The Mathematics of QR Codes

Adam S. Downs, Neil P. Sigmon
asdowns@radford.edu, npsigmon@radford.edu
Department of Mathematics
Radford University
Radford, VA, USA

Richard E. Klima
klimare@appstate.edu
Department of Mathematical Sciences
Appalachian State University
Boone, NC, USA

Abstract: *Quick Response (QR) codes are two-dimensional bar codes that have become a common medium for easily accessing information such as URLs, phone numbers, and small amounts of text. The creation of QR codes requires placing data in a prescribed format so that it can be displayed on a surface and then detected by common QR code scanner software. Like other physical means of storing data, QR codes are prone to errors when their data is interpreted in a digital format. Reed-Solomon codes provide a mechanism for ensuring that QR code scanners can reliably process information when errors occur. This involves sending information in the form of polynomial coefficients using finite field arithmetic. Utilizing Reed-Solomon codes allows logos and emblems to be embedded within QR codes to advertise their purpose. In this paper we will present how QR codes are constructed, and how Reed-Solomon codes are incorporated into them to provide error correction. To assist in demonstrating this, technology involving Maplets will be used.*

1 Introduction

Since their 1994 invention by the DENSO Corporation in Japan, QR codes have provided a widely used method for quickly accessing information. Originally, QR codes were used for parts manufacturing inventory tracking, but they can now be found on advertisements, web pages, business cards, any many other mediums. QR codes are notable for their long-term stability and higher capacity for storage than other bar codes. However, like other physical means of storing data, QR codes are prone to errors. Reed-Solomon codes provide a mechanism for ensuring that QR code scanners can reliably process information when errors occur.

In this paper we will give an overview of how QR codes are created, and how data is integrated into their construction. As part of this, we will describe the basics of how Reed-Solomon codes work, and how Reed-Solomon codes are incorporated into QR codes to increase the likelihood that they are able to transmit data reliably. This information provides a means for demonstrating a hands-on method for the use of mathematics in a practical real-life application.

2 Data Representation and Finite Fields

QR codes encode text represented as binary numbers. The QR code standard allows for text strings to be created using four encoding modes: numeric, alphanumeric, byte, and kanji. In this paper we will describe the byte mode. Descriptions of the other three modes can be found in [5].

The byte mode of encoding text for QR codes utilizes the ASCII character set. A list of the ASCII character set and the correspondences between characters and decimal numbers in ASCII for the printable characters on a modern keyboard is shown in Table 1.

Char	Num	Char	Num	Char	Num	Char	Num	Char	Num
(space)	32	3	51	F	70	Y	89	l	108
!	33	4	52	G	71	Z	90	m	109
"	34	5	53	H	72	[	91	n	110
#	35	6	54	I	73	\	92	o	111
$	36	7	55	J	74	]	93	p	112
%	37	8	56	K	75	^	94	q	113
&	38	9	57	L	76	_	95	r	114
’	39	:	58	M	77	‘	96	s	115
(	40	;	59	N	78	a	97	t	116
)	41	<	60	O	79	b	98	u	117
*	42	=	61	P	80	c	99	v	118
+	43	>	62	Q	81	d	100	w	119
,	44	?	63	R	82	e	101	x	120
-	45	@	64	S	83	f	102	y	121
.	46	A	65	T	84	g	103	z	122
/	47	B	66	U	85	h	104	{	123
0	48	C	67	V	86	i	105	|	124
1	49	D	68	W	87	j	106	}	125
2	50	E	69	X	88	k	107	~	126

Table 1: Correspondences between characters and decimal numbers in ASCII.

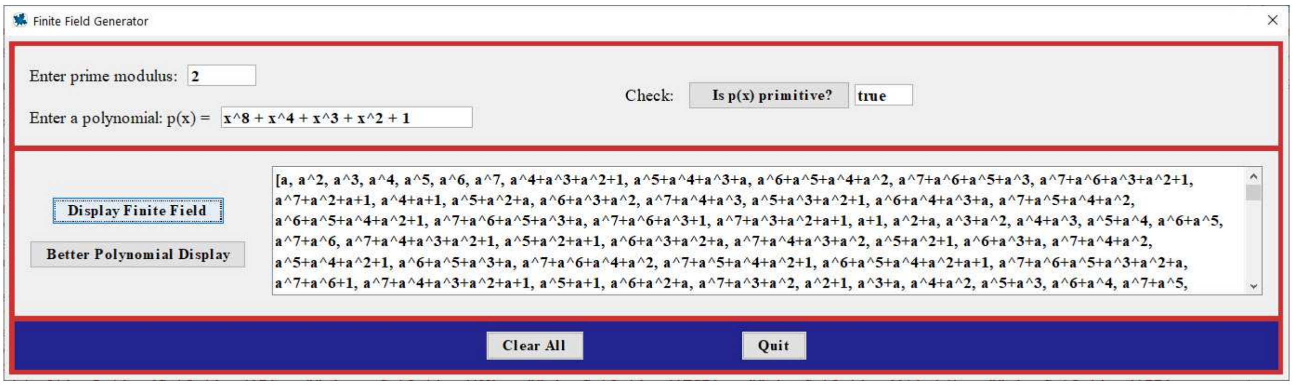
Each character in ASCII corresponds to a decimal number, which can then be expressed as a string of eight binary digits, or a *byte*. For example, the character `M` corresponds to the decimal number 77, which can be expressed in binary as 1001101, or the byte 01001101. This byte can then be represented as a polynomial, by using these digits in reverse order as coefficients on powers of the variable a, with terms of increasing degree. For example, the byte 01001101 can be represented as the polynomial $1+0a+1a^2+1a^3+0a^4+0a^5+1a^6+0a^7 = 1+a^2+a^3+a^6$. Similarly, every character in ASCII can be expressed as a byte, and as a polynomial in a of maximum degree 7. Since all characters can be expressed as binary numbers, all computations that follow will be done using modulo 2 arithmetic. This means that all numerical computations will result in a 0 or 1. Specifically, if a result gives a number that is divisible by 2, it can be reduced to 0, and if it gives a number that is not divisible by 2, it can be reduced to 1. For example, the number 6 with modulo 2 arithmetic reduces to 0; that is, $6 \bmod 2 = 0$. The number 7 with modulo 2 arithmetic though reduces to 1; that is, $7 \bmod 2 = 1$. This can also work with coefficients of polynomials. For example, with modulo 2 arithmetic, the polynomial $7a^3+4a^2-a+2$ reduces to $(7a^3+4a^2-a+2) \bmod 2 = 1a^3+0a^2+1a+0 = a^3+a$.

A particular finite field provides the method with QR codes for performing operations on polynomials that represent information. The finite field used with QR codes contains $2^8 = 256$ elements, and uses the polynomial $p(x) = x^8 + x^4 + x^3 + x^2 + 1$ to generate the 255 of these field elements which are nonzero. These 255 nonzero field elements can be expressed as all nonzero polynomials in the variable a of maximum degree seven and with coefficients that are all either 0 or 1, which can be generated as follows. Let $x = a$ be a root of $p(x)$, called a *primitive* element in the field. To generate all of the nonzero elements in the field, we can raise a to consecutive integer powers, from the first through the 255th. That is, the 255 nonzero field elements are $\{a, a^2, \ldots, a^{254}, a^{255}\}$, the last of which will equal 1 (an explanation for which can be found in [3]). The finite field is then completed by including 0 with this set. To represent the 255 nonzero field elements as polynomials in a of maximum degree seven and with coefficients that are all either 0 or 1, we can use the fact that a is a root of $p(x)$. As such, after the first seven nonzero field elements have been formed (which are simply $a, a^2, \ldots, a^7$) and a^8 is reached, since a is a root of $p(x)$, then $p(a) = a^8 + a^4 + a^3 + a^2 + 1 = 0$. Solving for a^8 and reducing the coefficients with modulo 2 arithmetic gives $a^8 = -a^4 - a^3 - a^2 - 1 = a^4 + a^3 + a^2 + 1$. That is, $a^4 + a^3 + a^2 + 1$ is the polynomial of maximum degree seven that can be used to represent a^8 in the field. Subsequent nonzero field elements can be generated similarly.

$$\begin{array}{rcl} a^9 & = & aa^8 & = & a^5 + a^4 + a^3 + a \\ a^{10} & = & aa^9 & = & a^6 + a^5 + a^4 + a^2 \\ a^{11} & = & aa^{10} & = & a^7 + a^6 + a^5 + a^3 \\ a^{12} & = & aa^{11} & = & a^8 + a^7 + a^6 + a^4 = a^4 + a^3 + a^2 + 1 + a^7 + a^6 + a^4 = a^7 + a^6 + a^3 + a^2 + 1 \\ \vdots & & & & \end{array}$$

This process could be continued until the last nonzero element, $a^{255} = 1$, were reached. Since doing this would generate all 255 nonzero polynomials in the variable a of maximum degree seven and with coefficients that are all either 0 or 1, then $p(x)$ is also characterized as a *primitive* polynomial.

We will now demonstrate a Maplet[1] written by the authors that can be used to generate the elements in this finite field, with the nonzero elements expressed as polynomials in a of maximum degree seven and with coefficients that are all either 0 or 1. Specifically, Figure 1 shows how the Maplet **FiniteFieldGenerator** can be used for this purpose.


Finite Field Generator

Enter prime modulus: 2

Enter a polynomial: p(x) = x^8 + x^4 + x^3 + x^2 + 1

Check: Is p(x) primitive? true

```
[a, a^2, a^3, a^4, a^5, a^6, a^7, a^4+a^3+a^2+1, a^5+a^4+a^3+a, a^6+a^5+a^4+a^2, a^7+a^6+a^5+a^3, a^7+a^6+a^3+a^2+1,
a^7+a^2+a+1, a^4+a+1, a^5+a^2+a, a^6+a^3+a^2, a^7+a^4+a^3, a^5+a^3+a^2+1, a^6+a^4+a^3+a, a^7+a^5+a^4+a^2,
a^6+a^5+a^4+a^2+1, a^7+a^6+a^5+a^3+a, a^7+a^6+a^3+1, a^7+a^3+a^2+a+1, a+1, a^2+a, a^3+a^2, a^4+a^3, a^5+a^4, a^6+a^5,
a^7+a^6, a^7+a^4+a^3+a^2+1, a^5+a^2+a+1, a^6+a^3+a^2+a, a^7+a^4+a^3+a^2, a^5+a^2+1, a^6+a^3+a, a^7+a^4+a^2,
a^5+a^4+a^2+1, a^6+a^5+a^3+a, a^7+a^6+a^4+a^2, a^7+a^5+a^4+a^2+1, a^6+a^5+a^4+a^2+a+1, a^7+a^6+a^5+a^3+a^2+a,
a^7+a^6+1, a^7+a^4+a^3+a^2+a+1, a^5+a+1, a^6+a^2+a, a^7+a^3+a^2, a^2+1, a^3+a, a^4+a^2, a^5+a^3, a^6+a^4, a^7+a^5,
```

Figure 1: Finite field elements for primitive polynomial $p(x) = x^8 + x^4 + x^3 + x^2 + 1$.

[1]A Maplet is like an applet, but uses (and requires) the engine of the computer algebra system Maple, and is written using Maple functions and syntax.

A QR code formats a message with specifications, and places the result on a two-dimensional grid made up of pixels. QR codes can have grid sizes ranging from the smallest for version 1 at size 21×21, to the largest (as of this writing) for version 40 at size 177×177. Each new QR code version after the first uses a grid size with 4 more rows and 4 more columns than the grid size used in the previous version. The size of the grid for a particular message depends on the number of characters in the message and the error correction level specified for the code. Error correction is included to help ensure that messages can still be read correctly even if part of them are unreadable. Reed-Solomon codes, as described in section 3, are used for this purpose.

To demonstrate how the characters in a message can be formatted for a QR code, consider a QR code that when scanned, portrays the message `https://atcm.mathandtech.org/`. To begin, an error correction level is selected. There are four possible error correction levels with QR codes: L, which allows the information to be accessible if up to 7% is unreadable; M, which provides up to 15% error correction; Q, which provides up to 25% error correction; and H, which provides up to 30% error correction. For our example, we will use error correction level H.

The next step is to determine the smallest possible grid size that can be used to encode the message with the specified error correction level. Table 2 shows the character capacities for each QR code version in byte mode and with error correction level H. A table showing the character capacities for each QR code version in all modes and with all error correction levels can be found in [5].

Version	Char Cap	Version	Char Cap	Version	Char Cap	Version	Char Cap
1	7	11	137	21	403	31	790
2	14	12	155	22	439	32	842
3	24	13	177	23	461	33	898
4	34	14	194	24	511	34	958
5	44	15	220	25	535	35	983
6	58	16	250	26	593	36	1051
7	64	17	280	27	625	37	1093
8	84	18	310	28	658	38	1139
9	98	19	338	29	698	39	1219
10	118	20	382	30	742	40	1273

Table 2: QR code character capacities in byte mode and with error correction level H.

Since `https://atcm.mathandtech.org/` contains 29 characters, the lowest QR code version that could use to encode it in byte mode and with error correction level H is version 4, with a 33×33 grid.

Next, information that will be encoded into the grid is represented by a string of binary digits. The first four digits are the *mode indicator*, which for byte mode are 0100. The four mode indicator digits for all mode types can be found in [5]. A *character count indicator* is then included, which gives the number of characters in the message. Since `https://atcm.mathandtech.org/` contains 29 characters, and the decimal number 29 is 11101 in binary, these digits are included next, although padded with three 0s at the start to have full length 8, as is required for the character count indicator for all QR code versions 1–9. That is, the character count indicator for our message would actually be included as 00011101. QR code versions 10–40 also require character count indicators of a specified length, although this required length is 16 bits rather than 8. Next, the characters in the actual message are converted into their ASCII code decimal representations, which are then converted into binary and themselves each padded with 0s at the start to have full length 8. These bytes are then included in order after the character count indicator. Using Table 1, we see the characters in `https://atcm.mathandtech.org/` correspond to the following ASCII decimal representations.

```
104  116  116  112  115   58   47   47   97  116   99  109   46  109  97
116  104   97  110  100  116  101   99  104   46  111  114  103   47
```

These decimal representations can then be converted to binary and padded to give the following bytes.

```
01101000  01110100  01110100  01110000  01110011  00111010  00101111
00101111  01100001  01110100  01100011  01101101  00101110  01101101
01100001  01110100  01101000  01100001  01101110  01100100  01110100
01100101  01100011  01101000  00101110  01101111  01110010  01100111
00101111
```

For each version and error correction level, QR codes require that binary strings completely fill the *total capacity* of the code. For codes of version 4 and with error correction level H, the total capacity is 288 bits. As such, after obtaining the bits for a mode indicator, character count indicator, and message, it is almost always necessary to add bits, in particular, some 0s and some padding bytes. First, a *terminator* would be included at the end of the binary string. If the string were four or more bits shorter than the total capacity, then the terminator would consist of exactly four 0s. If the bit string were fewer than four bits shorter than the total capacity, then the terminator would consist of only the number of 0s needed to reach the total capacity. Since in our example, the mode indicator, character count indicator, and message give a total of 244 bits, the terminator 0000 would be included, raising the total length of the binary string to 248 bits.

If the length of a string with the terminator is not a multiple of 8, then additional 0s would be included to bring the length up to a multiple of 8. In our example, since 248 is a multiple of 8, additional 0s are not required. Since 248 is short of the 288-bit total capacity of a code of version 4 and with error correction level H though, the string must still be padded to reach 288 bits. For this, bytes alternating between 11101100 and 00010001, representing 236 and 17 in decimal, respectively, are included until the total capacity is reached. For our example, since we need 40 additional bits to extend our 248-bit string to the required 288 bits, we include 11101100 00010001 11101100 00010001 11101100 to complete the data. A final list of the 288 bits for our example is shown in Table 3.

Mode	**Char Count**	**Message**	**Terminator**	**Pad Bytes**
0100	00011101	01101000 01110100 01110100 01110000 01110011 00111010 00101111 00101111 01100001 01110100 01100011 01101101 00101110 01101101 01100001 01110100 01101000 01100001 01101110 01100100 01110100 01100101 01100011 01101000 00101110 01101111 01110010 01100111 00101111	0000	11101100 00010001 11101100 00010001 11101100

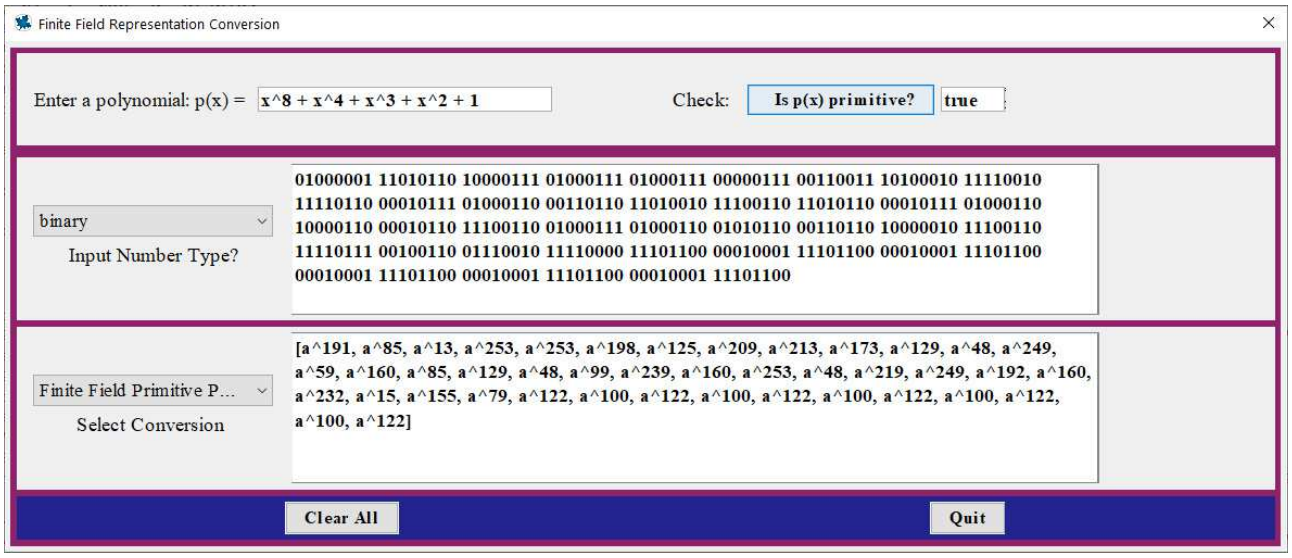
Table 3: QR code example for the message `https://atcm.mathandtech.org/`.

Regrouping these 288 bits into blocks of 8 bits each gives the following full data binary string for our example.

```
01000001  11010110  10000111  01000111  01000111  00000111  00110011
10100010  11110010  11110110  00010111  01000110  00110110  11010010
11100110  11010110  00010111  01000110  10000110  00010110  11100110
01000111  01000110  01010110  00110110  10000010  11100110  11110111
00100110  01110010  11110000  11101100  00010001  11101100  00010001
11101100
```

To work with this sequence of bytes mathematically, we convert each byte into its representation as a polynomial and then as a power of a in the finite field used with QR codes. For example, the byte 11010110 has polynomial representation $a^7 + a^6 + a^4 + a^2 + a$, which turns out to be a^{85}.

We will now demonstrate a Maplet written by the authors that can be used to convert bytes into their corresponding powers of a in the finite field used with QR codes. Specifically, Figure 2 shows how the Maplet **FFConversion** can be used for this purpose.



Enter a polynomial: p(x) = x^8 + x^4 + x^3 + x^2 + 1

true

binary

```
01000001 11010110 10000111 01000111 01000111 00000111 00110011 10100010 11110010
11110110 00010111 01000110 00110110 11010010 11100110 11010110 00010111 01000110
10000110 00010110 11100110 01000111 01000110 01010110 00110110 10000010 11100110
11110111 00100110 01110010 11110000 11101100 00010001 11101100 00010001 11101100
00010001 11101100 00010001 11101100 00010001 11101100
```

Finite Field Primitive P...

```
[a^191, a^85, a^13, a^253, a^253, a^198, a^125, a^209, a^213, a^173, a^129, a^48, a^249,
a^59, a^160, a^85, a^129, a^48, a^99, a^239, a^160, a^253, a^48, a^219, a^249, a^192, a^160,
a^232, a^15, a^155, a^79, a^122, a^100, a^122, a^100, a^122, a^100, a^122, a^100, a^122,
a^100, a^122]
```

Figure 2: Primitive element power representation of data.

Summarizing this result, the full data binary string for our example can represented as the following string of powers of a.

$$\begin{array}{ccccccccccccc}
a^{191} & a^{85} & a^{13} & a^{253} & a^{253} & a^{198} & a^{125} & a^{209} & a^{213} & a^{173} & a^{129} & a^{48} & a^{249} \\
a^{59} & a^{160} & a^{85} & a^{129} & a^{48} & a^{99} & a^{239} & a^{160} & a^{253} & a^{48} & a^{219} & a^{249} & a^{192} \\
a^{160} & a^{232} & a^{15} & a^{155} & a^{79} & a^{122} & a^{100} & a^{122} & a^{100} & a^{122} & & &
\end{array} \tag{1}$$

Next we will consider error correction in QR codes.

3 Reed-Solomon Codes

Since their description in a 1960 paper by Irving Reed and Gustave Solomon, Reed-Solomon codes have been widely used to ensure reliable transmission of data. In addition to their utility in QR codes, Reed-Solomon codes have been used extensively to correct data transmission errors in mobile phones, compact discs, satellites, space probes, and numerous other examples.

The objects in which Reed-Solomon codes can actually correct errors are finite field elements, in a field containing 2^m elements for some positive integer m, generated by a primitive polynomial $p(x)$ of degree m, with primitive element we will again denote by a. To create a Reed-Solomon code, the desired maximum number of position (i.e., polynomial coefficient) errors guaranteed to be correctable upon the arrival of information at its destination must first be chosen, and specified. Let t be this desired maximum number of position errors guaranteed to be correctable, which can be any positive integer t with $2t < n$, where $n = 2^m - 1$ is the number of nonzero elements in the field.

With any error correcting code, transmitted messages are called *codewords*. Reed-Solomon codewords are polynomials consisting of a *prefix* and a *suffix*.[2] The prefix of a codeword is constructed starting with a polynomial of the form

$$m(x) = b_{k-1}x^{k-1} + \cdots + b_1x + b_0,$$

whose coefficients are understood to contain the information actually needing to be sent. As such, we will call $m(x)$ the *message polynomial*. To form the prefix, we use the polynomial

$$g(x) = (x-1)(x-a)\cdots(x-a^{u-1}),$$

where $u = 2t$ or $u = 2t+1$. The polynomial $g(x)$ is called the *generating polynomial*. We then form the prefix by computing $x^u m(x)$.

The suffix polynomial $s(x)$ is the remainder when the prefix polynomial is divided by $g(x)$. That is, if

$$x^u m(x) = q(x)g(x) + s(x) \tag{2}$$

with $\deg s(x) < \deg g(x)$ or $s(x) = 0$, then the suffix is $s(x)$.

For the various QR code versions and error correction levels, different numbers of prefix polynomials of given degrees along with generating polynomials of specified degrees are used to create codewords. Table 4 summarizes these numbers for versions 1–10 with error correction level H.

Version	**Num Codewords**	**Num** $m(x)$	**Deg** $m(x)$	**Deg** $g(x)$
1	1	1	8	16
2	1	1	15	27
3	2	2	11	21
4	4	4	8	16
5	4	2; 2	10; 11	22
6	4	4	14	28
7	5	4; 1	12; 13	26
8	6	4; 2	13; 14	26
9	8	4; 4	11; 12	24
10	8	6; 2	14; 15	28

Table 4: Codeword formulations for QR code versions 1–10 with error correction level H.

[2] Other descriptions of Reed-Solomon codes specify codewords as single polynomials. Separating the prefix and suffix is useful though, because it allows for the coefficients of the data to be visible in the prefix.

Codeword formulations for all of the various QR code versions and error correction levels can be found in [5].

From Table 4, we can see that a QR code of version 3 with error correction level H requires that information be formulated into two codewords, both with prefixes of degree 11, resulting in 12 polynomial coefficients containing data. The generating polynomial is then of degree 21, and has the form $g(x) = (x-1)(x-a)\cdots(x-a^{20})$. Since this code is guaranteed to correct $t = 10$ position errors, each suffix corresponding to each prefix will have maximum degree 20, resulting in 21 polynomial coefficients.

Also from Table 4, we can see that a QR code of version 8 with error correction level H requires that information be formulated into six codewords, four having prefixes of degree 13, resulting in 14 polynomial coefficients containing data, and two having prefixes of degree 14, resulting in 15 polynomial coefficients containing data. The generating polynomial is then of degree 26, and has the form $g(x) = (x-1)(x-a)\cdots(x-a^{25})$. Since this code is guaranteed to correct $t = 13$ position errors, each suffix corresponding to each prefix will have maximum degree 25, resulting in 26 polynomial coefficients.

In Section 2, the data given by (1) for the message `https://atcm.mathandtech.org/` was represented for a QR code of version 4 with error correction level H. Table 4 indicates that this data should be formulated into four codewords, all with prefixes of degree 8, resulting in 9 polynomial coefficients containing data. The generating polynomial for the code is then of degree 16, and has the form $g(x) = (x-1)(x-a)\cdots(x-a^{15})$. Since this code is guaranteed to correct $t = 8$ position errors, each suffix corresponding to each prefix will have maximum degree 15, resulting in 16 polynomial coefficients. The data given by (1) shows the 288-bit full data string expressed as 36 primitive power finite field elements. Since there are four prefixes, we must divide these 36 powers of a into four equal parts, with each part representing the 9 coefficients of each of the prefix polynomials of degree 8. Once these prefixes are formed, we divide each by the generating polynomial $g(x)$, and using (2), find the four corresponding suffixes.

To demonstrate this process, for the 36 primitive power finite field elements given by (1), we take the first 9 of these elements, which are

$$a^{191}, a^{85}, a^{13}, a^{253}, a^{253}, a^{198}, a^{125}, a^{209}, a^{213},$$

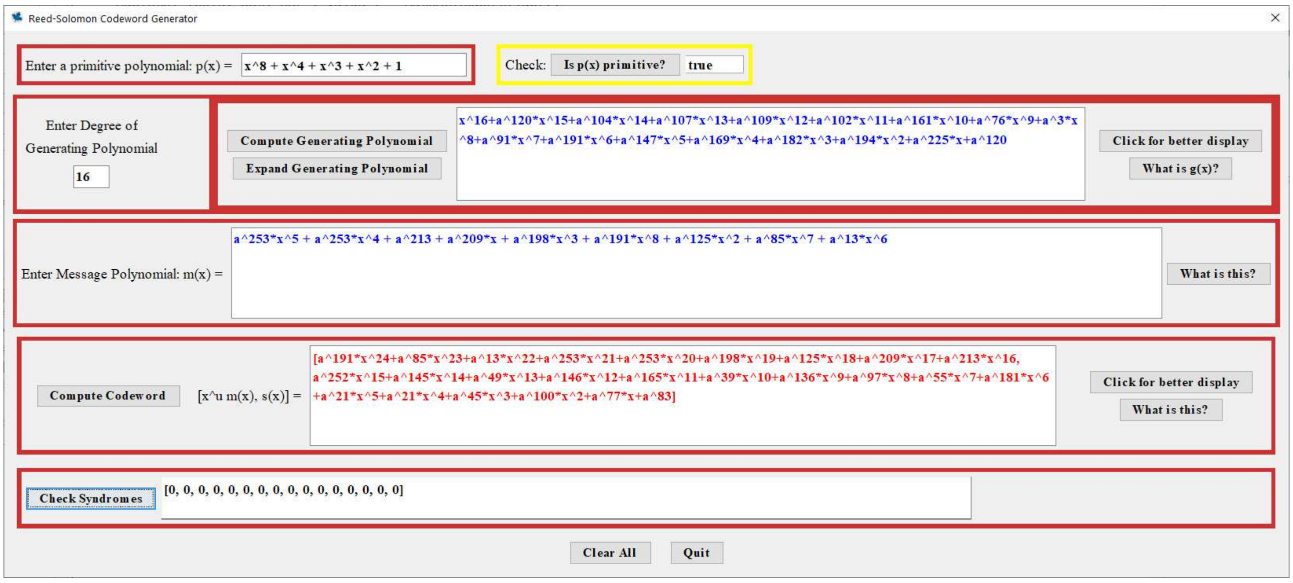
and form the following message polynomial in descending degree order.

$$m(x) = a^{191}x^8 + a^{85}x^7 + a^{13}x^6 + a^{253}x^5 + a^{253}x^4 + a^{198}x^3 + a^{125}x^2 + a^{209}x + a^{213} \tag{3}$$

Since the degree of the generating polynomial is $u = 16$, we multiply $m(x)$ by $x^u = x^{16}$ to form the following prefix.

$$x^{16}m(x) = a^{191}x^{24} + a^{85}x^{23} + a^{13}x^{22} + a^{253}x^{21} + a^{253}x^{20} + a^{198}x^{19} + a^{125}x^{18} + a^{209}x^{17} + a^{213}x^{16}$$

To form the suffix, we must divide this prefix by the generating polynomial $g(x)$. To do this division, we will use another Maplet written by the authors, **ReedSolomonCodewordGeneratorATCM**, which has been designed to do this division and form the prefix/suffix codeword pair. Figure 3 shows how this Maplet can be used for this purpose. The Maplet specifically allows a user to enter the primitive polynomial, the degree of the generating polynomial, and the message polynomial, and then by clicking the appropriate buttons, output the generating polynomial in expanded form and the prefix/suffix codeword pair.



```
Reed-Solomon Codeword Generator

Enter a primitive polynomial: p(x) = x^8 + x^4 + x^3 + x^2 + 1
Check: true

Enter Degree of Generating Polynomial: 16
x^16+a^120*x^15+a^104*x^14+a^107*x^13+a^109*x^12+a^102*x^11+a^161*x^10+a^76*x^9+a^3*x
^8+a^91*x^7+a^191*x^6+a^147*x^5+a^169*x^4+a^182*x^3+a^194*x^2+a^225*x+a^120

Enter Message Polynomial: m(x) = a^253*x^5 + a^253*x^4 + a^213 + a^209*x + a^198*x^3 + a^191*x^8 + a^125*x^2 + a^85*x^7 + a^13*x^6

[x^u m(x), s(x)] = [a^191*x^24+a^85*x^23+a^13*x^22+a^253*x^21+a^253*x^20+a^198*x^19+a^125*x^18+a^209*x^17+a^213*x^16,
a^252*x^15+a^145*x^14+a^49*x^13+a^146*x^12+a^165*x^11+a^39*x^10+a^136*x^9+a^97*x^8+a^55*x^7+a^181*x^6
+a^21*x^5+a^21*x^4+a^45*x^3+a^100*x^2+a^77*x+a^83]

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figure 3: Prefix/suffix Reed-Solomon codeword generation.

From Figure 3, we can see that the prefix/suffix pair corresponding to the message polynomial (3) is given by the following.

$$\begin{aligned}
x^{16}m(x) &= a^{191}x^{24}+a^{85}x^{23}+a^{13}x^{22}+a^{253}x^{21}+a^{253}x^{20}+a^{198}x^{19}+a^{125}x^{18}+a^{209}x^{17}+a^{213}x^{16}\\
s(x) &= a^{252}x^{15}+a^{145}x^{14}+a^{49}x^{13}+a^{146}x^{12}+a^{165}x^{11}+a^{39}x^{10}+a^{136}x^{9}+a^{97}x^{8}+a^{55}x^{7}\\
&\quad +a^{181}x^{6}+a^{21}x^{5}+a^{21}x^{4}+a^{45}x^{3}+a^{100}x^{2}+a^{77}x+a^{83}
\end{aligned}$$

The following summarizes all four prefix/suffix codeword pairs, which can be found using the Maplet.

$$\begin{aligned}
x^{16}m_1(x) &= a^{191}x^{24}+a^{85}x^{23}+a^{13}x^{22}+a^{253}x^{21}+a^{253}x^{20}+a^{198}x^{19}+a^{125}x^{18}+a^{209}x^{17}+a^{213}x^{16}\\
s_1(x) &= a^{252}x^{15}+a^{145}x^{14}+a^{49}x^{13}+a^{146}x^{12}+a^{165}x^{11}+a^{39}x^{10}+a^{136}x^{9}+a^{97}x^{8}+a^{55}x^{7}\\
&\quad +a^{181}x^{6}+a^{21}x^{5}+a^{21}x^{4}+a^{45}x^{3}+a^{100}x^{2}+a^{77}x+a^{83}
\end{aligned}$$

$$\begin{aligned}
x^{16}m_2(x) &= a^{173}x^{24}+a^{129}x^{23}+a^{48}x^{22}+a^{249}x^{21}+a^{59}x^{20}+a^{160}x^{19}+a^{85}x^{18}+a^{129}x^{17}+a^{48}x^{16}\\
s_2(x) &= a^{9}x^{15}+a^{83}x^{14}+a^{148}x^{13}+a^{241}x^{12}+a^{94}x^{11}+a^{42}x^{10}+a^{74}x^{9}+a^{161}x^{8}+a^{52}x^{7}+a^{82}x^{6}\\
&\quad +a^{188}x^{5}+a^{4}x^{4}+a^{167}x^{3}+a^{241}x^{2}+a^{111}x+a^{97}
\end{aligned}$$

$$\begin{aligned}
x^{16}m_3(x) &= a^{99}x^{24}+a^{239}x^{23}+a^{160}x^{22}+a^{253}x^{21}+a^{48}x^{20}+a^{219}x^{19}+a^{249}x^{18}+a^{192}x^{17}+a^{160}x^{16}\\
s_3(x) &= a^{146}x^{14}+a^{84}x^{13}+a^{32}x^{12}+a^{206}x^{11}+a^{47}x^{10}+a^{109}x^{9}+a^{36}x^{8}+a^{151}x^{7}+a^{208}x^{6}\\
&\quad +a^{105}x^{5}+a^{65}x^{4}+a^{134}x^{3}+a^{74}x^{2}+a^{61}x+a^{38}
\end{aligned}$$

$$\begin{aligned}
x^{16}m_4(x) &= a^{232}x^{24}+a^{15}x^{23}+a^{155}x^{22}+a^{79}x^{21}+a^{122}x^{20}+a^{100}x^{19}+a^{122}x^{18}+a^{100}x^{17}+a^{122}x^{16}\\
s_4(x) &= a^{151}x^{15}+a^{48}x^{14}+a^{123}x^{13}+a^{235}x^{12}+a^{234}x^{11}+a^{25}x^{10}+a^{68}x^{9}+a^{45}x^{8}+a^{169}x^{7}\\
&\quad +a^{245}x^{6}+a^{219}x^{5}+a^{138}x^{4}+a^{43}x^{3}+a^{51}x^{2}+a^{229}x+a^{63}
\end{aligned}$$

To check whether the prefix and/or suffix contain any errors after their transmission, (2) is solved as $q(x)g(x) = x^u m(x) - s(x)$. Since $x^u m(x) - s(x)$ is a multiple of $g(x)$, the roots of $g(x)$ must also be roots of $x^u m(x) - s(x)$. As such, when $x^u m(x) - s(x)$ is evaluated at the roots of $g(x)$, which are $1, a, \ldots, a^{u-1}$, each result must be 0. These roots, $1, a, \ldots, a^{u-1}$, are called the *syndromes* of $x^u m(x) - s(x)$.

As shown in the Maplet window in Figure 3, the syndromes for $x^{16} m_1(x) - s_1(x)$, as evaluated for the roots $1, a, \ldots, a^{15}$ of the generating polynomial $g(x)$ of degree 16, are all 0, indicating that the prefix and suffix in that example are correct. When errors occur in either the prefix or suffix coefficients, Reed-Solomon codes provide an error-correction process through which the location of the errors can be found, and the errors corrected. We provide some details of this process in [2].

After the prefix/suffix codeword pairs are created, the data is then interleaved together. The purpose of this interleaving is so that when the data is placed into the QR code, localized damage to the QR code and an overwhelming of its capacity to correct errors can be prevented. This interleaving of the data occurs as follows.

- The coefficients of each prefix are written consecutively in order across the rows of a matrix, and then taken in order down the columns of this matrix, skipping over any blank entries that may have resulted from prefixes of different lengths.

- The coefficients of each suffix are written consecutively in order across the rows of a matrix, and then taken in order down the columns of this matrix.

- The interleaved prefixes and suffixes are then joined together to represent the data.

As an example of this, with the four prefix/suffix pairs we gave previously, we first write the coefficients of the prefixes in order across the rows in a 4×9 array.

$$\begin{array}{ccccccccc}
a^{191} & a^{85} & a^{13} & a^{253} & a^{253} & a^{198} & a^{125} & a^{209} & a^{213} \\
a^{173} & a^{129} & a^{48} & a^{249} & a^{59} & a^{160} & a^{85} & a^{129} & a^{48} \\
a^{99} & a^{239} & a^{160} & a^{253} & a^{48} & a^{219} & a^{249} & a^{192} & a^{160} \\
a^{232} & a^{15} & a^{155} & a^{79} & a^{122} & a^{100} & a^{122} & a^{100} & a^{122}
\end{array}$$

We then interleave these prefix coefficients by taking them in order down the columns of this array. This results in the following.

$$\begin{array}{cccccccccccc}
a^{191} & a^{173} & a^{99} & a^{232} & a^{85} & a^{129} & a^{239} & a^{15} & a^{13} & a^{48} & a^{160} & a^{155} \\
a^{253} & a^{249} & a^{253} & a^{79} & a^{253} & a^{59} & a^{48} & a^{122} & a^{198} & a^{160} & a^{219} & a^{100} \\
a^{125} & a^{85} & a^{249} & a^{122} & a^{209} & a^{129} & a^{192} & a^{100} & a^{213} & a^{48} & a^{160} & a^{122}
\end{array}$$

We next write the coefficients of the suffixes in order across the rows in a 4×16 array.

$$\begin{array}{cccccccccccccccc}
a^{252} & a^{145} & a^{49} & a^{146} & a^{165} & a^{39} & a^{136} & a^{97} & a^{55} & a^{181} & a^{21} & a^{21} & a^{45} & a^{100} & a^{77} & a^{83} \\
a^{9} & a^{83} & a^{148} & a^{241} & a^{94} & a^{42} & a^{74} & a^{161} & a^{52} & a^{82} & a^{188} & a^{4} & a^{167} & a^{241} & a^{111} & a^{97} \\
0 & a^{146} & a^{84} & a^{32} & a^{206} & a^{47} & a^{109} & a^{36} & a^{151} & a^{208} & a^{105} & a^{65} & a^{134} & a^{74} & a^{61} & a^{38} \\
a^{151} & a^{48} & a^{123} & a^{235} & a^{234} & a^{25} & a^{68} & a^{45} & a^{169} & a^{245} & a^{219} & a^{138} & a^{43} & a^{51} & a^{229} & a^{63}
\end{array}$$

We then interleave these suffix coefficients by taking them in order down the columns of this array. This results in the following.

$$\begin{array}{cccccccccccccccc}
a^{252} & a^{9} & 0 & a^{151} & a^{145} & a^{83} & a^{146} & a^{48} & a^{49} & a^{148} & a^{84} & a^{123} & a^{146} & a^{241} & a^{32} & a^{235} \\
a^{165} & a^{94} & a^{206} & a^{234} & a^{39} & a^{42} & a^{47} & a^{25} & a^{136} & a^{74} & a^{109} & a^{68} & a^{97} & a^{161} & a^{36} & a^{45} \\
a^{55} & a^{52} & a^{151} & a^{169} & a^{181} & a^{82} & a^{208} & a^{245} & a^{21} & a^{188} & a^{105} & a^{219} & a^{21} & a^{4} & a^{65} & a^{138} \\
a^{45} & a^{167} & a^{134} & a^{43} & a^{100} & a^{241} & a^{74} & a^{51} & a^{77} & a^{111} & a^{61} & a^{229} & a^{83} & a^{97} & a^{38} & a^{63}
\end{array}$$

Combining the interleaved prefix and suffix coefficients gives the following full data for the QR code.

$$\begin{array}{ccccccccccccccccc}
a^{191} & a^{173} & a^{99} & a^{232} & a^{85} & a^{129} & a^{239} & a^{15} & a^{13} & a^{48} & a^{160} & a^{155} & a^{253} & a^{249} & a^{253} & a^{79} & a^{253} \\
a^{59} & a^{48} & a^{122} & a^{198} & a^{160} & a^{219} & a^{100} & a^{125} & a^{85} & a^{249} & a^{122} & a^{209} & a^{129} & a^{192} & a^{100} & a^{213} & a^{48} \\
a^{160} & a^{122} & a^{252} & a^{9} & 0 & a^{151} & a^{145} & a^{83} & a^{146} & a^{48} & a^{49} & a^{148} & a^{84} & a^{123} & a^{146} & a^{241} & a^{32} \\
a^{235} & a^{165} & a^{94} & a^{206} & a^{234} & a^{39} & a^{42} & a^{47} & a^{25} & a^{136} & a^{74} & a^{109} & a^{68} & a^{97} & a^{161} & a^{36} & a^{45} \\
a^{55} & a^{52} & a^{151} & a^{169} & a^{181} & a^{82} & a^{208} & a^{245} & a^{21} & a^{188} & a^{105} & a^{219} & a^{21} & a^{4} & a^{65} & a^{138} & a^{45} \\
a^{167} & a^{134} & a^{43} & a^{100} & a^{241} & a^{74} & a^{51} & a^{77} & a^{111} & a^{61} & a^{229} & a^{83} & a^{97} & a^{38} & a^{63} & &
\end{array} \tag{4}$$

This interleaved prefix and suffix data gives a total of 100 primitive element powers. Each power can be converted into a byte, giving a total of $8 \cdot 100 = 800$ bits to represent the data in full for the code. Some QR code versions also require a string of 0s to be included to completely fill out the data. This number of additional 0s, which depends on the version, is given in [5]. QR codes of version 4 require seven additional 0s. Thus, for our example, the string 0000000 would be appended to the 800 bits representing the data, giving a total of 807 bits for the data section of the QR code. Next we will describe how this data is placed into the QR code, and the code itself created.

4 Data Placement and QR Code Creation

QR codes include several components placed in precise locations in a two-dimensional array. Figure 4 gives a summary of the placement of components in a QR code of version 4.

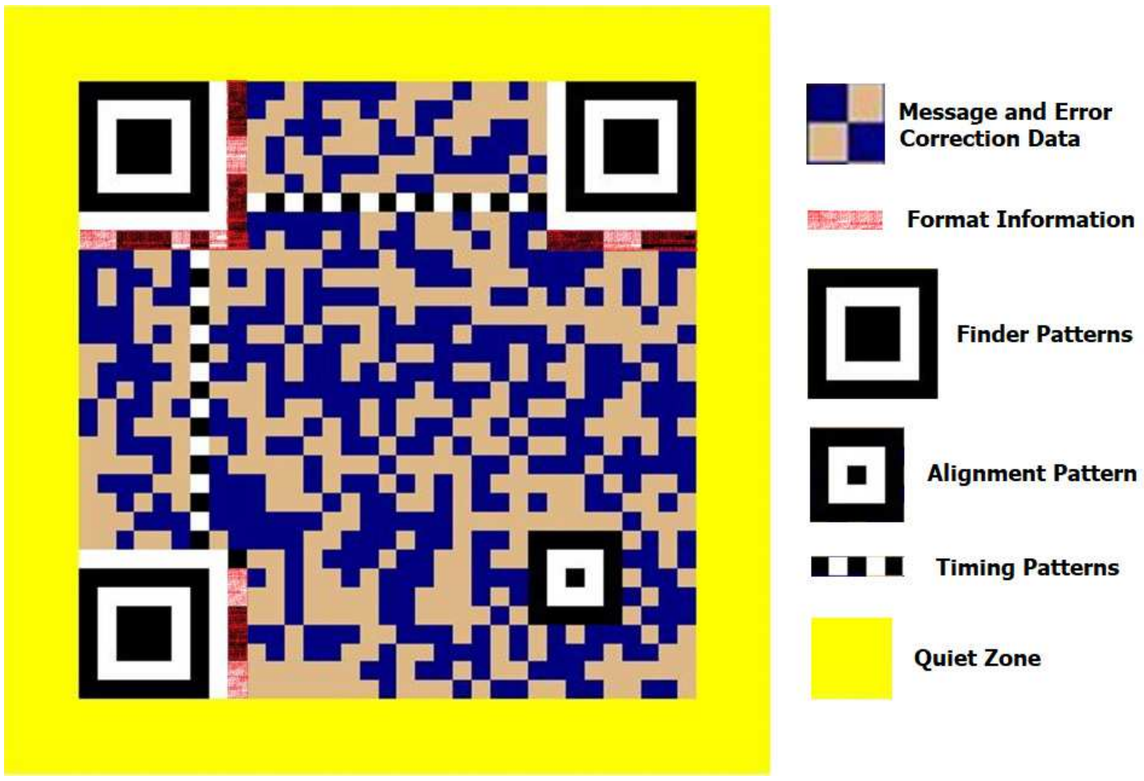


Figure 4: Component summary for a QR code of version 4.

A QR code of version 4 is essentially a 33×33 matrix. In Figure 4, the message and error correction data is located in the small dark blue and light brown squares. Each square in this data placement represents a binary digit, for which normally a 1 is represented by a darker color (dark blue, in this case), and a 0 by a lighter color (light brown, in this case). The data is placed in the image starting in the lower right corner. The regions outlined in red in Figure 4 are reserved for the *format information*, which is a string of bits giving, among other things, the error correction level (L, M, Q, or H) for the code. The format information also gives the *mask pattern*, which we will describe in more detail later. Error correction bits are also included as part of the format information in case any of the information is lost. The *function patterns* consist of the *finder patterns*, *alignment patterns*, and *timing patterns*, as well as a *dark module*, which is a single black square located directly above the format information block near the lower left corner of the matrix. *Separators* shown as solid white rectangular strips in Figure 4 detach each finder pattern from the rest of the QR code. The function patterns are designed to be placed in specific areas of the QR code, and their purpose is to ensure that QR code scanners can correctly identify and orient an image for decoding. More information about how the function patterns are aligned in a QR code can be found in [5]. Finally, a *quiet zone* is included, as indicated in yellow in Figure 4, as an area of lighter color designed to enclose the main QR code matrix.

QR codes of versions 7–40 require more alignment patterns and an additional 18-bit block placed in two parts of the matrix containing information about the QR code version. More information can be found about their generation and placement in [5].

For the message `https://atcm.mathandtech.org/`, Figure 5 shows how the data given by (4) is placed in a QR code of version 4.

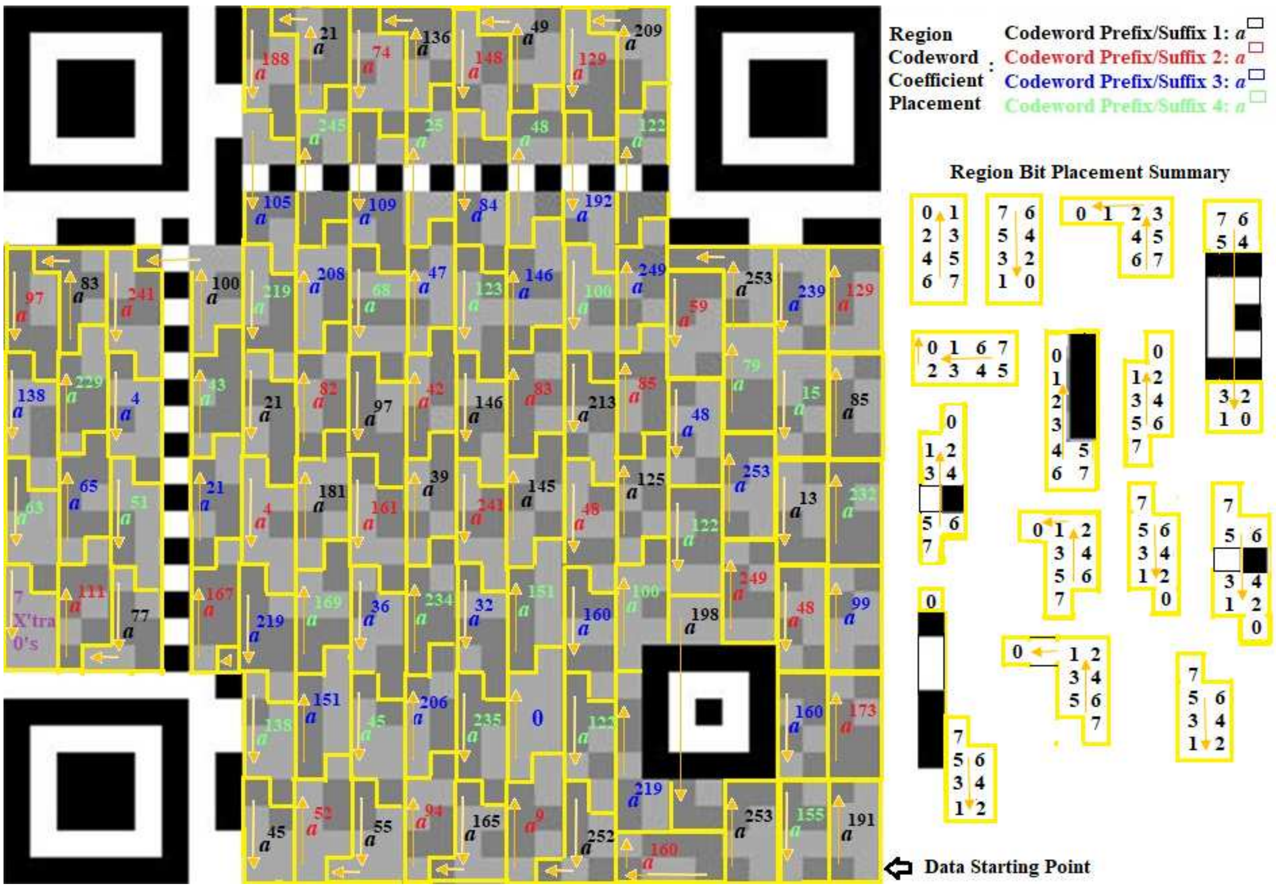

Region Codeword Coefficient Placement:
Codeword Prefix/Suffix 1: a
Codeword Prefix/Suffix 2: a
Codeword Prefix/Suffix 3: a
Codeword Prefix/Suffix 4: a
Region Bit Placement Summary
X'tra 0's
Data Starting Point

Figure 5: QR code data placement for the message `https://atcm.mathandtech.org/`.

Starting in the lower right corner of a QR code matrix, the data for each codeword prefix/suffix is placed in blocks in an up-and-down pattern. Each primitive power element is color-coded to illustrate the codeword prefix/suffix with which it is associated, as well as how the interleaving spreads each codeword prefix/suffix throughout the matrix. For each individual block, the eight binary digits representing the primitive power element are placed in a zig-zag pattern. The region bit placement summary section in Figure 4 indicates the order in which the binary digits are placed within similarly shaped blocks, with the coefficients of each polynomial of degree 7 representing each primitive element power placed in descending degree order. For example, the first primitive power element placed in the lower right corner, a^{191}, is represented by the polynomial $a^6 + 1$. The coefficients of this polynomial are read in descending order, 01000001, and placed in the first rectangular block. The order of the placement is indicated by the first rectangular block under the region bit placement summary section in Figure 4.

After the data is placed into the QR code matrix, to make the data as readable as possible by a QR code scanner, the process of *masking* is applied. When data in a QR code is masked, particular data squares, depending on their location in the matrix, are flipped, with a dark square becoming light, and vice versa. Only data squares are flipped though. Squares involving function patterns, version, and format information are left intact. QR code specifications define eight mask patterns that can be applied to a QR code. The mask pattern chosen is dependent on a calculated penalty score from certain observed square occurrences in the QR code matrix. The different type of mask patterns and how the penalty score is calculated for each are summarized in [5].

For the QR code of version 4 representing `https://atcm.mathandtech.org/`, mask pattern 2 is chosen. Numbering the 33 columns of this matrix from $0, 1, \ldots, 32$, mask pattern 2 flips all of the data squares in all columns with (column number) mod $3 = 0$. Figure 6 illustrates this QR code before and after masking, with the column numbers labeled where the masking occurs.

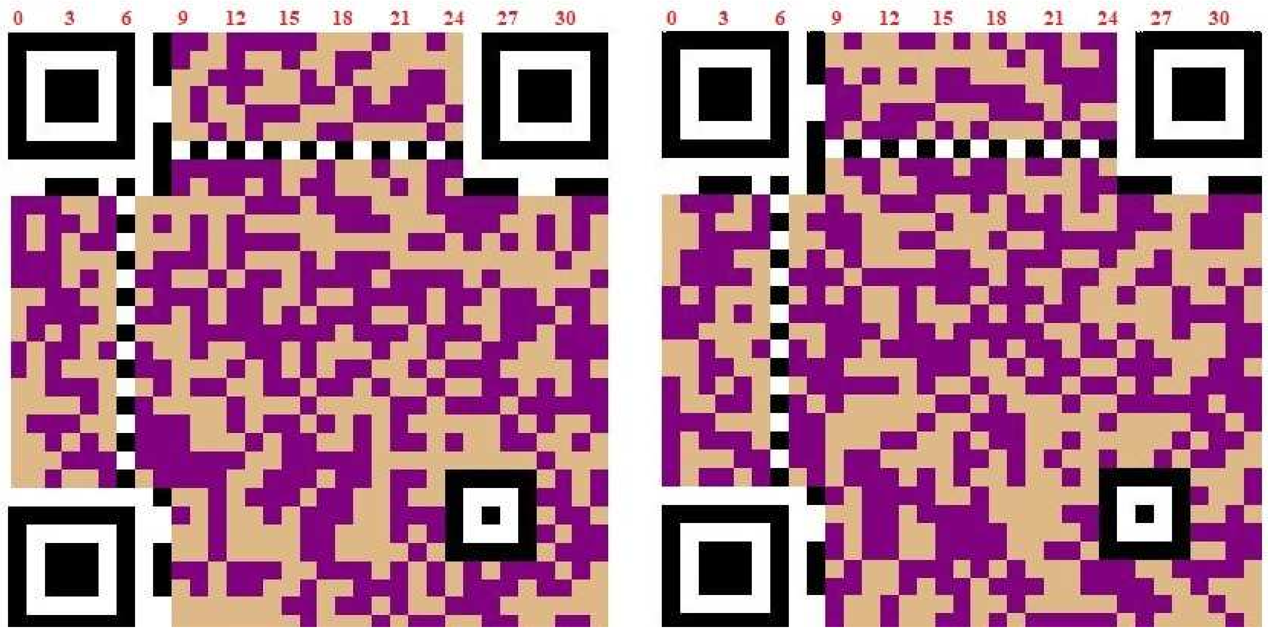
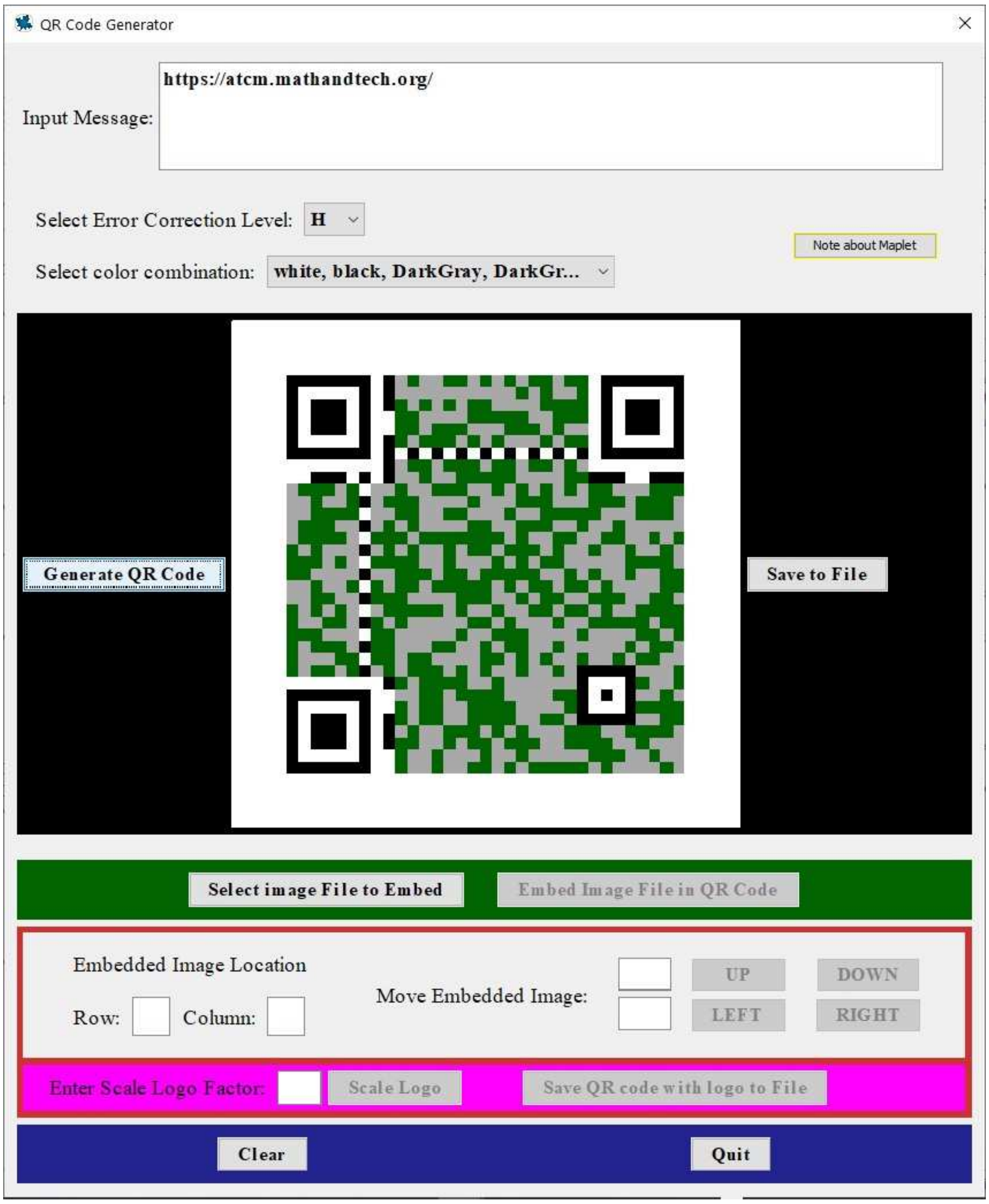


Figure 6: QR code example before and after masking.

After masking, a QR code is ready to be output. For generating QR codes, we will use another Maplet written by the authors, **QRcode**. This Maplet allows users to input a message to be included, select an error correction level, and then generate the resulting QR code. The Maplet automatically generates the QR code with a specific version based on how much and what type of data is entered and the error correction level selected. Figure 7 shows the output for a QR code of version 4 and with error correction level H that will scan to the message `https://atcm.mathandtech.org/`.


QR Code Generator

Input Message: https://atcm.mathandtech.org/

Select Error Correction Level: H

Select color combination: white, black, DarkGray, DarkGr...

Figure 7: QR code Maplet generation example for the message `https://atcm.mathandtech.org/`.

With error correction included, images up to a certain size can be overlaid onto a QR code to help identify its message contents. With error correction level H, even with 30% of the message data squares covered by the image, the QR code would still scan correctly. The **QRcode** Maplet can select a jpg file of a user's choice and overlay it onto a generated QR code. Figure 8 shows the result with the ATCM logo overlaid onto a QR code that would still scan to `https://atcm.mathandtech.org/`.

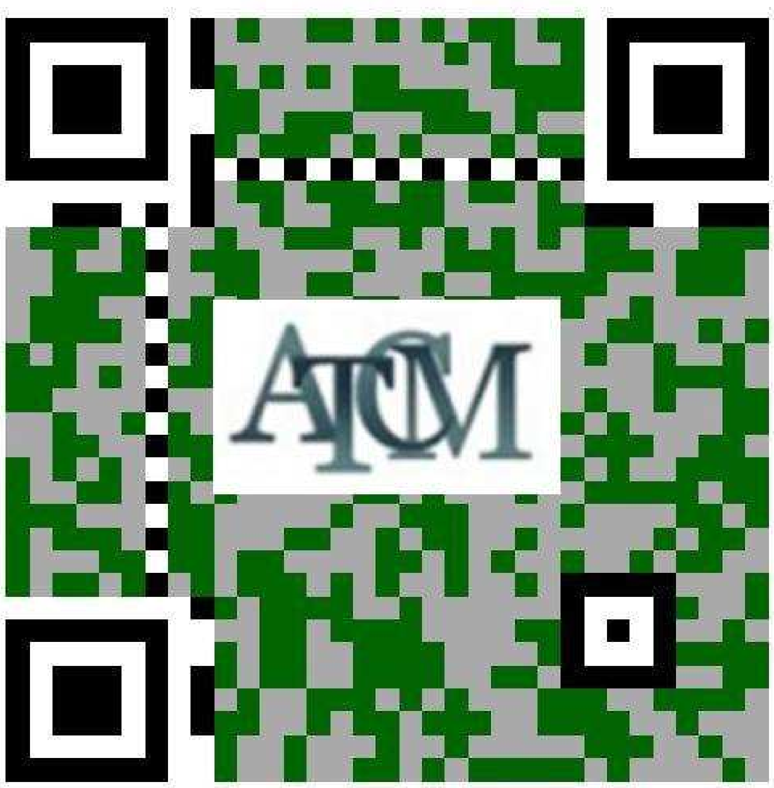


Figure 8: QR code with an overlaid image.

5 Conclusion

In this paper, we showed the basics of how data is formatted and placed into QR codes, with error correction included via Reed-Solomon codes. Maplets written by the authors were used to demonstrate this. These Maplets are available for download at [4].

The QR code generated in this paper focused on specific data that was generated for a particular version. Additional explanations for how other data can be formatted and placed into QR codes of other versions can be found in [5].

References

[1] Eugenia Cheng, 2021. The Numbers Hiding Behind That QR Code. Available at: https://www.wsj.com/articles/the-numbers-hiding-behind-that-qr-code-11629411899.

[2] Richard Klima, Neil Sigmon, and Ernest Stitzinger, *Applied Abstract Algebra with Maple and MATLAB, Third Edition*, Taylor & Francis, Boca Raton, FL, 2016.

[3] Rudolf Lidl and Harald Neiderreiter, *Introduction to Finite Fields and their Applications*, Cambridge U. Press, New York, 1986.

[4] Neil Sigmon, 2021. Maplet Download Page for The Mathematics of QR Codes. Available at: https://www.radford.edu/npsigmon/qrcodes/paper.html.

[5] Thonky.com, 2021. QR Code Tutorial. Available at: https://www.thonky.com/qr-code-tutorial/.