix and produces as output an encrypted message string according to the method outlined in the encryption case study. Your function should work with any size message and any size key; if the length of the message is not evenly divisible by the size of the key, then pad the message with trailing spaces (ASCII code 32) before using the `reshape` function. The encrypted string produced by your function should only contain printable characters (ASCII range 32–126). Your function should work for both encryption and decryption: to decrypt an encrypted message, simply call your function with the encrypted message and the *inverse* of the key (rounded) as inputs. Your function need not verify that the given key is valid (i.e., that it has an integer-values inverse). The following MATLAB session illustrates how your function should work:

```
>> K = [ 1    -3    -4     2
         2    -5    -6     6
        -3    10    15     1
         1    -1     3    22 ];
>> s = 'Let''s try to encrypt this sentence using our key.';
>> c = crypt(s,K)
```

(double(msg)-32)*key

```
c =

uI4!&[PQ&U;tz`;8K<<vq]}<or4Mvk)1>.$_jL,!^^YpY>e .<U.

>> crypt(c,round(inv(K)))

ans =

Let's try to encrypt this sentence using our key.
```

Note that to enter an apostrophe as part of a string, you must include two apostrophes in a row so that MATLAB knows you are not yet finished with the string. Use the above key to test your function on various message strings, making sure it encrypts and decrypts correctly.

Your next task is to write MATLAB functions which implement the process of transmitting and receiving binary data over a noisy communication channel using Hamming codes. To begin, write a function called `int2bin` according to the following specifications:

ASCII rep—>ASCII values N—>Binary B—>Add Hamming Code W

- The first input argument `N` is mandatory and should be a row vector of nonnegative integers to be converted to binary form. The second input argument `b` is optional and represents the minimum number of bits to use when representing values in `N`.
- The output argument `B` should be an array of 1's and 0's with as many columns as there are entries in `N` and with at least `b` rows. The actual number of rows of `B` should be the larger of `b` (if specified) and the minimum number of bits needed to represent the maximum value in `N`, namely, `floor(log2(max([N 1]))) + 1`.
- Each column of `B` should be the binary representation of the corresponding column of `N`, with powers of 2 increasing as one proceeds down the column. Thus your function should operate as follows:

```
>> int2bin(1:7)  % Hamming matrix for (7,4) code

ans =

     1     0     1     0     1     0     1
     0     1     1     0     0     1     1
     0     0     0     1     1     1     1

>> int2bin(1:7, 5)  % Same numbers, but use at least 5 bits

ans =

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

For those of you who do not know how to convert a nonnegative integer into binary form, you may use the following algorithm. Suppose that `N` is the number `75` and that `b` is `8`; then `B` will be a column vector with eight entries. Start with the eighth entry of `B`: is `N` at least $2^{8-1} = 128$? No, so set `B(8)=0`. Next, is `N` at least $2^{7-1} = 64$? Yes, so set `B(7)=1` and subtract `64` from `N` so that `N` is now `11`. Next, is `N` at least $2^{6-1} = 32$? No, so set `B(6)=0`. Next, is `N` at least $2^{5-1} = 16$? No, so set `B(5)=0`. Next, is `N` at least $2^{4-1} = 8$? Yes, so set `B(4)=1` and subtract `8` from `N` so that `N` is now `3`. Next, is `N` at least $2^{3-1} = 4$? No, so set `B(3)=0`. Next, is `N` at least $2^{2-1} = 2$? Yes, so set `B(2)=1` and subtract `2` from `N` so that `N` is now `1`. Finally, set `B(1)=N`. Thus we have the following:

```
>> int2bin(75,8)

ans =
1
1
0
1
0
0
1
0
```

Once your function `int2bin` is working, write a function `bin2int` which does the exact opposite operation, namely, converts a binary array into a row vector of nonnegative integers. This function should have one input argument `B` (the binary array) and one output argument `N` (the corresponding row vector of integers). As above, the columns of `B` should be interpreted as binary numbers with powers of 2 increasing as one proceeds down the column. One way to test that the function is working is to make sure that when `N` is a row vector of nonnegative integers, the expression `bin2int(int2bin(N,b))` returns exactly `N` for any nonnegative (scalar) integer `b`. [Hint: the conversion in `bin2int` is easier than the conversion in `int2bin`; in fact, the body of `bin2int` can contain a single line of code (not including comments and error checking).]

The next function you need to write, called `int2hamm`, will use the function `int2bin` you wrote above to convert a row vector `N` of nonnegative integers into a corresponding array of binary Hamming code words according to the following specifications: Put 0s in position of parity bits of message

p=mod(H*W,2)

- The first input argument `N` is mandatory and should be a row vector of nonnegative integers to be converted into binary code words. The second input argument `L` is mandatory and should be an integer representing the total number of bits in each code word (data bits plus parity bits). The third input argument `d` is mandatory and should be the number of data bits in each code word. The resulting Hamming code will thus be an `(L,d)` code. The fourth and last input argument `b` is optional and represents the minimum number of bits to use when representing values in `N` (should default to 1).

- There should be two output arguments, `w` and `bits`. The binary array `w` of Hamming code words should be constructed as described below, and `bits` should be a positive integer whose value is the actual number of bits used to represent the values in `N` (note that `bits` will be at least as big as the input argument `b`).

- As part of the usual checks for bad input data, your function should generate an error if `(L,d)` does not represent a valid Hamming code, that is, if either `L<3`, `d<1`, `L>2^p-1`, or `L<2^(p-1)`, where `p=L-d` represents the number of parity bits in the code word. (In class we only considered the case where `L=2^p-1`, if `L` is smaller than this value but larger than `2^(p-1)`, then a Hamming code is given by simply dropping the final `2^p-1-L` columns from the parity check matrix.)

- The first step in generating the Hamming codes should be to convert `N` to its binary representation `B` using the `int2bin` function you wrote above, namely, `B=int2bin(N,b)`. The output argument `bits` is now simply the number of rows of `B`.

- Each code word should have `d` data bits, but this might not be the same as the number of rows of your array `B`. Thus you should use the `reshape` function to change your data array `B` into an array with `d` rows and `n` columns, where `n` is just large enough to accommodate all entries in your original array `B`, namely, `n=ceil(prod(size(B))/d)`. Note that if `prod(size(B))` is not evenly divisible by `d`, then you will have to pad `B` with zeros before using `reshape`.
- The output argument `w` should be an array of 1's and 0's (binary array) with `L` rows and `n` columns. Each column of `w` should be the `L`-bit Hamming code word for the `d`-bit binary data word stored in the corresponding column of the reshaped array `B`. Note that the code words are to be read down each column (in contrast to how we did it in class where we wrote the code words from left to right). In particular, the parity bits in the code words represented by the columns of `w` correspond to rows 1, 2, 4, 8, 16, *etc.*, whereas the data bits correspond to rows 3, 5, 6, 7, 9, *etc.* Thus the first step in constructing `w` is to simply copy the rows of `B` into the appropriate rows of `w`, namely, row 1 of `B` should become row 3 of `w`, row 2 of `B` should become row 5 of `w`, row 3 of `B` should become row 6 of `w`, *etc.* Finally, the parity rows should be filled in with 1's and 0's so that `mod(H*w,2)` is the zero vector, where `H` is the Hamming matrix for the `(L,d)` code.

You can test your function `int2hamm` on this example: to convert the ASCII codes for the string `'abcde'` into `(7,4)` Hamming code words, you would type

```
>> [w,bits] = int2hamm(double('abcde'),7,4)

w =

     1     1     1     1     1     0     1     1     1
     1     1     1     1     1     0     0     0     1
     1     0     1     1     0     1     0     1     0
     0     0     0     1     1     1     0     1     0
     0     1     0     1     0     0     0     0     1
     0     1     0     1     0     0     1     1     1
     0     0     0     1     1     1     1     0     0


bits =

     7
```

Thus we need 9 Hamming code words to represent the 5 ASCII integers; this is because 7 bits are needed to represent each ASCII character in this string, but each code word can accommodate only 4 data bits. The total number of data bits is `5*7`, so we need a total of `ceil(35/4)` (namely 9) code words.

As you may have already guessed, the next function you need to write is called `hamm2int` which should convert a binary array `w` of Hamming code words back into a row vector `N` of integers according to the following specifications:

- There should be three input arguments, `w`, `d`, and `bits` (all mandatory), and a single output argument `N`. The first input argument is the binary array `w` of code words to be converted back into integer values. The second input argument is `d`, the number of data bits in each code word (note that we do not need `L` as an input argument because we can get its value from `w`). The third input argument is `bits`, the number of data bits used to represent each entry in the output argument `N`.
- As part of the usual checks for bad input data, your function should generate an error if `(L,d)` does not represent a valid Hamming code, that is, if either `L<3`, `d<1`, `L>2^p-1`, or `L<2^(p-1)`, where `L` is the number of rows of `w` and `p=L-d` represents the number of parity bits in each code word.
- Your function should use the appropriate Hamming matrix `H` to check for correct parity in `w` and automatically correct bit errors (assuming at most 1 bit error per code word). If the product `mod(H*w,2)` has a column which is not a column of `H`, then issue a warning that multiple bit errors were detected in that column and do not perform error correction on the corresponding code word. (Note this will never occur when `L=2^p-1`.)
- To calculate `N`, you should first extract the rows of the (corrected) array `w` corresponding to the data bits (rows 3, 5, 6, 7, 9, *etc.*), placing these extracted rows into a binary array `B` (which should have `d` rows). You should then reshape `B` so that it has `bits` rows, but instead of padding `B` with zeros before reshaping, throw out the extra zeros that were originally padded to the data during the conversion in `int2hamm`. Thus the reshaped array `B` should have `bits` rows and `floor(d*n/bits)` columns, where `n` denotes the number of columns of `w`. After reshaping `B`, you can calculate `N` by simply calling the function `bin2int` you wrote above.

With `w` and `bits` defined as in the example above, your function should produce the following:

```
>> char(hamm2int(w,4,bits))

ans =

abcde
```

However, this simple test will not check whether or not the error correction part of your function `hamm2int` is working properly. To do this, you will need to generate errors in the binary matrix `w`. For example, to flip an average of 1.3% of the bits in `w` at random, you could use the following:

```
>> w = double(xor(w, (rand(size(w)) <= 0.013)));
```

Here the constant `0.013` represents the bit error rate of the channel; for a noisier channel you would increase this rate and for a cleaner channel you would decrease it.

At this point you have all the pieces necessary for simulating the block diagram shown on the first page of this assignment. Upload all five functions you have written. Also upload a `diary` file which demonstrates that your functions work as specified. Use the key `K` from the sample diary file below to perform the encryption/decryption steps. You may choose any message string for testing, but for best results it should have at least 40 characters. Also, try at least two different Hamming codes for transmitting your message, and try at least two different bit error rates (a small rate for which the error correction works flawlessly and a larger rate for which error correction fails and some characters are garbled on the receiving end).

```
>> % Sample diary file for this assignment
>> K = [ 1     3    -4     2     1
        -2    -5     9    -2    -3
        -3    -7    15    -1    -3
         1     5    -5     4    -4
         4    15   -12    18    13 ];
>> message = 'My social security number is 123-45-6789 and I was born yesterday.';
>> c = crypt(message,K)

c =

r,^?Ns=|?StXE*8O!)5irCgxb~N2hCti4Lj\KU$(3Sm1<!E-)G=n;N^SKm!UhO"Kh.cU.X

>> [w,bits] = int2hamm(double(c),15,11);      % (15,11) Hamming code
>> w = double(xor(w, (rand(size(w)) <= 0.01)));  % create bit errors
>> crypt(char(hamm2int(w,11,bits)), round(inv(K)))

ans =

My social security number is 123-45-6789 and I was born yesterday.

>> [w,bits] = int2hamm(double(c),12,8);       % (12,8) Hamming code
>> w = double(xor(w, (rand(size(w)) <= 0.02)));  % create bit errors
>> crypt(char(hamm2int(w,8,bits)),round(inv(K)))

ans =

My social security number.V!Y}23-45-6789 and I was bornm$f'Ferday.
```