Project Information Theory
group A

Lennert Jacobs, Sander Cornelis, Rafał Muszyński
{Lennert.Jacobs,Sander.Cornelis,Rafal.Muszynski}@UGent.Be

February 21, 2023

1 Introduction

This document describes the project assignment for the course on information theory (odd group numbers). In this project, the theory from the course will be applied in the creation of Quick-Response codes (QR-codes). This project is carried out by groups of 3 or 4 people and consists of writing a report (in English) and implementing the QR-code in Python. The report and code should be uploaded to Ufora as a zip-file with the name groupXX.zip with a map called groupXX containing all the files. **The project deadline is the 5th of May 2023.**

2 Task division

The project report should contain a section on how you divided the tasks among the group members and who eventually did what. You should take into account that the implementation of the BCH and Reed-Solomon decoders are the hardest tasks. Try to make a fair division! After the project deadline, there will also be an oral defense and a discussion of your task division. For the oral defense it will be assumed that every group member can answer questions about every part of the project. The planning of this will be made available on Ufora.

3 QR code

QR-codes are two-dimensional barcodes that have grown very popular in the last years due to their high data capacity and quick readability. To protect the information inside the barcode from errors caused by dirt, scratches, etc., the data is encoded by a Reed-Solomon code and codewords are scattered over the barcode. The full specification of the QR code can be found in the file *QR_specification.pdf* on Ufora. This file will be required for solving the problems in the project.

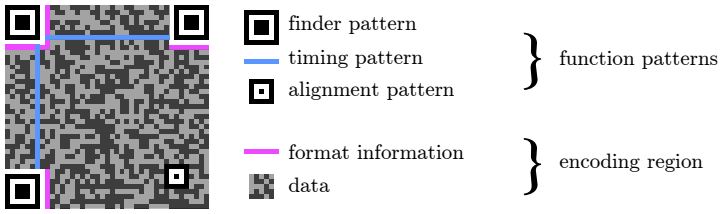

Figure 1: Structure of a version 6 QR code.

The QR code is designed such that it scales in size, depending on the amount of information inside the barcode and the error correcting capability of the code. The size of the matrix is determined by the version number, starting from 1 through 40, where version 1 is a 21×21 matrix and every version adds an additional 4 rows and columns to the matrix. In this project we will work with a version 6 code, which amounts to a 41×41 matrix. For every version, there are 4 available error correction levels with increasing capability of correcting errors: L, M, Q and H.

In Figure 1, the structure of a version 6 QR code is shown. In this barcode we can distinguish different areas. Most prominent are the 3 large square structures at the left bottom and two top corners, these are required to identify the code as a QR code. The blue row and column are the timing patterns that alternate between black and white pixels. This is used by the QR-reader to create a matrix grid. The smaller square pattern in the bottom right corner is the alignment pattern and is required to correct for any skew. For higher versions, more of these alignment patterns will appear. The pink area is reserved to contain format information required to properly read the data from the QR code. The actual (encoded) data is contained in the gray area. The area taken by the data and format information is called the encoding region, the remaining area is used by the function patterns.

4 Assignment

4.1 Creating the QR code

1. In the first step we must generate the binary data stream that will be encoded in the barcode. The QR code offers the possibility to store data of different formats such as numeric, alphanumeric, kanji, etc. This way, special Greek or Japanese symbols can be stored in the QR code. For the Python implementation we will only consider the alphanumeric format. The first step will consist of mapping the alphanumeric symbols to a binary sequence. This sequence is preceded by header information bits that indicate what format was used for the symbols. More information and examples can be found in the specifications under section 8.3 Modes.

- Describe step by step how to obtain the binary sequence (including headers) for the alphanumeric string: *X5*P$*
- Will this data fit in a version 1 QR code?
- Explain what is the minimal required version number to encode the following data in a QR code using error-correction level H: (a) *149382746106937491* and (b) *I <3 Information Theory*
- The ECI mode offers the possibility to use custom alphabets or to use the 'structured append'. Explain what the latter is.
- Implement the function ***generate_dataStream()*** that only works with the alphanumeric mode.

2. In order for the decoder to decode the data from the QR code, it must know what error-correction level and mask (see later) was used. This information is stored in the format information (FI) which is placed at the designated spots in the QR code. This FI, totaling 5 bits, is coded by a binary (15,5) BCH code with the following generator polynomial:

$$g_{\mathrm{BCH}}(x) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1 \tag{1}$$

   - Draw the shift register that computes the systematic codewords of this code.
   - Determine the error correcting capability t of this code.
   - Implement the function ***encodeFormat()***

3. Next, we will encode the data stream. We wish to use the error correction level Q. From the specifications, we know that we must split the data stream in 4 blocks where every block represents a single codeword. The code used is a shortened ($n < p^m - 1$) Reed-Solomon code with the following generator polynomial:

$$\begin{aligned} g_{\mathrm{RS}}(x) \quad = \quad & x^{24} + \alpha^{229}x^{23} + \alpha^{121}x^{22} + \alpha^{135}x^{21} + \alpha^{48}x^{20} + \alpha^{211}x^{19} \\ & + \alpha^{117}x^{18} + \alpha^{251}x^{17} + \alpha^{126}x^{16} + \alpha^{159}x^{15} + \alpha^{180}x^{14} + \alpha^{169}x^{13} \\ & + \alpha^{152}x^{12} + \alpha^{192}x^{11} + \alpha^{226}x^{10} + \alpha^{228}x^{9} + \alpha^{218}x^{8} + \alpha^{111}x^{7} \\ & + x^6 + \alpha^{117}x^5 + \alpha^{232}x^4 + \alpha^{87}x^3 + \alpha^{96}x^2 + \alpha^{227}x + \alpha^{21} \end{aligned}$$

   where α is the primitive element of GF(2^8). The primitive polynomial is given by: $p(x) = x^8 + x^4 + x^3 + x^2 + 1$; this is also the default primitive polynomial of the galois package.

   - Implement the function ***makeGenerator()***, verify the generator polynomial $g_{RS}(x)$ and give the generator polynomial of a 7 error-correcting Reed-Solomon code in GF(5^4). Use power representation for the coefficients of the polynomial. Note: we follow the QR code specifications, i.e. $\boldsymbol{m_0 = 0}$.

- Implement the functions ***encodeRS()*** and ***encodeData()***. Note: *encodeRS()* should be a general Reed-Solomon encoder; use it in *encodeData()* to construct the final message codeword sequence as described in the specifications.
- Can you give a possible explanation why the BCH code is used for the format information and the RS-code for the actual data?

4. In this last step we will create our first QR code using the function *generate()*. This function places the format information and encoded data in the QR-matrix according to the specifications, and adds the function patterns. When a specific mask is selected, the *generate()* function uses this mask to generate the QR code. However, when 'optimal' is selected, the *generate()* function first determines the optimal mask by generating the QR code for each mask and selecting the mask that optimizes the mask score. Subsequently, the QR code is generated using the optimal mask.

   - What is the purpose of the mask and why do there exist multiple masks? What is going on with the pattern 1011101 in the QR code?
   - Generate a QR code with the following data: *GROUPXX SOME TEXT,* where XX represents the group number and the text is free to choose. Show this QR code with mask code 000 in the format information. Now encode the QR code with the optimal mask. Discuss the difference. Test the QR-codes using your smartphone QR reader.
   - In the specifications the Reed-Solomon code operates over GF(2^8), what if GF(5^4) was used instead? How would you change the specifications, what would be the consequences? Remember that the information to be encoded in the QR code is binary.

4.2 Reading the QR code

Now that we can generate a QR code, we shall try to read it as well. The reading process consists of two main steps. In a first step, the scanned QR code must be converted to a binary matrix. For this, the function patterns are employed to rotate, scale and skew the scanned image. Once the QR code is obtained in binary format, the data will be decoded and possible errors will be corrected in the second step. In this project we will focus on this second step, i.e., we will assume that we have the QR code available in matrix format.

1. First we must decode the format information (FI) to obtain the error-correcting level and mask. Notice that the FI is embedded twice in the QR code. Specifications state that whenever it fails to decode the upper-left format information, the other format information should be used. If this also fails, the QR code cannot be read.

   - Implement the function ***decodeFormat()***, which is the inverse of the function *encodeFormat()*. In the specifications, in Annex C (p. 76),

a decoding procedure is outlined that differs from the decoding techniques seen in the course. However, it is closely related. You are free to choose the decoding algorithm. Specify in your report which algorithm you chose.

- Calculate the probability of a decoding error of a single FI codeword when the bits have been altered by a binary symmetric channel (BSC) with parameter p. Give an approximation for $p \ll 1$. Plot the curve for $p = 5 \times 10^{-1},\ 2 \times 10^{-1},\ 1 \times 10^{-1}, 5 \times 10^{-2},\ 2 \times 10^{-2},\ 1 \times 10^{-2}$. Compare this with the values obtained by simulation. Use a logarithmic scale for both axes and make sure to simulate enough (more than 1 codeword) to obtain statistically relevant results!
- In the QR code the FI-codeword is embedded twice. What is the probability that neither of the FIs can be reconstructed correctly? Plot the result together with the previous plot for various values of p.

2. With this information, the mask can now be released and the data decoded.

   - Implement the functions ***decodeRS()*** and ***decodeData()***; these are the inverse functions of *encodeRS()* and *encodeData()*, respectively. You are free to choose the decoding algorithm in *decodeRS()* but note that the algorithm in Annex B cannot be used as is when not working in GF(2^8). The frequency-domain decoding algorithm from the course (Algorithm 2.4.1) works for arbitrary galois fields, but assumes full-length codes ($n = p^m - 1$). However, the algorithm can be readily applied to decode shortened codes ($n < p^m - 1$), by taking into account that the parameter n in the decoding process denotes the full length $p^m - 1$ rather than the length of the shortened code. For instance, the vector $\hat{\mathbf{E}}$ will have length $p^m - 1$ despite the shortened code length. Specify in your report which decoding algorithm you chose.
   - Verify that the 7 error-correcting Reed-Solomon code over GF(5^4) you created in 4.1 question 3 can indeed correct up to 7 errors.
   - Simulate the bit error rate after decoding of the version 6-Q QR code where the bits have gone through a BSC with parameter p. Plot the curve for $p = 2 \times 10^{-1}, 1 \times 10^{-1}, 5 \times 10^{-2}, 2 \times 10^{-2}, 1 \times 10^{-2}, 5 \times 10^{-3}$ and discuss.
   - Is a BSC representative for barcodes? Under what circumstances would the BSC apply and when not? Also, can you explain the function of the interleaver?

3. In the final step, the binary sequence obtained after decoding must again be mapped to a readable alphanumeric sequence.

Figure 2: Create your customized QR code.

- Implement the function ***read_dataStream()***, which is the inverse of the function *generate_dataStream()*.

4. Congratulations! You are now able to generate and decode a QR code, capable of sustaining (heavy) damage. Due to the strong error correcting capabilities, some designers have purposely introduced errors in the QR code to make it more customized or beautiful.

   - Try to customize your own QR code by introducing a figure or a logo. Let your creativity run wild! Explain why you are sure your code will still work after the customization. (You are allowed to use a different error correction level if you wish to make more severe alterations)

5 Python implementation

As a basis of the Python implementation, a Python script is provided on Ufora. In the *QR_code.py* file you will find a class that describes the QR code with functions on how to generate and read a QR code. You will be asked to complete some of the functions in *QR_code.py*. It is important that the functions use the correct input and output. This is necessary such that the files you submit can be tested for proper functionality. Notice that all files will also be tested for plagiarism.

To do your calculations in a Galois field, you must use the galois package (https://galois.readthedocs.io/en/stable/index.html). For this project, you will be asked to implement the BCH and Reed-Solomon encoders and decoders, using the galois.GF() and galois.Poly() classes. Do not use built-in Python implementations or packages for RS encoding/decoding. The galois.GF() class is used to define finite fields and their elements, whereas the galois.Poly() class is used to create polynomials over finite fields. Note that galois.Poly() has a number of interesting alternate constructors based on, e.g., the polynomial's non-zero

degrees or roots. The quotient and remainder of a polynomial division can be determined with the // and % operators, respectively. The product of polynomials is obtained using the regular $*$ operator. When defining an extension field, always use the default primitive polynomial, given by *galois.primitive_poly(p, m)*. Some useful commands are given in Table 1. Additional commands and information can be found in the documentation of the galois package. Make sure to provide sufficient comment in your code.

Function	**Description**
pp = galois.primitive_poly(p, m) *GF = galois.GF(p**m, irreducible_poly=pp)*	Creates an extension field GF(p^m) with the default primitive polynomial
print(GF.repr_table()) *[or print(GF.display_table()) in older versions of galois]*	Shows all elements of the Galois field GF (power, polynomial, vector and integer representation)
x = GF(y)	Converts an integer y to an element x of the GF (also works for arrays)
f = galois.Poly(coeffs, field=GF) *or f = galois.Poly(GF(coeffs))*	Converts an array/list of coefficients into a polynomial f with coefficients in GF

Table 1: Some useful Python commands.