School of Electronic and Electrical Engineering

UNIVERSITY OF LEEDS

# ELEC1401 Communications Networks and Signals

# MATLAB Sheet IV

---

**This practice sheet has been designed to help you better understand some forwarding and error detection techniques. You will continue practicing how to write MATLAB functions and employ modular programming techniques.**

**The tasks in this worksheet are for learning purposes, the marks associated with the tasks are for guidance only and they will not contribute to your final module mark. However, completing these tasks is essential to ensure that you understand the material as Matlab will be assessed in the January assessment (Assessment 2). You are expected to finish the tasks by the end of Unit 2 of this module. The sheet includes empty boxes below each task to write your answers, this should be useful when you review the Matlab sheet before the final assessment. You might find some of the tasks already explained in the screencasts, they are repeated here for completeness and for you to try it yourself if you have not done so.**

**MATLAB Help: In MATLAB you can get help in various ways. The simplest way, if you know the exact name of a particular function, is to type "help name_of_function" in the MATLAB command line. Alternatively, you can browse MATLAB help to find more about what you are looking for. In any of the problems below, if anything is unclear, or you don't know how to use a function, you should first try MATLAB Help. MATLAB commands and built-in functions appear in red in this sheet.**

**********

## Section 1: Error Detection

**Recommended time for completing the tasks in this section: 90 minutes**

We have seen that there are certain ways by which we can detect an error in a segment or a frame. Here we implement two of such techniques known as parity bits and checksum bits, and study how reliable they are in detecting errors.

a. Write a function that accepts a sequence of bits, and, as the output, calculates their corresponding parity bit, i.e., it tells you if there is an even number of "1"s, in which case the parity bit would be 0, or an odd number of "1"s, in which case, the parity bit would be 1. You may think of using xor in your code, or alternatively you may use sum and rem.

**Task 1:** Show a sketch of your algorithm, i.e., how you plan to write your code, here:

[1 mark]

These marks are for guidance only and they will **not** contribute to your final module mark

b. Find out what bitget does. Write a function that gets an integer number between 0 and 255 and it gives you its 8-bit representation.

c. Now write a function that accepts an arbitrary vector of integer numbers between 0 and 255, converts them to their 8-bit representations, and finds the parity bit for each column. For instance, if your input vector is [9 254 1], their 8-bit representation would be:

9 → 0 0 0 0 1 0 0 1

254 → 1 1 1 1 1 1 1 0

1 → 0 0 0 0 0 0 0 1

and the corresponding parity bits to each of the 8 columns would be

1 1 1 1 0 1 1 0

**Task 2:** Write your code below, how can you verify it works correctly?

[1 mark]

d. In the following function, A represents the matrix consisted of bits representing the original numbers in part (c), and their corresponding parity bits. That is, in the case of example above, A will be given by

0 0 0 0 1 0 0 1

1 1 1 1 1 1 1 0

0 0 0 0 0 0 0 1

1 1 1 1 0 1 1 0

Now, consider a general matrix A of bits zero and one. What does the following function do for 0 < p < 1? If you cannot figure it out, ask!

```
% start of the function
function Rec_A = Channel (A,p)
Err = rand(size(A));
Bit_Error = Err < p;
Rec_A = xor(A,Bit_Error)
%end of function
```

e. Optional: Use the function in part (d) to verify how reliable your parity bit solution is.

f. Now, let's find the checksum bits for the input vector in part (c) of integer numbers between 0 and 255. We have to add numbers one by one; if there is a carry-out bit, we wrap it around and add it to the result. For instance, in the example of part (c),

9

+ 254

= 263 > 255,

which means that there is a final carry-out bit. In fact, because 263 = 256 + 7, if we wrap around the bit 1 representing 256 to the first 8 bits, represented by 7, we will get 7 + 1 = 8. That will be the effective sum of the first two numbers. We add 8 to the last number on the list to get:

8

+ 1

= 9 < 256,

which means that there is no carry-out bit. The checksum bits are then given by 255 – 9 = 246.

**Task 3:** write a code that calculates the checksum bits for a vector of integer numbers between 0 and 255. How can you verify it works correctly?

[1 mark]

Optional: Similar to part (e), examine how reliable this technique is in detecting errors.

**Section 2: Longest Prefix Algorithm**

**Recommended time for completing the tasks in this section: 90 minutes**

One of the key tasks of the network layer is routing packets from the source to the destination. This can be done by having a forwarding table at each router that specifies to which output port of the router the received packet must be forwarded. Imagine the following forwarding table is used at a router:

| Destination IP address | Output port |
|---|---|
| 11000000110********** | 1 |
| 11000000111********** | 2 |
| 1100000010********** | 3 |
| Otherwise | 4 |

a. Using the longest prefix matching algorithm, specify the output port for each of the below IP addresses:

i) 192.255.109.203
ii) 192.194.109.203
iii) 192.128.109.203
iv) 192.127.109.203

b. Now let's write a code that, for the *specific* table above (and not necessarily a general one), takes the IP address in its conventional 4-digit form and return the relevant output port for that packet.

**Task 4:** Think of the algorithm by which you can specify the output port, and write it below. What functions/modules you would need for your code?

[1 mark]

c. Implement your code using MATLAB. Note that you might have already implemented some of the required functions.

**Task 5:** Write your code below. Verify it works correctly for the given IP addresses.

[1 mark]