Information Theory for Data Science

Group 17

Pouria Mohammadalipourahari s327015

Reihaneh Kharazmi s328816

Arezoo Parsian s327860

Armi Okshtuni s328780

Prof. Garello

28 January 2024

Exercise 1

Point 1:

Generate a 127-bit M-sequence v(n) with the characterization:

- **m=7 cells**
- **polynomial $D^7 + D^4 + 1$**

The procedure is that we will shift the register to the right by one bit at each step and calculate the new leftmost bit (the feedback bit) based on the values at the tap positions. For the polynomial $D^7 + D^4 + 1$, the feedback bit is the XOR (exclusive OR) of the bits in positions 7 and 4. The next steps are as follows:

- Shift the register right.
- Calculate the feedback bit: XOR the bits in positions 7 and 4 of the LFSR.
- Place this feedback bit into the leftmost position of the LFSR.
- The bit that is shifted out (from the rightmost position) is the next bit in your M-sequence.

These steps are continued to generate 127 bits.

Consequently, the generated 127-bit M-sequence is:

'0 1 1 0 1 1 1 1 0 0 1 1 1 0 0 1 0 1 0 1 1 0 0 1 1 0 0 0 0 0 1 1 0 1 1 0 1 0 1 1 1 0 1
0 0 0 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 1 0 1 0 0 1 1 1 1 0 1 1 1 0 0 0 0 1
1 1 1 1 1 1 0 0 0 1 1 1 0 1 1 0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 0 1 0 1 0 1 0 0 0 0 1'

Number of 1s (N1): 64

Number of 0s (N0): 63

Is N1 equal to N0 + 1? Yes, the result satisfies the property that N1 = N0 + 1.

Point 2:

Prove that for an M-sequence we always have N1 = N0 + 1.

An LFSR cannot start in the all-zero state when generating an M-sequence, as this state would trap the LFSR in an endless loop of zeros. Thus, the sequence must start with at least one '1'. Since the sequence cycles through all other non-zero states, there must be at least one more '1' than '0' to balance the count when it returns to the initial state. This leads to $N1=N0+1$.

Point 3 and 4:

- **Write a table with the values of NR0(i) and NR1(i).**
- **Compare the values of NR(0) and NR(1) against the run properties of an ideal binary random sequence.**

The number of runs of 0s and 1s for each run length is written in the following table:

Run Length	No. of runs for '0'	No. of runs for '1'	Ideal No. of runs
1	16	16	31.75
2	8	8	15.875
3	4	4	7.9375
4	2	2	3.9688
5	1	1	1.9844
6	1	0	0.99219
7	0	1	0.496094
8	0	0	0.248047
9	0	0	0.124023
10	0	0	0.062012
…	*	*	**

* The number of runs is 0 from here onwards

** The number of runs is less than 0.05 from here and tends to zero.

Point 5:

Prove that we cannot have a run of m zeros, or a run of m + 1 ones.

A run of m zeros in the sequence would imply the LFSR reaches the all-zero state. This is because an LFSR with m stages holds its last m outputs. If all these outputs were zero, the LFSR's current state would be all zeros.

However, it is known the all-zero state is not part of the M-sequence, as it would halt the sequence generation. Therefore, a run of m zeros is not possible in the M-sequence.

A run of m+1 ones implies that, for m+1 consecutive steps, the feedback bit (which becomes the new leftmost bit after each shift) is a '1'. The feedback bit is determined by the XOR operation of specific bits according to the LFSR's feedback polynomial.

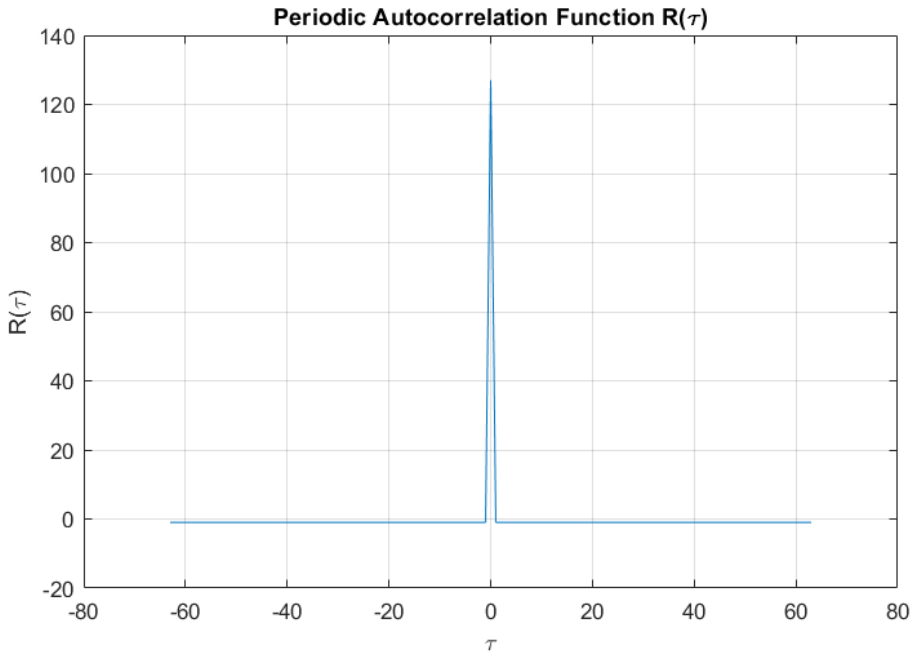
For an LFSR of length m, the feedback bit being '1' m+1 times consecutively requires a specific sequence of states. However, due to the properties of XOR and

the nature of the feedback polynomial, it's not possible for this to occur m+1 times in a row without encountering the all-zero state, which, as established, cannot occur in the sequence. Thus, a run of m+1 ones is not possible in the M-sequence.

Point 6:

Compute and plot the periodic autocorrelation function.

The periodic autocorrelation function R(τ) is depicted in the following plot:



It can be seen that the R value is always -1 for all non-zero points and is 127 at point 0.

Point 7 and 8:

- **Compute the primitive polynomial of the LFSR.**
- **Generate the next 20 bits of the key.**

To answer to these parts, it is tried to ascertain the internal configuration of a Linear Feedback Shift Register by analyzing a portion of its output, which is in this case, a sequence of 40 bits.

The steps to fulfil the requirements are as follows:

- First, we assume that m = 3.

- Then according to the binary linear equation b = Ax, we will extract b and A matrices to calculate x matrix.
- If x exists, then we will check if the successive bits are correct or not.
- If yes, we can say that m is degree of polynomial, and we can generate the next 20 bits of the key. Otherwise, we increase m by one and repeat the steps.

According to our specific key:

s = [1 0 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 0 1 1 1 1 0 1 0 0 1 1 1 0 0 1 0 0 0 0 1 0 1 0]

The answer is as follows:

The primitive polynomial is:

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

The polynomial degree is:

19

The next generated 20 bits are:

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

2. Mc Eliece cryptosystem

In this question we followed the instructions and implemented a python code which resulted in name of a city, DUBLIN.

Matrix G, H, S and P:

```
G =
[0,1,1,0,0,1,0;1,1,0,1,0,0,0;0,0,1,0,1,1,1;1,0,1
,0,0,0,1];
H = [0,1,1;1,0,0;0,0,1;1,1,1;1,1,0;1,0,1;0,1,0];
S = [0,0,0,1;1,0,0,1;1,0,1,1;0,1,0,0];
P =
[0,0,0,0,0,0,1;0,1,0,0,0,0,0;0,0,1,0,0,0,0;0,0,0
,0,1,0,0;0,0,0,1,0,0,0;1,0,0,0,0,0,0;0,0,0,0,0,1
,0];
```

Our cipher text was:

```
[0,1,0,1,0,0,1,0,1,1,1,1,0,1,0,1,0,1,0,0,0,0,1,0
,1,0,1,1,1,1,0,0,1,1,1,0,1,0,1,0,0,1,0,1,1,1,1,0
,0,0,1,1,1,0,1,1,0,1,1,1,1,1,0,0,0,1,1,0,0,1,1,1
,1,0,1,1,1,1,1,1,1,0,0,1,];
```

A. Program

- LUT:

To implement a code for LUT we initialised and populated a Look-Up Table (LUT) for error correction in a coding system. It starts by setting up an empty dictionary (**lut**). It then considers the all-zero error vector, calculates its syndrome using a parity check matrix (**H**), and stores the mapping in the LUT. Following that, it generates error vectors of weight 1, calculates their syndromes, and adds these mappings to the LUT as well.

Therefore LUT looks like this:

```
{(0, 0, 0): [0, 0, 0, 0, 0, 0, 0],
 (0, 1, 1): [1, 0, 0, 0, 0, 0, 0],
 (1, 0, 0): [0, 1, 0, 0, 0, 0, 0],
 (0, 0, 1): [0, 0, 1, 0, 0, 0, 0],
 (1, 1, 1): [0, 0, 0, 1, 0, 0, 0],
 (1, 1, 0): [0, 0, 0, 0, 1, 0, 0],
 (1, 0, 1): [0, 0, 0, 0, 0, 1, 0],
 (0, 1, 0): [0, 0, 0, 0, 0, 0, 1]}
```

1. First we devided the cipher text to n-bit vectors (n = G.shape[1] = 7). So as the cipher length is 84 we would have 12 vectors with 7-bit length:

```
y1: [0, 1, 0, 1, 0, 0, 1]
y2: [0, 1, 1, 1, 1, 0, 1]
y3: [0, 1, 0, 1, 0, 0, 0]
y4: [0, 1, 0, 1, 0, 1, 1]
y5: [1, 1, 0, 0, 1, 1, 1]
y6: [0, 1, 0, 1, 0, 0, 1]
y7: [0, 1, 1, 1, 1, 0, 0]
y8: [0, 1, 1, 1, 0, 1, 1]
y9: [0, 1, 1, 1, 1, 1, 0]
y10: [0, 0, 1, 1, 0, 0, 1]
y11: [1, 1, 1, 0, 1, 1, 1]
y12: [1, 1, 1, 1, 0, 0, 1]
```

2. In order to computing y_1 = y(P_inverse) our code performs operations on a matrix P, including finding its inverse (**P_inv**). It then multiplies a set of matrices (**y_matrices**) with the inverse matrix (**P_inv**), storing the results in **y1_matrices**. Finally, it converts the resulting matrices back to binary strings (**y1_binary_strings**). The code prints the inverse matrix **P_inv**, the matrices obtained by multiplying each **y_matrix** with **P_inv**, and the binary representation of these matrices.

```
P_inv = [[0. 0. 0. 0. 0. 1. 0.]
 [0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0.]
```

[0. 0. 0. 0. 1. 0. 0.]

[0. 0. 0. 1. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 1.]

[1. 0. 0. 0. 0. 0. 0.]]

y_1 for each vector:

```
y1_1: 1100100
y2_1: 1111100
y3_1: 0100100
y4_1: 1100101
y5_1: 1101011
y6_1: 1100100
y7_1: 0111100
y8_1: 1110101
y9_1: 0111101
y10_1: 1010100
y11_1: 1111011
y12_1: 1110110
```

3. Applying syndrome decoding to decode y_1 in order to obtain v1

We computed the syndrome s using the formula s = y_1* H. After finding syndrome, we extracted the corresponding error vector from LUT. Then we obtained the received codeword c1 using this formula: c1 = y_1 + e.

Then we obtained the corresponding information vector v using a function in python similar to gflineq(G',c',2) which was gf2_solve(G',c').

This function takes a matrix **G.T** and a vector **c.T** as input, augments the matrix, performs Gaussian elimination in GF(2), and then applies back-substitution to solve the system of linear equations. The result, **solution_GF2**, is a vector representing the solution in GF(2).

Results for each vector is shown in the code.

4. Obtain v = v1*S_inverse

We defined a function for calculating the multiplication of two matrixes in GF(2), then in order to computing v. The function takes two matrices, **A** and **B**, as input and returns their product in the GF(2) field.

```
S_inverse = [[1 1 0 0]
 [0 0 0 1]
 [0 1 1 0]
 [1 0 0 0]]
```

Therefore v is:

```
[[0  0  1  0]
 [0  0  1  0]
 [1  0  1  0]
```

```
[1 0 1 0]
[0 1 0 0]
[0 0 1 0]
[0 0 1 1]
[0 0 1 0]
[1 0 0 1]
[0 0 1 0]
[0 1 1 1]
[0 0 1 0]]
```

5. Final step: Apply the ASCII decoding to obtain the name of the city

We First change the shape of v and extract every paires of it. so it will be 6 pairs as v has shape 12x4.

Then for each pair flip them and then change the order of 2 vectors.

Now we can combine them together and calculate the decimal value of each pair which is 8 bit.

Now we can find the corresponding char of the decimal value.

The decimal values we got for each pair was: 68, 85, 66, 76, 73, 78

Which was DUBLIN!

B. Question

In the context of the McEliece cryptosystem, the parameters k and n are crucial in determining the security and efficiency of the scheme.

1. k (Information Word Length):
   The parameter k represents the length of the information word, which is the original message or data that is to be transmitted securely. In the McEliece cryptosystem, the information word is encoded into a longer codeword of length n.

2. n (Code Word Length):
   The parameter n represents the length of the codeword, which is the result of encoding the information word using an error-correcting code. The codeword is then transmitted over the communication channel.

In practical use, picking values for k and n depends on how much security is needed and how efficient the system should be.

Bigger n values usually make it harder for attackers to figure out the original message, boosting security. However, larger n

values also mean longer keys and slower encryption/ decryption.

For instance, the NIST PQC project sets standards for post-quantum cryptographic algorithms, offering recommended values for various cryptographic methods, including McEliece. These recommendations aim to find a balance between security and efficiency. We checked the NIST PQC project documentation about those values:

In another article the use of quasi-cyclic codes permits to reach a public key of as low as 6500 bits for a security of 280 or 11,000 bits for 2107.

Reference

(Just click on “Reference” word)

For the classical McEliece, its public key size ranges from 0.25 to 1.3 megabytes.

Reference

(Just click on “Reference” word)

There is also another useful article which examines security with trying with different values of n an k:

Sec. Level	Ref.	(n, k, t)	PK size [kB]	
			Full PK	Systematic
50	[37]	(1024,524,50)	66	32
80	[6]	(2048,1751,27)	438	64
80	[40]	(1702,1219,45)	254	72
80	[12]	(2048,1696,32)	424	73
128	[7]	(3178,2384,68)	925	232
128	[12]	(4096,3604,41)	1802	217
256	[7]	(6944,5208,136)	4415	1104

Typical implementation then selects t in such a way that the code rate based on k = n − mt is the most secure choice for given n = 2^m.

PKC **n, t, Security bits**	**Device**	**Computation time** **Encrypt, Decrypt**	**Ref.**
Goppa MECS $n = 1024, t = 40$, 62-bit sec.	Infineon SLE76CF5120P controller, 16-bit CPU @ 33 MHz	970ms, 690ms	[60]
Goppa MECS $n = 2048, t = 50$, 102-bit sec.	Infineon SLE76CF5120P controller, 16-bit CPU @ 33 MHz	1390ms 1060ms	[60]
Goppa MECS $n = 2048, t = 27$, 80-bit sec.	AVR ATxMega192, 8-bit CPU @ 32MHz	450ms, 618ms	[19]
QC-MDPC MECS $n = 9600, t = 84$, 80-bit sec.	AVR ATxMega256A3, 8-bit CPU @ 32MHz	800ms, 2700ms	[32]
Cryptosystem	**Device**	**Computation time**	**Ref.**
ECC-P160 (SECG)	ATMega128@8MHz	203ms (at 32MHz)	[19]
RSA-1024	ATMega128@8MHz	20748ms (at 32MHz)	[19]

…

Reference
(Just click on “Reference” word)

When choosing parameters, it's crucial to think about the trade-offs between security, key size, and how fast the system works, based on what the application needs.