OENG1207 – Digital Fundamentals

Laboratory Exercise 2

Ciphers

Laboratory Exercise 2: A Cipher Algorithm

Learning Outcomes

Upon successful completion of this practical task you will be able to:

- Explain how characters and other symbols are stored on computers using the ASCII system of encoding.
- Demonstrate how to use casting functions in MATLAB to modify data-types.
- Design and implement an algorithm utilising conditional/iterative programming structures and user-defined functions.

Introduction to the practical

'Secret codes' or **ciphers** have been around for hundreds if not thousands of years and have been used throughout history to secure communication between groups of people.

Often ciphers have been used in times of war to ensure sensitive information is communicated only to certain people; even if the enemy intercepts the communication all they should be able to read is gibberish unless they know the appropriate key to use to decrypt the message.

One well-known example of a cipher was the **Enigma code** used by the German military during World War II, since then ciphers have extended into the realms of **computer science** and **computer network security** with very complex algorithms now being used to encrypt communication between computers.
Modern encryption schemes include algorithms such as the **Advanced Encryption Standard (AES)**, the **Rivest-Shamir- Adleman (RSA)** system and the **Transport Layer Security (TLS)** system.

Part 2: Create a Program to Encrypt Messages

In this part of the practical we'll be taking the concepts from last week on converting characters to and from ASCII and binary to create a very simple encryption algorithm based on the **Exclusive OR** (XOR) operator.

Principle

The principle behind this type of encryption is based on a property of the XOR operator where:

$$(A \oplus B) \oplus B = A$$

Where the ⊕ symbol represents the **Exclusive OR operator**.

If we look at this in a **bitwise sense** this can be described as follows:

- We have two binary numbers (one could be the **plaintext** character we're encrypting and the other could be the **encryption key**).
- Performing a **bitwise XOR** on these two numbers produces some output, the **ciphertext**, e.g.:

0 1 0 0 0 0 0 1	**(plaintext character)**
⊕	
0 0 1 1 0 0 1 1	(encryption key)
=	
0 1 1 1 0 0 1 0	**(ciphertext)**

- Taking the result of this and performing a *second* XOR operation using the same **encryption key** will result in our original **plaintext** input, e.g.:

0 1 1 1 0 0 1 0	**(ciphertext)**
⊕	
0 0 1 1 0 0 1 1	(encryption key)
=	
0 1 0 0 0 0 0 1	**(plaintext character)**

Use this principle together with the preliminary task results from last week to:

- Design and implement a **user-defined function** that **generates an encryption key** and uses this to **encrypt the characters** in a character array. Write a second function to invert this operation (i.e. A **decoder**).

Task 1 – State the Problem and Determine the Input/Output

Exercise:

Using the information given on the previous page state the problem concisely and determine the input you have/need to solve the problem and what you need to output to accomplish the objective.

Points to be addressed:

- The problem statement should be **clear and concise**, double-check with your tutor if you are not sure of anything written in the problem statement.
- What input/output **data types** do you need (numeric, words, images, graphical output?).
- What **assumptions** are you making (if any) to help you solve/simplify the problem.

Task 2 – Design your Algorithms

Exercise:

Using the information from task 1 and your preliminary task, design how you should structure your program and the steps needed to perform the task. Note a few important requirements for your program:

- You need to have **at least one user-defined function** to encrypt/decrypt the message.
- You need at least one other script that **obtains the message** to be encoded and **calls the functions** that perform the encrypting/decrypting.

Points to be addressed:

- Make sure you check your algorithm will work using a short test message.
- Make sure you show the manual working for this part and ensure this working is neatly typed (not handwritten) in your write-up.

Task 3 – Write your program

Exercise:

Now, in a new MATLAB script file write the program that will encrypt and decrypt your message.

Points to be addressed:

- Compare the answers you get from MATLAB to your handworked answers from task 2. They should be the same, if not troubleshoot your algorithm/script to find where the mistake is.
- Also, **help your colleagues test their solutions** by writing an encrypted message into a text file. Give that text file to your colleagues along with the encryption key you used and see if their program can decode your message.
- Below is some code that will **write the data** to a text file and **read it back** into MATLAB:

```
% Write encoded message to a text file
fid = fopen('mycode.txt', 'w'); % Open the file for write
access 'w'
fwrite(fid, codedmessage); % Write message to open file
fclose(fid); % important, we must release the file back to
the Operating System after we're done
```

```
% Read encoded message from a text file
fid = fopen('mycode.txt', 'r'); % Open the file for read
access 'r'
codedmessage = fread(fid); % Read the data out of the file
fclose(fid); % important, we must release the file back to the
Operating System after we're done
```

Part 3 : A Brute-Force Decryption Algorithm

Breaking a code is always a security concern with any cipher. Our cipher is not particularly sophisticated so would take very little to break; there are only 256 possible combinations of encryption keys in our scheme which can be easily broken using a **'brute-force' attack**.

A brute-force attack on this type of cipher could utilise an **exhaustive key search** method which will basically try every possible encryption key combination against your encrypted message.

For this part of the practical create a function that can **brute-force decode an encrypted message** without knowing the encryption key that was used by using an **exhaustive key search method**.

Task 4 – State the Problem and Determine the Input/Output

Exercise:

Using the information given state the new problem concisely and determine the input you have/need to solve the problem and what you need to output to accomplish the objective.

Points to be addressed:

- The problem statement should be **clear and concise**, double-check with your tutor if you are not sure of anything written in the problem statement.
- What input/output **data types** do you need (numeric, words, images, graphical output?).
- What **assumptions** are you making (if any) to help you solve/simplify the problem.

Task 5 – Design your New Algorithm

Exercise:

Design how you should structure this part of the program and the steps your program will need to perform to achieve the objective.

Points to be addressed:

- Make sure you check your algorithm will work, step through the logic and make sure it can achieve the objective.

Task 6 – Use MATLAB to Solve the Problem

Exercise:

Now, in MATLAB write the code to perform your **brute-force attack**.

Points to be addressed:

- Compare the answers from MATLAB to what you expect, make sure your original message is shown somewhere in your output. If the original message is nowhere to be seen troubleshoot your algorithm/script to find where the mistake is.
- Again, you can **help your colleagues test their solutions** by writing an encrypted message into a text file (using the same method as in Task 3). Give that text file to your colleagues without the encryption key this time and see if their program can decode your message.

Table I: Some useful MATLAB functions

Function	Description
Casting Functions	

`cast(x, 'newclass')`	Changes the data type of x into the data type `'newclass'`.
	
	`'newclass'` can be one of the following:
	- `'single'` – single-precision float
	- `'double'` – double-precision float
	- `'int8'`- signed 8-bit integer
	- `'int16'` – same, but 16-bit integer
	- `'int32'` - same, but 32-bit integer
	- `'int64'` - same, but 64-bit integer
	- `'uint8/16/32/63'` – Same as above but unsigned integers
	- `'logical'` – logical (Boolean)
	- `'char'` - character
	
	Syntax examples:
	
	To change a double-precision floating point number to an 8-bit unsigned integer:
	
	`x = 17; % Defaults to double`
	`y = cast(x, 'uint8'); % casts x to an unsigned, 8-bit integer`
	
	You can also do this simply by using the new data type's name:
	
	`y = uint8(x); % Also casts x to an unsigned, 8-bit integer (alternative method)`
	
	To change a double-precision floating point number to a single-precision floating point:
	
	`y = cast(x, 'single');`
	`y = single(x); % alt. method`

Decimal to binary conversion
`de2bi(x, n, 'flag')`
	Converts a decimal number (whole-number) into its binary equivalent.
	
	x is the **number to be converted**.
	n is **how many bits** to return (small numbers will be zero-padded).
	`'flag'` can be either `'left-msb'` or `'right-msb'` and tells MATLAB if the binary number returned should have its **most significant bits** on the right or left.
	(**Note:** The default is right-msb).
	
	Syntax example:
	`x = 44; % Original decimal number`
	`y = de2bi(x); % Returns 001101`
	`y = de2bi(x, 8); % Returns 00110100`
	`y = de2bi(x, 8, 'left-msb'); % Returns 00101100`

Binary to decimal conversion.
`bi2de(x, 'flag')`
	Opposite of above; converts a binary number back into its decimal equivalent.
	
	x is the **binary number** to be converted.
	`'flag'` is the same as above `'left-msb'` or `'right-msb'` and tells MATLAB if the binary number being input has its **most significant bits** on the right or left.
	(Note: The default is right-msb).
	
	Syntax example:
	`x = [0 0 1 0 1 1 0 0]; % Original binary number`
	`y = bi2de(x); % Returns 52`
	`y = bi2de(x, 'left-msb'); % Returns 44`

Bitwise Operators	

|	Performs a bitwise OR operation on two variables or two arrays.
	Returns TRUE if at least one variable is TRUE.
	
	Syntax example:
	Using | on single values:
	`x = 1; % Single value`
	`y = 0; % single value`
	`z = x|y; % Returns 1`
	Or using | with an array:
	`a = [1 0 1 1 0 0]; % Array`
	`b = [0 0 1 0 0 0]; % Array`
	`c = a|b; % Returns [1 0 1 1 0 0]`

&	Performs a bitwise AND operation on two variables or two arrays.
	Returns TRUE if both variables are TRUE.
	
	Syntax example:
	Using & on single values:
	`x = 1; % Single value`
	`y = 0; % single value`
	`z = x&y; % Returns 0`
	Or using & with an array:
	`a = [1 0 1 1 0 0]; % Array`
	`b = [0 0 1 0 0 0]; % Array`
	`c = a&b; % Returns [0 0 1 0 0 0]`

`xor(x, y)`	Performs a bitwise Exclusive OR operation on two variables or two arrays. Only returns TRUE if one of the variables is TRUE and the other is FALSE.

	Syntax example:
	Using `xor()` on single values: x
	`= 1; % Single value y`
	`= 0; % single value`
	`z = xor(x,y); % Returns 1`
	Or using `xor()` with an array
	`a = [1 0 1 1 0 0]; % Array`
	`b = [0 0 1 0 0 0]; % Array`
	`c = xor(a,b); % [1 0 0 1 0 0]`

General useful functions	

`clear`	`clear` by itself will clear all variables from the workspace.
	If `clear` is followed by a variable's name(s), only that variable(s) will be cleared from the workspace.
	Syntax example:
	`clear % clears every variable`
	`clear var % clears var from workspace`
	`clear var1 var2 var3 % clears var1, var2 & var3 from workspace`

`clc`	Clears the command window.

`disp(x)`	Displays x on the command window. x can either be a string or a variable.
	Syntax example:
	`disp('Hello World!') %Displays Hello World! on the command window`
	`disp(x) %Displays the contents of the variable x on the command window`

`x=input('prompt')`
`x=input('prompt', 's')`
	Gets user input from the command window and stores that input in the variable x.
	Syntax example:
	`x=input('Enter a number: ')`
	`%Prompts user to enter a number onto the command line`
	`x=input('Enter a string: ','s')`
	`%Prompts user to enter a string onto the command line`

`length(x)`	Returns the number of elements in the array x.
`randi([min max], n, m)`	Generates an `n` × `m` matrix of pseudorandom integers between `min` and `max`.