Numerical Methods (NUM101) — Optional Coursework Cipher

This coursework consists of one part. It is worth 17 marks which will replace your worst coursework mark of the three regular courseworks (overall 17% of the credits for this unit).

Deadline	hand-in or upload?
18 May (Wed)	23:30 upload to Victory Assignment `cipher` **no** hardcopy to CAM
18 May (Wed) or before	demonstration of working demo script and function in lab to lecturer

Instructions and rules

- Material to be uploaded to Victory: function files `GenerateKeys.m`, `encrypt.m` and `decrypt.m` and a working demo script `demo.m`, and, if you need them, function files of other functions that you call inside your main functions.
- Marks will be awarded for your work only if you demonstrate the usage and the inner workings of your solution to the lecturer in the lab.
- Credit:

  100% code performs computation correctly and efficiently, is well structured and commented;

  ≥80% code performs computation correctly and efficiently

  ≥60% code performs computation correctly but has problems[1];

  ≥40% code does not perform computations correctly but could be made to work with minor corrections;

  ≥20% the intentions behind the code are discernible with some effort.

- This is **individual** coursework. **No declared collaboration is permitted**.
- For questions, clarifications and further help contact:

Jan Sieber (`jan.sieber@port.ac.uk`, office LG.146).

[1]for example, the function works correctly most of the time but fails for some valid arguments. Other examples of problematic constructions (look also for warnings in the Matlab editor):

– hard-coded 'magic' numbers spread throughout the code,
– functions that should be general but only work for this example,
– one part of the code is a repetition of another part,
– stray brackets, misleading variable names or variable usages (say, using `x(:)` if `x` is scalar),
– arrays grow inside a loop,
– a variable is defined but not used.

Question 1: A simple exponential cipher method

Write the necessary functions to implement a simple (Pohlig-Hellman) exponential cipher encryption. Specifically, you need three functions:

```
function [e,d,p]=GenerateKeys()
function enc=encrypt(message,e,p)
function msg=decrypt(enc,d,p)
```

Refer to your lecture on mathematical ciphers or to the short intro `expciphers.pdf` from NYU on Victory for background (or look around on the web).

Your function `GenerateKeys()` has to generate a random prime number p between 2^{51} and 2^{53}. This is as large as one can get with Matlab without introducing round-off errors for integers. The number e should also be random and less than $p-1$ and have no common divisor with $p-1$. The number $d < p-1$ has to satisfy $ed \equiv 1 \mod p-1$ (that is, the product ed should have the remainder 1 when divided by $p-1$) which makes d unique once e and p are chosen. All numbers, e, d and p, are the secret key.

The function `encrypt` should take a message as its first input, for example,

```
enc=encrypt('The Answer is 42!',e,p);
```

The second and third input are e and p as generated by `GenerateKeys()`. First it has to convert the message into a sequence of numbers (you can use the provided function `Alphabet`):

```
na=Alphabet();
num=Alphabet(message);
```

Then `num` is a sequence of numbers between 1 and `na-1` (inclusive). The output `na` is your alphabet length. Combine the message into blocks that are as long as possible For example, if `na=96` then the blocksize should be 7 (such that $\mathsf{na}^7 \leq p-1$). Say, `num=Alphabet(message);` produced the number sequence

```
num=[1,2,3,4,5,6,7,8,9];
```

then your plain text code $P_1 P_2$ should consist, for example, of the numbers

$$P_1 = 1 \cdot \mathsf{na}^6 + 2 \cdot \mathsf{na}^5 + 3 \cdot \mathsf{na}^4 + 4 \cdot \mathsf{na}^3 + 5 \cdot \mathsf{na}^2 + 6 \cdot \mathsf{na} + 7,$$
$$P_2 = 8 \cdot \mathsf{na}^6 + 9 \cdot \mathsf{na}^5.$$

How you arrange the numbers into blocks is your decision but you have to choose the blocks **as large as possible**. Then you have to encrypt each P_j using the formula: $E_j = P_j^e \mod p$. The sequence $E_1 E_2$ is the encrypted output of `encrypt`. The function `decrypt` then has to take $D_j = E_j^d \mod p$ and convert the D_j ($D_j = P_j$ hopefully) back into a sequence `num` of numbers $\leq \mathsf{na}$. The call

```
msg=Alphabet(num)
```

converts `num` back into text.

Total for Question 1: 17 marks

Hints and further instructions

- **Marking scheme** Marks get awarded only if all three functions work as required by the encryption scheme. Deductions from the maximal mark are then taken for flaws in the code according to the front sheet.
- Example demo script:

```
%% Cipher demo
clc;
[e,d,p]=GenerateKeys();

fprintf('p=%16.0f\ne=%16.0f\nd=%16.0f\n',p,e,d);
message='The Answer is 42!'

enc=encrypt(message,e,p);
fprintf('\nEncypted message: enc=\n');
fprintf('%16.0f\n',enc);

dec=decrypt(enc,d,p);
fprintf('\nDecypted message: dec=\n');
disp(dec);
```

Output:

```
p=3908325813950423
e=3053876101975591
d=2444341395459853
message =
The Answer is 42!

Encypted message: enc=
2867114737127414
 622392206836725
3666607571746716

Decypted message: dec=
The Answer is 42!
```

- There are several useful functions on Victory, which you can use:
  - `r=powermod(p,e,n)` calculates $r = p^e \mod n$ without calculating p^e first. You **must** use something like this to avoid overflow for large p and e.
  - `isprime=MillerRabin(p)` applies the Miller-Rabin test to check if p is a prime number (`isprime` is either `true` or `false` on return).
  - `Alphabet` can be used in three different ways:

```
na=Alphabet(); % gives length of alphabet
num=Alphabet(message) % converts text message to number sequence
message=Alphabet(num) % converts number sequence to text message
```

Useful Matlab functions: `r=gcd(p,q)`, `r=mod(p,q)`. Useless Matlab functions: `isprime` (works only for small numbers), `primes`.

- Double-checking with `Maple` is also very useful!