ERROR CORRECTION AND QR CODES

GRANT BARLAND
ADVISOR: SARAH E. ANDERSON

1. Introduction

Coding theory is the study of how one communicates reliably and efficiently. When information is sent across a channel, noise is likely to occur which will corrupt the original message that was sent across the channel. In coding theory, one wants to detect, or even correct, errors that occur due to noise. In today's digital world, questions about how one can communicate effectively and reliably are incredibly important. Applications of coding theory include: ISBNs, flash memories, CDs, DVDs, and much more. *We study coding theory through the lens of QR codes, and our project was to create a program that encodes QR codes in SAGE.*

A QR (quick response) code is a two-dimensional bar code initially designed by the automotive industry in Japan in 1994 to track parts. Today they are used in advertising, storing bank account or credit card information, visas in passports, coins, and much more. Since QR codes are seen everywhere from product packaging to coins to visas, they must stand up to the wear of everyday use. QR codes may also be customized by advertisers such shown in Figure 1. This inserted logo is error intentionally introduced to the data. Reed-Solomon codes are used in QR codes to recover the missing information.

Figure 1. Reed-Solomon codes correct errors introduced to QR codes.

The paper is organized as follows. In Section 2, we give an outline of our program. In Sections 3 and 4, we explain converting our message to a data string and adding error correction codewords using a Reed-Solomon code based the given level of error correction. In Section 5, we describe the structure of a QR code and how to put our encoded message into this structure.

2. Outline of program

Our program will allow a user to input data and level of error correction, and it will output a QR code. There are four different types of data that may be encoded with a QR code: numeric, alphanumeric, byte, and Kanji. Our program will only allow a user to input alphanumeric data, which consists of the characters shown in Table 4 (see Appendix A). In addition to the input data, the user chooses a level of error correction which determines how much data can be corrected for the program to correctly decode the message. There are four levels of error correction, which are shown in Table 1.

Table 1. Different Levels of Error Correction

Level	**Percentage of information that can be restored**
L (Low)	7 %
M (Medium)	15 %
Q (Quartile)	25 %
H (High)	30 %

Encoding the data with the specified level of error correction is a two step process. First, one encodes and breaks the message up into 8-bit long codewords. Next, one adds error correction codewords using a Reed-Solomon code. Once the message is encoded, one puts the data into the correct structure of a QR code. A full version of our program may be found in Appendix C and follows the steps described in [2].

3. Encoding The Message

In this section, we will describe how the data and error correction level is encoded into 8-bit codewords. Below is an example input message and error correction level. This example also shows mode and version levels which will be explained next.

```
input='QR CODES'
EC = 'M'
```

The data is encoded using the following steps:

(1) First, determine the smallest version of QR code needed for the data we are encoding. There are 40 versions which all have different capacities for data storage, and we want to use the least amount of data as possible. The smallest QR code size is 21×21 modules (version 1), and the largest is 177×177 modules (version 40). The version should be automatically determined from the error correction level, the data type, and amount of data. In our example, we are only concerned with messages in alphanumeric format, which means we can determine the amount of data by observing the error correction level and message length. Our example “QR CODES” will be version 1.

```
if EC == 'L':
    if l<25:
        version=1
    elif 25<=l<47:
        version=2
elif EC == 'M':
    if l<20:
        version=1
    elif 20<=l<38:
        version=2
```

(2) Second, we determine the mode indicator. The mode based which on the five character types shown in Table 2 is inputed. Our program only allows alphanumeric messages, so our mode will always be ‘0010.’

TABLE 2. Different Character Modes

Mode Name	**Mode Indicator**
Numeric	0001
Alphanumeric	0010
Byte	0100
Kanji	1000
ECI	0111

(3) Next, we break the message up into pairs of twos, and convert the pairs to a binary equivalent. The function “split” will loop through the message and return a list of the characters in pairs of twos.

```
def split(fn):
   while fn:
      yield fn[:2]
      fn = fn[2:]
```

```
list(split(input)) = ['QR', 'C', 'OD', 'ES']
```

Next, loop through the new list to perform the following functions:

(a) Convert letters to their numeric equivalent shown in Table 4 (see Appendix A).
(b) Multiply the first number in every pair by 45, then add it to the second number.
(c) Convert the remaining number in each position of the string to its 11 bit binary equivalent.
(d) Convert the last remaining number to its 6 bit binary equivalent if it is an odd length.

The resulting code is as follows.

```
a=list(split(input))
length=len(a)
encoded=''

for i in range(0,(length)):
   a[i]=a[i].replace('0','00').replace('1','01').replace('2','02').replace('3','03')...etc

   if len(a[i])==4:
      a[i]=(int((a[i])[:2])*45)+(int((a[i])[2:5]))
      a[i]='[0:011b]'.format(a[i])
   elif len(a[i])==2:
      a[i]='[0:06b]'.format(int(a[i]))

   encoded=encoded+a[i]
```

The output after program is executed for the example of “QR CODES” is shown below.

```
encoded = '1001010110111001100000100010001010101010010010'
```

(4) Finally, to create the encoded message, add the mode indicator shown in Table 2 and a binary message length indicator to the front of the encoded data string, then break it up into bits of length 8.

```
l = len(input)
charcount = '{0:09b}'.format(l) #creates binary equivalent of message length
datastring = (mode+charcount+encoded)
```

Note, it may be necessary to pad the end of message with zeros if it does not meet length requirements. The steps are as follows.

(a) Add up to four zeros onto the end of the message until it reaches the required length in Table A (see Appendix A.). If after adding four zeros, the message still does not meet the requirement, then move on to step b.

(b) Pad the message on the right with zeros until its length is divisible by 8. If the message is still too short for the requirement, then add the bytes "11101100" and "00010001" in an alternating pattern until the requirement is fulfilled.

The resulting data string is for the example "QR CODES."

```
datastring = '00100000 01000100 10101101 11001100 00010001 00010101 01001001
    00000000 11101100 00010001 11101100 00010001 11101100 00010001 11101100
    00010001'
```

4. Error Correction

The next step is to add error correction codewords to the message. The amount of error correction codewords needed based on the version and error correction level can be found in Table A (see Appendix A.).

QR codes use Reed-Solomon codes to construct error correction codewords. Reed-Solomon codes were introduced in 1960 [3] and use polynomials over finite fields to correct for missing data. They can be found in technologies such as DVD's, broadcast systems, and computer storage. The first step to encode a message $(m_0, m_1, ..., m_{n-1})$, where $x_i \in \mathbb{F}_q$, a finite field of size q, using Reed-Solomon codes is to create a message polynomial

$$p(x) = m_0 + m_1 x + \ldots + m_{n-1} x^{n-1}.$$

Let e be an integer, $k = n + e$, and $\alpha \in \mathbb{F}_q^k$ be a primitive element. Next, construct the generator polynomial

$$g(x) = (x - \alpha^0)(x - \alpha^1) \ldots (x - \alpha^{e-1}).$$

Then one finds the remainder, $r(x)$, of $x^e p(x)$ when divided by $g(x)$, which has degree at most $e - 1$. Hence,

$$r(x) = r_0 + r_1 x + \ldots + r_{e-1} x^{e-1}.$$

Our codeword is our original message plus our error correction digits, which are the coefficients of the remainder polynomial. Thus, our codeword is

$$c = (m_0, m_1, \ldots, m_{n-1}, r_0, r_1, \ldots, r_{e-1}) \in \mathbb{F}_q^k.$$

Note, that decoding utilizes the fact that $\alpha^0, \alpha^1, \ldots,$ and α^{e-1} are the roots of the generator polynomial $g(x)$.

Our program creates these error correction codewords in following 4 steps.

(1) Construct the message polynomial. Using the finished data string created in Section 3, we will convert each 8-bit segment to its decimal equivalent integer.

```
messagepoly=[int(line[i:i+8],2) for i in range(0, len(line), 8)]
-----------------------------------------------------------------------
messagepoly=[32, 68, 173, 204, 17, 21, 73, 0, 236, 17, 236, 17, 236, 17, 236, 17]
```

These integers will represent the coefficients of our message polynomial. The exponent for each term is

$$[m_0x^{e+n}, m_1x^{e+(n-1)}, m_2x^{e+(n-2)}, \dots, m_{n-1}x^{e}],$$

where e is the required amount of error correction codewords found in Table A (see Appendix A) and n is the length of messagepoly minus 1. For our example, the result is

$$[32x^{25}, 68x^{24}, 173x^{23}, \dots, 17x^{10}].$$

Note this is the polynomial $x^e p(x)$.

(2) Create a generator polynomial. Note, all arithmetic will be done over the finite field of size 256, which is isomorphic to $\mathbb{Z}_2[x]/(1+x^2+x^3+x^4+x^8)$. The message is broken into 8 bit strings, which represent polynomials in this field. For example, 0000111 represents x^2+x+1.

To add to elements in this field we may add byte-wise modulo 100011101, which represents $(1+x^2+x^3+x^4+x^8)$. Note, each 8 bit string is a binary representation of an integer from 0 to 255, and we can perform addition by converting our binary representations to decimal and performing an XOR. If the result is 256 or larger, then we XOR it with 285, which is the decimal representation of 100011101. To multiply elements in this field it is quicker to perform this as addition of base-2 exponents since $\alpha = 2$ is a primitive element of this field. Note, we may go between the two representations using a log antilog table.

The generator polynomial needed for e error correction codewords is

$$g(x) = (x-\alpha^0)\dots(x-\alpha^{e-1}),$$

where $\alpha = 2$ is the primitive element of this finite field. For example, when $e = 2$, our generator polynomial is

$$g(x) = (\alpha^0 x - \alpha^0) * (\alpha^0 x - \alpha^1)$$

since $\alpha^0 = 1$. To multiply the terms, use exponent addition, so

$$g(x) = (\alpha^{(0+0)}x^{(1+1)}) + (\alpha^{(0+0)}x^{(0+1)}) + (\alpha^{(0+1)}x^{(1+0)}) + (\alpha^{(0+1)}x^{(0+0)}).$$

After combining like-terms, we have

$$g(x) = \alpha^0x^2 + (\alpha^0+\alpha^1)x^1 + \alpha^1x^0.$$

Addition is performed by converting the alphas to their integer representation, then performing an XOR operation. Thus,

$$g(x) = x^2 + (1 \oplus 2)x^1 + 2x^0 = x^2 + 3x^1 + 2x^0.$$

Converted back into alpha notation the generator polynomial for two error correction codewords is

$$g(x) = \alpha^0x^2 + \alpha^{25}x^1 + \alpha^1x^0.$$

To create generator polynomials of higher error correction words, multiply the previous polynomial by the next factor. For example, the 3 error correction codeword polynomial is

$$g'(x) = (\alpha^0x^2 + \alpha^{25}x^1 + \alpha^1x^0) * (x - \alpha^2) = \alpha^0x^3 + \alpha^{198}x^2 + \alpha^{199}x^1 + \alpha^3x^0.$$

In the case of our message "QR CODES," generator polynomial is

$$g(x) = \alpha^0x^{10} + \alpha^{251}x^9 + \alpha^{67}x^8 + \alpha^{46}x^7 + \alpha^{61}x^6 + \alpha^{118}x^5 + \alpha^{70}x^4 + \alpha^{64}x^3 + \alpha^{94}x^2 + \alpha^{32}x^1 + \alpha^{45}x^0.$$

(3) Next, divide the message polynomial by the generator polynomial and find the remainder. Multiply the generator polynomial by a factor of x, so that the first term is the same order as the message polynomial. For the message "QR CODES" we multiply by x^{15}, which results in

$$x^{15}g(x) = \alpha^0x^{25} + \alpha^{251}x^{24} + \alpha^{67}x^{23} + \alpha^{46}x^{22} + \alpha^{61}x^{21} + \alpha^{118}x^{20} + \alpha^{70}x^{19} + \alpha^{64}x^{18} + \alpha^{94}x^{17} + \alpha^{32}x^{16} + \alpha^{45}x^{15}.$$

Then XOR the coefficients of the generator polynomial with the message polynomial. If the polynomials are in alpha notation, then use the power of alpha as the integer coefficient. Each time an XOR is performed, the message polynomial will lose a term.

Repeat this process of performing an XOR with the remainder of the message polynomial n times, where n is equal to the number of terms in the message polynomial. After n iterations of division, the remainder is our error correction polynomial whose coefficients are the error correction codewords. For our example, the remainder is

$$r(x) = \alpha^{230}x^9 + \alpha^{153}x^8 + \alpha^{254}x^7 + \alpha^{136}x^6 + \alpha^{133}x^5 + \alpha^{109}x^4 + \alpha^{196}x^3 + \alpha^{167}x^2 + \alpha^{37}x^1 + \alpha^{239}.$$

Thus, the error correction codewords are

$$[230, 153, 254, 136, 133, 109, 196, 167, 37, 239].$$

(4) Lastly, compile final message. Convert the error correction codewords from the previous step into each integers 8-bit binary equivalent. Our error correction codewords become

$$[11100110, 10011001, 11111110, 10001000, 10000101, 01101101, 11000100, 10100111, 00100101, 11101111].$$

Add this new string on to the end of the binary message string created in the encoding steps in Section 3. This 208-bit string will form our final QR message.

5. Graphical Array

The final component to building a QR code is to determine how to display the data we have encoded. This process can be broken down into four tasks: inserting markers within the graphic, writing the message data to the graphic, data masking, and then filling out format and version information.

The QR code standard uses markers throughout the graphic to help decode the data. The first step will be to insert these markers in the graphic area. First, determine the dimensions needed for the QR graphic based on

$$\text{Dimension} = (((V - 1) * 4) + 21),$$

where V is the version of the QR code. Since our "QR CODES" example is version 1, the dimensions of the graphic will be 21×21 modules.

The first marker to insert is the finder patterns (see Appendix B). The finder patterns will always consist of an outer black square which is 7×7 modules, and another black square within that which is 3×3 modules. Place these in the top left, top right, and bottom left corners of the QR graphic area. The code for creating the dimensions and inserting the upper left finder pattern is shown below.

```
dimension=((version-1)*4)+21
m = numpy.zeros([dimension,dimension])
for i in range(7):
    m[i][0:7]=3
for i in range(1,6):
    m[i][1:6]=0
for i in range(2,5):
    m[i][2:5]=3
```

Next, add the separators (see Appendix B) to the graphic. The separators are the white ring of modules directly outside each finder pattern, and they help the QR decoder quickly distinguish the finder patterns.

Then, add the timing patterns (see Appendix B). This consists of an alternating strip of black modules that run inbetween finder patterns. The timing patterns are used to help the QR decoder determine the dimensions of the QR code.

The last markers to add are the dark module and the reserved data areas. The reserved data areas are strips of modules outside the separators which are designated to store information about the QR code. The strip locations can be seen in Figure 4 (see Appendix B). The dark module is a module that is always left black in the same coordinate, $([4 * V) + 9], 8)$, where V is the version of QR code.

Next, write the message data to the graphic area. Starting from the beginning of our final binary message string, fill in modules in the pattern show in Figure 5 (see Appendix B). A bit value of 1 represents a dark module, and a value of 0 represents a white module. When one of the above markers are reached, skip the occupied modules until an empty one is reached.

After writing the data, we would typically perform data masking. This process is used to reduce the number of hard to read patterns found throughout the data graphic. Due to the robust nature of error correction in QR codes, most QR readers will be able to read any code with any data masking pattern. Due to time constraints, we chose to use only one masking pattern. The masking pattern we chose uses the following logic.

$$\text{if (row + column)} \mod 2 == 0\text{, then switch the data bit at that location in the graphic.}$$

The corresponding code is below.

```
def mask0(m):
    for r in range(dimension):
        for c in range(dimension):
            if m[r][c]==1 or 2:
                if (r+c)%2==0:
                    if m[r][c]==2:
                        m[r][c]=1
                    elif m[r][c]==1:
                        m[r][c]=2
```

The last step is to add the format and version information to the reserved information areas shown in Figure 4 (see Appendix B). First, generate a 15 bit format string. The first 2 bits will be a binary representation of the error correction level shown below in Table 3.

Table 3. Error Correction format string

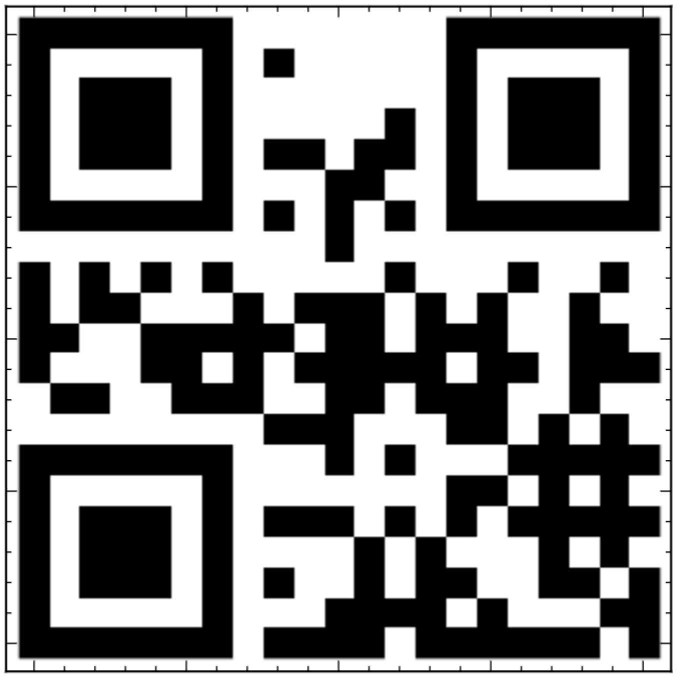
Error Correction Level	**Bits**
L	01
M	00
Q	11
H	10

The next 3 bits will be a binary equivalent of the number corresponding to the mask pattern used. Since we will be using mask 0 exclusively, this string will be '000.' The last 10 bits will be filled with error correction words for the 5 bit string we have created from the error correction and mask pattern. The process for creating these 10 error correction words will be the same as described in Section 4. After adding the error correction bits to the initial format string, XOR the created 15 bit string with '101010000010010.' After completing these steps for our example message "QR CODES", our format string is

$$[1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0].$$

Write this string into the reserved information areas as shown in Figure 6 (see Appendix B). Note, that 14 corresponds to the most significant bit in the format string. After adding the format information, you will have a scannable QR code. The graphical output from the example message "QR CODES" is shown in Figure 2.

Figure 2. Example message QR CODE



6. Future Plans

Our plans for the future of this project are to research and apply advanced forms of cryptography to QR codes. Currently, the only application to support public use of encrypted QR codes is an android app called QR Droid which utilizes private key encryption. This poses security threats for applications that require elevated levels of message protection. We hope to create an encoder that also applies public key encryption to reliably protect the QR code messages. This could have potential applications for small businesses or startups who are seeking to utilize QR codes in their products, but also require a more secure form of encryption.

References

[1] Ling, San, and Chaoping Xing, Coding theory: a first course, Cambridge University Press (2004).

[2] QR code tutorial, `http://www.thonky.com/qr-code-tutorial/introduction`, 2017.

[3] I. S. Reed and G. Solomon, Polynomial codes over certain finite fields, J. Soc. Indust. Appl. Math. **8** (1960), 300–304.

[4] William A. Stein et al. Sage Mathematics Software (Version 5.11), The Sage Development Team, 2014, http://www.sagemath.org.

Appendix A. QR Standards

Table 4 gives the characters allowed with alphanumeric mode and their numeric representation.

Table 4. Alphanumeric Characters

Character	Representation	Character	Representation	Character	Representation
0	0	F	15	U	30
1	1	G	16	V	31
2	2	H	17	W	32
3	3	I	18	X	33
4	4	J	19	Y	34
5	5	K	20	Z	35
6	6	L	21	(space)	36
7	7	M	22	$	37
8	8	N	23	%	38
9	9	O	24	*	39
A	10	P	25	+	40
B	11	Q	26	-	41
C	12	R	27	.	42
D	13	S	28	/	43
E	14	T	29	:	44

The following table lists the number of error correction codewords that will need to be generated based on the version and error correction level of the QR code.

Table 5. Code Word Requirements

Version and Error Correction Level	Total Data Codewords needed	Total Error Correction Codewords needed	Total number of data bits needed
1-L	19	7	152
1-M	16	10	128
1-Q	13	13	104
1-H	9	17	72
2-L	34	10	272
2-M	28	16	224
2-Q	22	22	176
2-H	16	28	128
3-L	55	15	440
3-M	44	26	352
3-Q	34	18	272
3-H	26	20	208

Appendix B. QR Code Markers

All of the following figures are from [2].

Figure 3. QR markers used to assist in the decoding process.

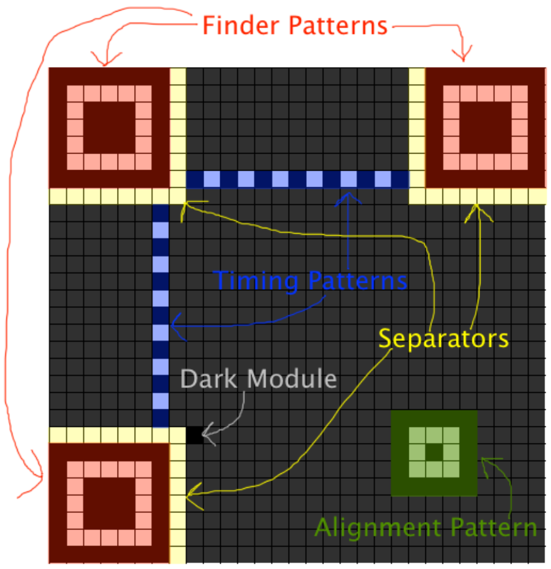


Figure 4. Reserved information areas marked in blue.

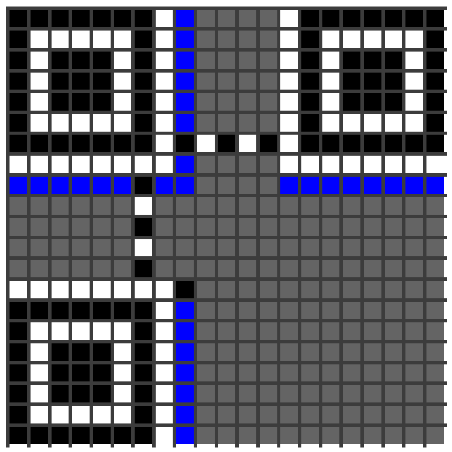


FIGURE 5. Pattern in which data is written to modules.

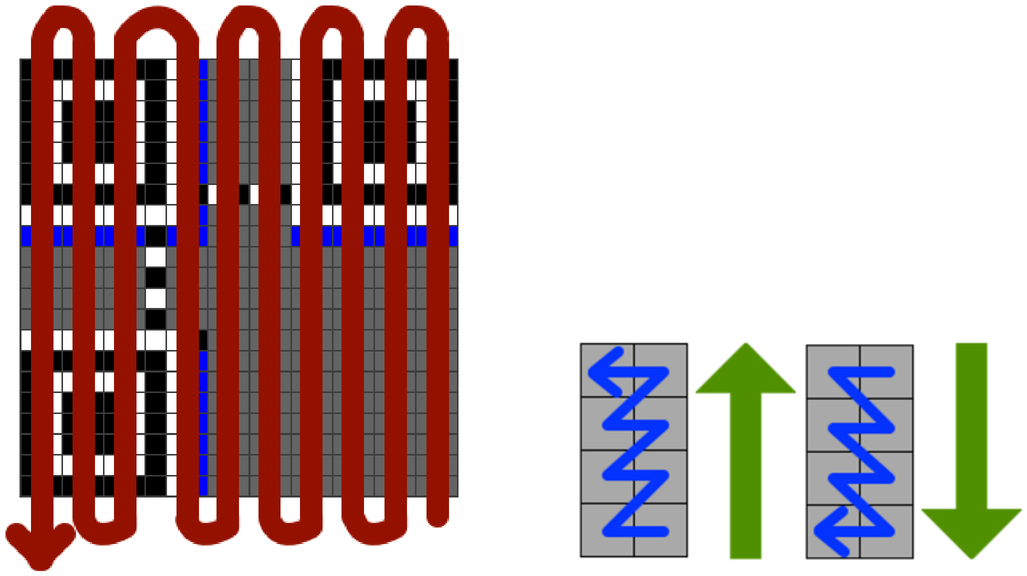
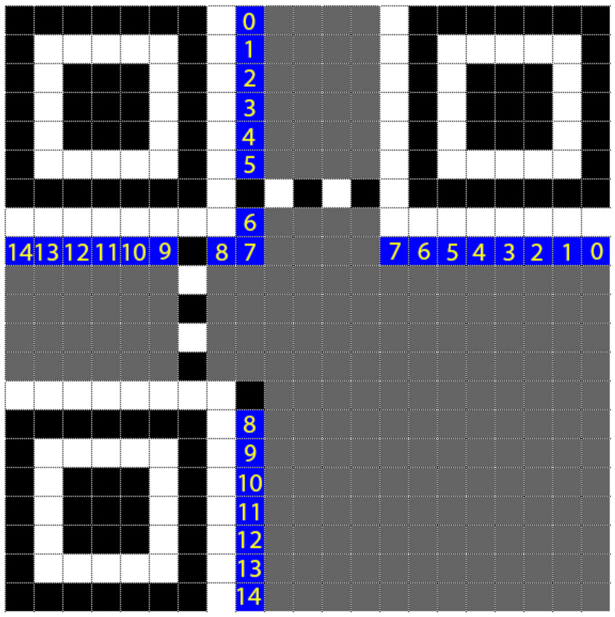


FIGURE 6. Order in which the format string is written to the reserved information areas.



Appendix C. Program

The following shows the entire code written to encode a message and output a QR code in SAGE.

```
############ Declarations ##############################################

input='QR CODES'
EC = 'M'

######### Version Selector(partial) ##########################################

mode = '0010'
l = len(input)
charcount = '{0:09b}'.format(l)
if EC == 'L':
    bits=[0,1]
    if l<25:
        version=1
        datacap=152
        groups=1
        cwcount=19
        eccw=7
        gpoly=[0,87,229,146,149,238,102,21]
    elif 25<=l<47:
        version=2
        datacap=272
        groups=1
        cwcount=34
        eccw=10
        gpoly=[0,251,67,46,61,118,70,64,94,32,45]
    elif 47<=l<77:
        version=3
        datacap=440
        groups=1
        cwcount=55
        eccw=15
        gpoly=[0,8,183,61,91,202,37,51,58,58,237,140,124,5,99,105]
    elif 77<=l<114:
        version=4
        datacap=640
        groups=1
        cwcount=80
        eccw=20
        gpoly=[0,17,60,79,50,61,163,26,187,202,180,221,225,83,239,156,164,212,212,188,190]
    elif 114<=l<154:
        version=5
        datacap=864
        groups=1
        cwcount=108
        eccw=26
        gpoly=[0,173,125,158,2,103,182,118,17,145,201,111,28,165,53,161,21,245,142,13,102,48,227,153,145
elif EC == 'M':
    bits=[0,0]
    if l<20:
        version=1
        datacap=128
        groups=1
        cwcount=16
        eccw=10
        gpoly=[0,251,67,46,61,118,70,64,94,32,45]
    elif 20<=l<38:
        version=2
        datacap=224
        groups=1
        cwcount=28
        eccw=16
        gpoly=[0,120,104,107,109,102,161,76,3,91,191,147,169,182,194,225,120]
    elif 38<=l<61:
        version=3
        datacap=352
        groups=1
        cwcount=44
        eccw=26
        gpoly=[0,173,125,158,2,103,182,118,17,145,201,111,28,165,53,161,21,245,142,13,102,48,227,153,145
    elif 61<=l<90:
        version=4
        datacap=512
        groups=1
        cwcount=64
        eccw=18
        gpoly=[0,215,234,158,94,184,97,118,170,79,187,152,148,252,179,5,98,96,153]
    elif 90<=l<122:
        version=5
        datacap=688
        groups=1
        cwcount=86
        eccw=24
        gpoly=[24,229,121,135,48,211,117,251,126,159,180,169,152,192,226,228,218,111,0,117,232,87,96,227
 elif EC == ’Q’:
    bits=[1,1]
    if l<16:
        version=1
        datacap=104
        groups=1
        cwcount=13
        eccw=13
        gpoly=[13,74,152,176,100,86,100,106,104,130,218,206,140,78]
    elif 16<=l<29:
        version=2
        datacap=176
        groups=1
        cwcount=22
        eccw=22
        gpoly=[0,210,171,247,242,93,230,14,109,221,53,200,74,8,172,98,80,219,134,160,105,165,231]
    elif 29<=l<47:
        version=3
        datacap=272
        groups=1
        cwcount=34
        eccw=18
        gpoly=[0,215,234,158,94,184,97,118,170,79,187,152,148,252,179,5,98,96,153]
    elif 47<=l<67:
        version=4
        datacap=384
        groups=1
        cwcount=48
        eccw=26
        gpoly=[0,173,125,158,2,103,182,118,17,145,201,111,28,165,53,161,21,245,142,13,102,48,227,153,145
    elif 67<=l<87:
        version=5
        datacap=496
        groups=2
        g1=15
        g2=16
        cwcount=62
        eccw=18
        gpoly=[0,215,234,158,94,184,97,118,170,79,187,152,148,252,179,5,98,96,153]
elif EC == 'H':
    bits=[1,0]
    if l<10:
        version=1
        datacap=72
        groups=1
        cwcount=9
        eccw=17
    elif 10<=l<20:
        version=2
        datacap=128
        groups=1
        cwcount=16
        eccw=28
    elif 20<=l<35:
        version=3
        datacap=208
        groups=1
        cwcount=26
        eccw=22
        gpoly=[0,210,171,247,242,93,230,14,109,221,53,200,74,8,172,98,80,219,134,160,105,165,231]
    elif 35<=l<50:
        version=4
        datacap=288
        groups=1
        cwcount=36
        eccw=16
        gpoly=[0,120,104,107,109,102,161,76,3,91,191,147,169,182,194,225,120]
    elif 50<=l<64:
        version=5
        datacap=368
        groups=2
        g1=11
        g2=12
        cwcount=46
        eccw=22
        gpoly=[0,210,171,247,242,93,230,14,109,221,53,200,74,8,172,98,80,219,134,160,105,165,231]

if version==1:
    remainder=0
else:
    remainder=7

print("version="+str(version))
print("datacap="+str(datacap))
print "groups=",groups

######### String Split Function ################################################

def split(fn):
    while fn:
        yield fn[:2]
        fn = fn[2:]

######## Loop to convert characters to binary encoded data ##############################

a=list(split(input))
length=len(a)
a
encoded=’’

for i in range(0,(length)):
    a[i]=a[i].replace(’0’,’00’).replace(’1’,’01’).replace(’2’,’02’).replace(’3’,’03’).replace(’4’,’04’)
   a[i]=a[i].replace(’5’,’05’).replace(’6’,’06’).replace(’7’,’07’).replace(’8’,’08’).replace(’9’,’09’)
    a[i]=a[i].replace(’B’,’11’).replace(’C’,’12’).replace(’D’,’13’).replace(’E’,’14’).replace(’F’,’15’)
   a[i]=a[i].replace(’G’,’16’).replace(’H’,’17’).replace(’I’,’18’).replace(’J’,’19’).replace(’K’,’20’)
    a[i]=a[i].replace(’M’,’22’).replace(’N’,’23’).replace(’O’,’24’).replace(’P’,’25’).replace(’Q’,’26’)
   a[i]=a[i].replace(’R’,’27’).replace(’S’,’28’).replace(’T’,’29’).replace(’U’,’30’).replace(’V’,’31’)
    a[i]=a[i].replace(’X’,’33’).replace(’Y’,’34’).replace(’Z’,’35’).replace(’
        ’,’36’).replace(’$’,’37’)
   a[i]=a[i].replace(’%’,’38’).replace(’*’,’39’).replace(’+’,’40’).replace(’-’,’41’).replace(’.’,’42’)
   a[i]=a[i].replace(’A’,’10’).replace(’L’,’21’).replace(’W’,’32’).replace(’/’,’43’).replace(’:’,’44’)

    if len(a[i])==4:
        a[i]=(int((a[i])[:2])*45)+(int((a[i])[2:5]))
        a[i]=’{0:011b}’.format(a[i])
    elif len(a[i])==2:
        a[i]=’{0:06b}’.format(int(a[i]))
    encoded=encoded+a[i]

print "encoded=",encoded
datastring2=(mode+charcount+encoded)
dist=(datacap-len(datastring2))


############## Pad bits for short string #########################################

if dist>=4:
    terminator="0000"
elif dist==3:
    terminator="000"
elif dist==2:
    terminator="00"
elif dist==1:
    terminator="0"
elif dist==0:
    terminator=""

datastring=(mode+charcount+encoded+terminator)
dist2=(datacap-len(datastring))
l2=len(datastring)

pad=(8-(l2%8))
pad="0"*pad
datastring=datastring+pad
extrapad=(datacap-len(datastring))/8

for i in range(1,extrapad+1):
        if i%2==1:
            datastring=datastring+"11101100"
        elif i%2==0:
            datastring=datastring+"00010001"

print "datastring=",datastring

############## Creating a log antilog table #########################

exp2int = [1];
int2exp = [i for i in range(256)];


for i in range(1, 256):
    b = exp2int[i - 1]*2;
    if b >= 256: #XOR if result is larger than or equal to 256. Use ^^ to XOR two
        integers
        b = b^^285;
    exp2int.append(b);
    int2exp[b] = i;
```

```
int2exp[0] = []; #setting zero blank
int2exp[1] = 0; #1 comes up at 255 again

############### Message Polynomial ###################################

line = datastring
mpoly=[int(line[i:i+8],2) for i in range(0, len(line), 8)]
print"mpoly=",mpoly

############# Long Division ############################

import numpy
print "generator polynomial=",gpoly
print "message polynomial=",mpoly

def longdivision(steps,mpoly,gpoly):
    for i in range(0,steps):
        gap=len(mpoly)-len(gpoly)
        m=mpoly[0]
        m=int2exp[m]
        if gap>0:
            newgpoly=[exp2int[(g+m)%255] for g in gpoly]+[0]*gap
        else:
            newgpoly=[exp2int[(g+m)%255] for g in gpoly]
        blank=[]
        if gap<0:
            mpoly=mpoly+[0]*abs(gap)
        for i in range(0,len(newgpoly)):
            b=[(mpoly[i]^^newgpoly[i])]
            blank=blank+b
        mpoly=numpy.trim_zeros(blank,trim='f')
    return mpoly

############### interleaving ###############################

if groups==1:
    steps=len(mpoly)
    ecwords=longdivision(steps,mpoly,gpoly)
    print "ecwords=",ecwords
    message=mpoly+ecwords
    message=['{0:08b}'.format(i) for i in message]+[0]*remainder
    blank=''
    for i in message:
        blank=blank+str(i)
    message=blank
    print"full message=",message

elif groups==2:
    b1=mpoly[0:g1]
    b2=mpoly[(g1):(2*g1)]
    b3=mpoly[(2*g1):(2*g1+g2)]
    b4=mpoly[(2*g1+g2):(2*g1+2*g2)]
    ec1=longdivision(g1,b1,gpoly)
    ec2=longdivision(g1,b2,gpoly)
    ec3=longdivision(g2,b3,gpoly)
    ec4=longdivision(g2,b4,gpoly)
    print "b1=",b1
    print "b2=",b2
    print "b3=",b3
    print "b4=",b4
    print "ec1=",ec1
    print "ec2=",ec2
    print "ec3=",ec3
    print "ec4=",ec4

    blank=[]
    for i in range(g1):
        blank=blank+[b1[i]]+[b2[i]]+[b3[i]]+[b4[i]]
    itldata=blank+[b3[-1]]+[b4[-1]]

    blank=[]
    for i in range(len(ec1)):
        blank=blank+[ec1[i]]+[ec2[i]]+[ec3[i]]+[ec4[i]]
    itlecw=blank

    message=itldata+itlecw
    message=['{0:08b}'.format(i) for i in message]+[0]*remainder
    blank=''
    for i in message:
        blank=blank+str(i)
    message=blank
    print"full message=",message

len(message)
################## Graphical Array ##############################

from sage.plot.matrix_plot import matrix_plot
import numpy

dimension=((version-1)*4)+21
print "dimension = ",dimension,"x",dimension

m = numpy.zeros([dimension,dimension])

############ Finder & Timing Pattern #################

for i in range(7):
    m[i][0:7]=3
    m[i][-7:dimension]=3
for i in range(-1,-8,-1):
    m[i][0:7]=3
```

```
for i in range(1,6):
    m[i][1:6]=0
    m[i][-6:dimension-1]=0
for i in range(-2,-7,-1):
    m[i][1:6]=0

for i in range(2,5):
    m[i][2:5]=3
    m[i][-5:dimension-2]=3
for i in range(-3,-6,-1):
    m[i][2:5]=3

for i in range(8,dimension-8,2):
    m[i][6]=3
    m[6][i]=3

######### Data graphic ############

shift=1
direction=1
l=0
u=0
g=1
cap1=(dimension-9)*2
cap2=(dimension)*2
cap1,cap2
count=0
counter=0
count3=0
maxcount=((dimension-17)/2)+4
maxcount3=(dimension-17)*2
maxcol=((dimension-7)/2)-1

for pos,i in enumerate(message):
    if pos>((dimension-9)*8):
        counter=counter+1
    if pos>0 and (pos)%cap1==0 and count<4:
        count=count+1
        shift=shift+2
        direction=-direction
        g=-g
        u=u+g
    elif count>=4 and count<maxcount and counter%cap2==0:
        count=count+1
        shift=shift+2
        direction=-direction
        g=-g
        u=u+g
    if (dimension-1-u)==6:
            u=u+(1*direction)
            counter=counter+2
    if count==maxcol and counter%cap2==0:
        u=u+8
        counter=counter+14
        count=count+1
    if count3==maxcount3:
        shift=shift+3
        direction=-direction
        g=-g
        u=u+g
    elif count3>maxcount3 and count3%maxcount3==0:
        shift=shift+2
        direction=-direction
        g=-g
        u=u+g
    if count>=maxcol:
        count3=count3+1
    m[dimension-1-u][dimension-shift-l]=int(i)+1
    l=(l+1)%2
    u=u+(pos%2)*direction

######### Masking ########################

masknum=0
maskbin=[0,0,0]
#maskbin=[0,0,1]
def mask0(m):
    for r in range(dimension):
        for c in range(dimension):
            if m[r][c]==1 or 2:
                if (r+c)%2==0:
                    if m[r][c]==2:
                        m[r][c]=1
                    elif m[r][c]==1:
                        m[r][c]=2
def mask1(m):
    for r in range(dimension):
        for c in range(dimension):
            if m[r][c]==1 or 2:
                if r%2==0:
                    if m[r][c]==2:
                        m[r][c]=1
                    elif m[r][c]==1:
                        m[r][c]=2
mask0(m)
#mask1(m)

########## Information Bits ###############

gpoly=[1,0,1,0,0,1,1,0,1,1,1]
bitstring=bits+maskbin
```

```
infostring=bitstring+[0]*10
infostring=numpy.trim_zeros(infostring[:5],trim='f')+infostring[5:]
print "infostring=",infostring

if len(infostring)==10:
    ecstring=infostring+[0]

while len(infostring)>=11:
    gap=len(infostring)-len(gpoly)
    if gap!=0:
        gpoly=gpoly+[0]*gap
    pos=0
    for i,j in zip(infostring,gpoly):
        infostring[pos]=i^^j
        pos=pos+1
    infostring=numpy.trim_zeros(infostring,trim='f')
    ecstring=infostring
    print"ecstring=",ecstring

if len(ecstring)<10:
    ecstring=(10-len(ecstring))*[0]+ecstring
print"final ecstring=",ecstring

ecstring=bitstring+ecstring
qrspec=[1,0,1,0,1,0,0,0,0,0,1,0,0,1,0]
pos=0
for i,j in zip(ecstring,qrspec):
    qrspec[pos]=(i^^j)*3
    pos=pos+1
formatstring=qrspec
print"format string=",formatstring

if EC == 'L':
    if masknum==0:
        formatstring=[3,3,3,0,3,3,3,3,3,0,0,0,3,0,0]
    elif masknum==1:
        formatstring=[3,3,3,0,0,3,0,3,3,3,3,0,0,3,3]
elif EC == 'M':
    if masknum==0:
        formatstring=[3,0,3,0,3,0,0,0,0,0,3,0,0,3,0]
    elif masknum==1:
        formatstring=[3,0,3,0,0,0,3,0,0,3,0,0,3,0,3]
elif EC == 'Q':
    if masknum==0:
        formatstring=[0,3,3,0,3,0,3,0,3,0,3,3,3,3,3]
    elif masknum==1:
        formatstring=[0,3,3,0,0,0,0,0,3,3,0,3,0,0,0]
elif EC == 'H':
    if masknum==0:
        formatstring=[0,0,3,0,3,3,0,3,0,0,0,3,0,0,3]
    elif masknum==1:
       formatstring=[0,0,3,0,0,3,3,3,0,3,3,3,3,3,0]
print"formatstring2=",formatstring

for pos,i in enumerate(range(-8,0,1)):
    m[8][i]=formatstring[(pos+7)]
for pos,i in enumerate(range(-1,-8,-1)):
    m[i][8]=formatstring[pos]
for pos,i in enumerate([0,1,2,3,4,5,7,8]):
    m[8][i]=formatstring[pos]
for pos,i in enumerate([8,7,5,4,3,2,1,0]):
    m[i][8]=formatstring[(pos+7)]
m[-8][8]=3

for i in range(dimension):
    for j in range(dimension):
        if m[i][j]==1:
            m[i][j]=0
        if m[i][j]==2:
            m[i][j]=3

M = matrix_plot(m,vmin=None, frame=True, axes=False, colorbar=False, aspect_ratio=1,
    subdivisions=False, cmap='Greys', ticks_integer=true, marker='.', vmax=None,
    norm=None, subdivision_boundaries=None, subdivision_style=None,
    colorbar_orientation='vertical', colorbar_format=None)
M
```