Creating QR Codes Using Maplets

Adam S. Downs, Neil P. Sigmon
asdowns@radford.edu, npsigmon@radford.edu
Department of Mathematics and Statistics
Radford University
Radford, Virginia 24142
USA

Richard E. Klima
klimare@appstate.edu
Department of Mathematical Sciences
Appalachian State University
Boone, North Carolina 28608
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. Creating a QR code 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 encoding information in the form of polynomial coefficients using finite field arithmetic. Utilizing Reed-Solomon codes can sometimes allow a logo and emblem to be embedded within a QR code to advertise its purpose, even while the logo or emblem covers part of the data to be scanned. In this paper we will describe how QR codes are constructed, and how Reed-Solomon codes are incorporated within them to provide error correction. To assist in demonstrating this, we will use technology involving Maplets.

1 Introduction

Since their invention by the DENSO Corporation in Japan in 1994, QR codes have provided a widely used method for quickly accessing information. Originally, QR codes were used for parts manufac-

turing inventory tracking, but they can now be found on advertisements, web pages, business cards, and 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 constructed, and how data is integrated into their formation. As part of this, we will describe the basics of how Reed-Solomon codes work, and how Reed-Solomon codes are incorporated within QR codes to increase the likelihood that they are able to convey data reliably. This information provides a means for demonstrating a hands-on method for the use of non-trivial mathematics in a practical real-life application.

To assist with this, we will use technology involving a series of Maplets that we have written specifically for this paper. 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. This means that our Maplets, which we provide for download at [5], are only directly usable by readers who have access to Maple, which is notably not an open-source system. However, our primary audience for this paper is academic professionals like ourselves, many of whom already have access to Maple through site licenses provided by their home institutions. In particular, according to a media release that can be found at [4], Maplesoft products and services are available for use at more than 8000 educational institutions, research labs, and companies, in over 90 countries.

Even for professionals (and amateurs) without access to Maple though, we still believe this paper has value. We ultimately have two purposes for this paper. First, as we have noted, we believe this paper demonstrates a hands-on method for the use of non-trivial mathematics in a practical real-life application. More to the point, while we have studied and written about Reed-Solomon codes extensively in other places such as [2], in these other places we have only been able to connect Reed-Solomon codes to applications like satellite transmissions and compact discs for which realistic examples like the kind that can be found in this paper are not reasonable to produce. Second, our hope is that readers of this paper, upon seeing how we were able to create QR codes using our favorite software system, will be motivated to explore the possibility of creating QR codes using their own favorite software system, open-source or not, to which they have access and of which they have their own expertise.

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 byte mode. Descriptions of the other three modes can be found in [6]. The byte mode of encoding text for QR codes utilizes the ASCII character set. A list of these characters and correspondences between them and decimal numbers in ASCII for the printable characters on a modern keyboard is shown in Table 1.

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

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

The decimal number to which an ASCII character corresponds can be expressed as a string of binary digits, or *bits*, in a block of eight digits called 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 a 1. Specifically, if a computation results in a number that is divisible by 2, then it can be reduced to 0, and if it results in a number that is not divisible by 2, then it can be reduced to 1. For example, the number 7 with modulo 2 arithmetic reduces to 1; that is, $7 \bmod 2 = 1$. On the other hand, the number -4 with modulo 2 arithmetic reduces to 0; that is, $-4 \bmod 2 = 0$. 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 field elements which are nonzero. These 255 nonzero field elements can be expressed as all nonzero polynomials in 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 the elements in this set.

To represent these 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.

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

This process could be continued through the last nonzero element, $a^{255} = 1$. 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, $p(x)$ is also characterized as a *primitive* polynomial.

We will now demonstrate a Maplet written by the authors that can be used to generate the elements in this finite field, with the nonzero elements in the field expressed as polynomials in a of maximum degree seven and with coefficients that are all either 0 or 1. This Maplet is available for download at the link labeled [S1] in Section 6. Figure 1 shows the result of using the Maplet for this purpose.

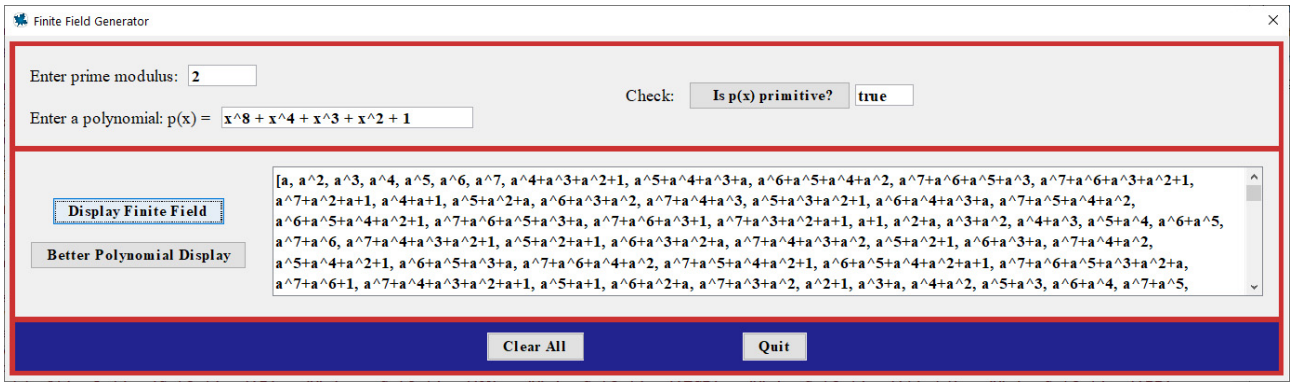

Figure 1: Finite field elements for the primitive polynomial $p(x) = x^8 + x^4 + x^3 + x^2 + 1$.

A QR code formats information 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 processed correctly even if parts 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 code that when scanned, results in the following message: `https://atcm.mathandtech.org/`. To begin, an error correction level is selected. Four possible error correction levels are available for QR codes: L, which provides error correction (i.e., allows a message to be processed correctly) if up to 7% of the code is unreadable; M, which provides error correction if up to 15% of the code is unreadable; Q, which provides error correction if up to 25% of the code is unreadable; and H, which provides error correction if up to 30% of the code is unreadable. 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 all QR code versions in byte mode and with error correction level H. A table showing the character capacities for all QR code versions in all possible modes and with all possible error correction levels can be found in [6].

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

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

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

Next, the information to be encoded into the grid is represented by a string of bits. The first four bits are the *mode indicator*, which for byte mode are 0100. The four mode indicator bits for all possible mode types can be found in [6]. 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 when converted to binary is 11101, these bits are included next, although they are first padded with three 0s at the start to provide 8 digits in total, as is required for the character count indicator for all QR code versions 1–9.[1] That is, the character count indicator for our message would actually be included as 00011101. 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 provide 8 digits each. These bytes are then included in order after the character count indicator. Using Table 1, we see that the characters in the message `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, which results in 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 all versions and error correction levels, QR codes require binary strings to completely fill what is called the *total capacity* of the code. For QR 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 fewer than four bits shorter than the total capacity, then the terminator would consist only of the number of 0s needed to reach the total capacity. If the binary string were four or more bits shorter than the total capacity though, then the terminator would consist of exactly four 0s. 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.

If the length of a string with the terminator is not a multiple of 8, then additional 0s must 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. However, since 248 bits is 40 bits shy of the 288-bit total capacity of a code of version 4 and with error correction level H, the string must still be padded with 40 additional bits. For this, bytes alternating between 11101100 and 00010001 are included until the total capacity is reached.

[1] QR code versions 10–40 also require character count indicators of a specified length, although this required length is 16 digits rather than 8.

Thus, for our example, to extend our string of length 248 to the required 288, we complete the data by including 11101100 00010001 11101100 00010001 11101100. A final list of the 288 bits for our example is shown in Table 3 (with color-coding we will use later).

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

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

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 convert this polynomial into a power of a in the finite field used with QR codes. For example, the second 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. This Maplet is available for download at the link labeled [S2] in Section 6. Figure 2 shows the result of using the Maplet for this purpose. Summarizing this result, the full data binary string for our example can be represented as the following string of powers of a.

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

Next we will consider error correction in QR codes.

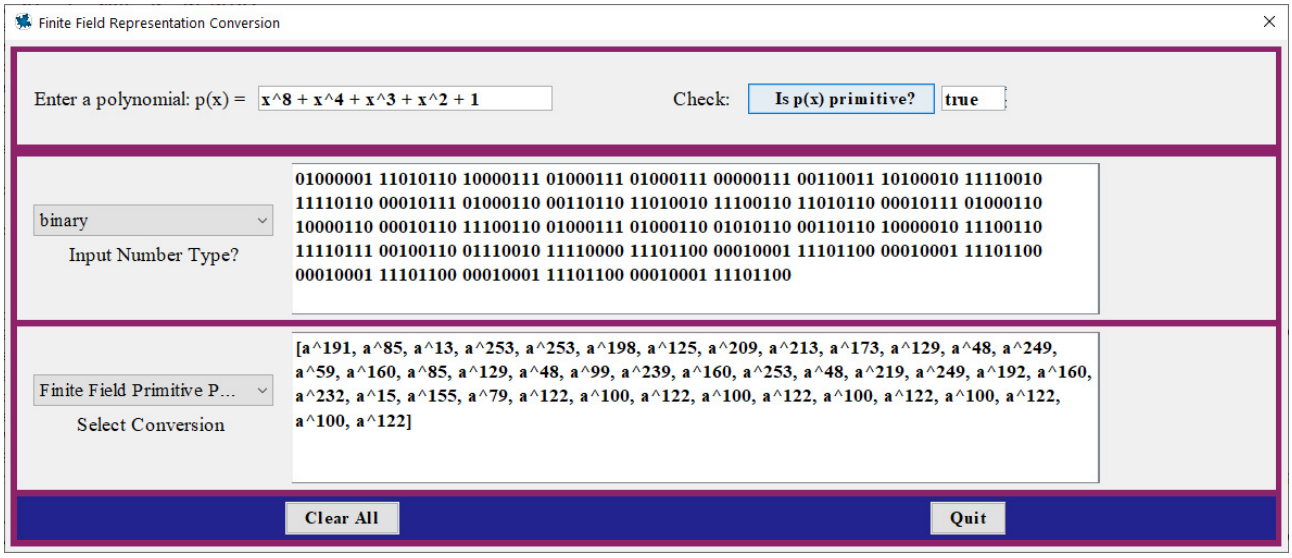

Figure 2: Primitive element power representation of data.

3 Reed-Solomon Codes

Since their description by Irving Reed and Gustave Solomon in 1960, 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 many 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. 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 $n = 2^m - 1$, which 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 formed starting with a polynomial of the form

$$m(x) = b_{k-1}x^{k-1} + \cdots + b_1 x + b_0,$$

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

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

[2]Some 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.

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 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 form codewords. Table 4 summarizes these numbers for versions 1–10 with error correction level H. Codeword formulations for all of the possible QR code versions and error correction levels can be found in [6].

Table 4: Codeword formulations for QR code 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

For example, 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 with prefixes of degree 11, resulting in 12 coefficients containing data. The generating polynomial is then of degree 21, of the form $g(x) = (x-1)(x-a)\cdots(x-a^{20})$. Since this code is guaranteed to correct up to $t = 10$ errors, the suffix corresponding to each prefix will have maximum degree 20, resulting in 21 coefficients.

As another example, 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 coefficients containing data. The generating polynomial is then of degree 26, of the form $g(x) = (x-1)(x-a)\cdots(x-a^{25})$. Since this code is guaranteed to correct up to $t = 13$ position errors, the suffix corresponding to each prefix will have maximum degree 25, resulting in 26 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 coefficients containing data. The generating polynomial for the code will then be of degree 16, of the form $g(x) = (x-1)(x-a)\cdots(x-a^{15})$. Since this code is guaranteed to correct up to $t = 8$ position errors, the suffix corresponding to each prefix will have maximum degree 15, resulting in 16 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 corresponding suffix.

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

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

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, which has been designed to do this division and form the prefix/suffix codeword pair. This Maplet is available for download at the link labeled [S3] in Section 6. Figure 3 shows the result of using the Maplet 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 on the appropriate buttons, find the generating polynomial in expanded form and the prefix/suffix codeword pair.

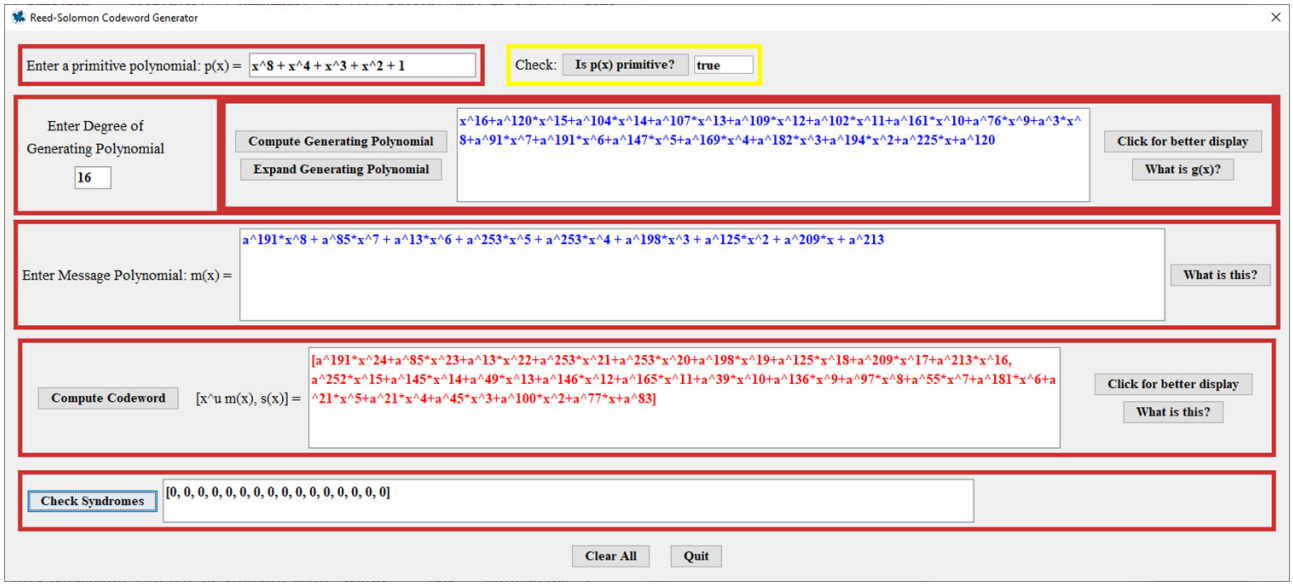

Figure 3: Prefix/suffix Reed-Solomon codeword generation.

From Figure 3, we can see that the prefix/suffix pair corresponding to the message polynomial (3) is 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 the four prefix/suffix codeword pairs for our example, all of 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 a prefix and/or suffix polynomial contain any errors after they have been received, equation (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 are called the *syndromes* of $x^u m(x) - s(x)$.

As can be seen at the bottom of 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. This indicates that the prefix and suffix in that example are correct.

When errors occur in either prefix or suffix coefficients, Reed-Solomon codes provide an error-correction process through which the locations of the errors can be discovered, and the errors themselves corrected. While there is not enough room for a description of the mathematics of this error correction process to be included in this paper, the mathematics of the process is described by the authors in [2], and the process can be done using another Maplet written by the authors. This Maplet is available for download at the link labeled [S4] in Section 6. Figure 4 shows the result of using the Maplet for this purpose.

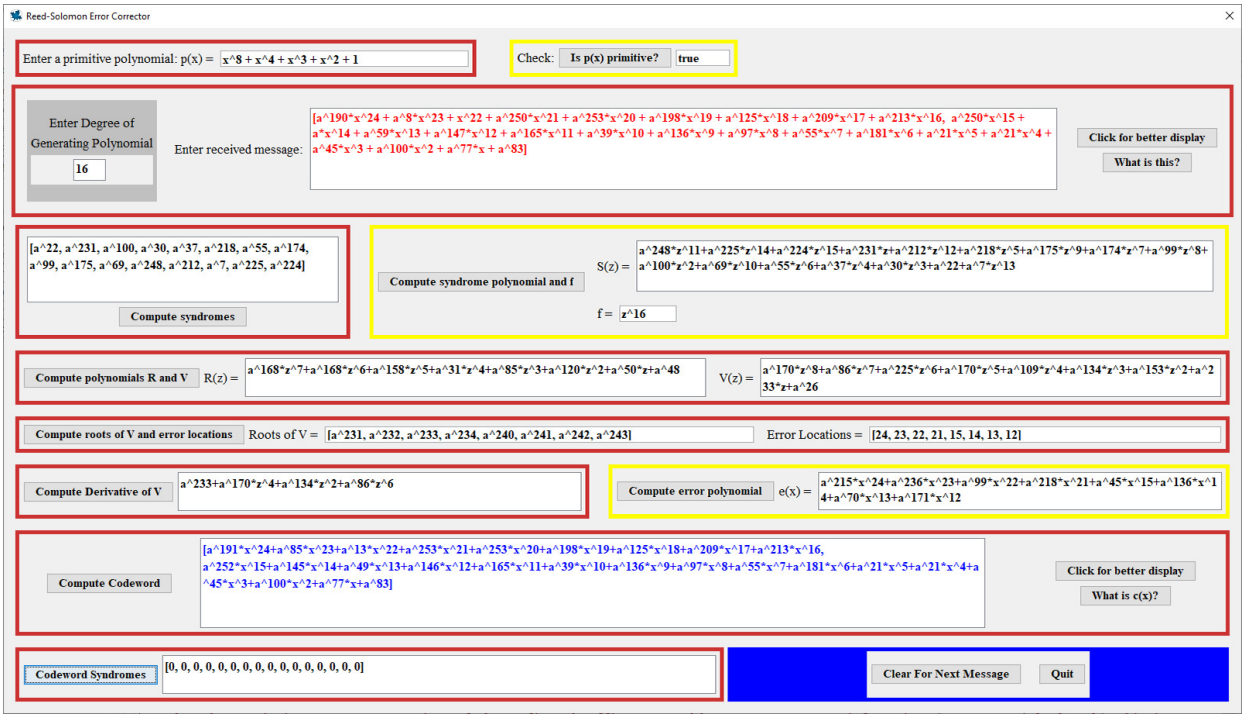

Figure 4: Prefix/suffix Reed-Solomon codeword error correction.

This Maplet specifically allows a user to enter the primitive polynomial, the degree of the generating polynomial, and a potential message expressed as a prefix/suffix polynomial pair, and then by clicking on the appropriate buttons, find the corrected prefix/suffix codeword pair. The example in the Maplet window in Figure 4 in particular shows a potential message being corrected to the previously-constructed prefix/suffix codeword pair

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

even though the potential message differs from the corrected pair in eight coefficients (specifically, in the x^{24} through x^{21} and x^{15} through x^{12} terms). This is consistent with the error correction capability of the code, which recall from its construction is guaranteed to correct up to $t = 8$ position errors.

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

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

2. The coefficients of each suffix are written consecutively in order across the rows of an array, and then taken in order down the columns of the array.

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

For example, with the four prefix/suffix pairs given previously, we first write the coefficients of the prefixes in order across the rows of 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 the array. This results in the following.

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

Next we write the coefficients of the suffixes in order across the rows of 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 the array. This results in the following.

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

Finally, combining the interleaved prefix and suffix coefficients gives the following full data for the QR code.

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

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. 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 of the code, is given in [6]. For QR codes of version 4, seven additional 0s are required. 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 a QR code, and how the code itself is constructed.

4 Data Placement and QR Code Construction

QR codes include several components placed in precise locations in a two-dimensional array. Figure 5 gives a summary of the placement of the various components of a QR code of version 4, which is essentially a 33×33 matrix.

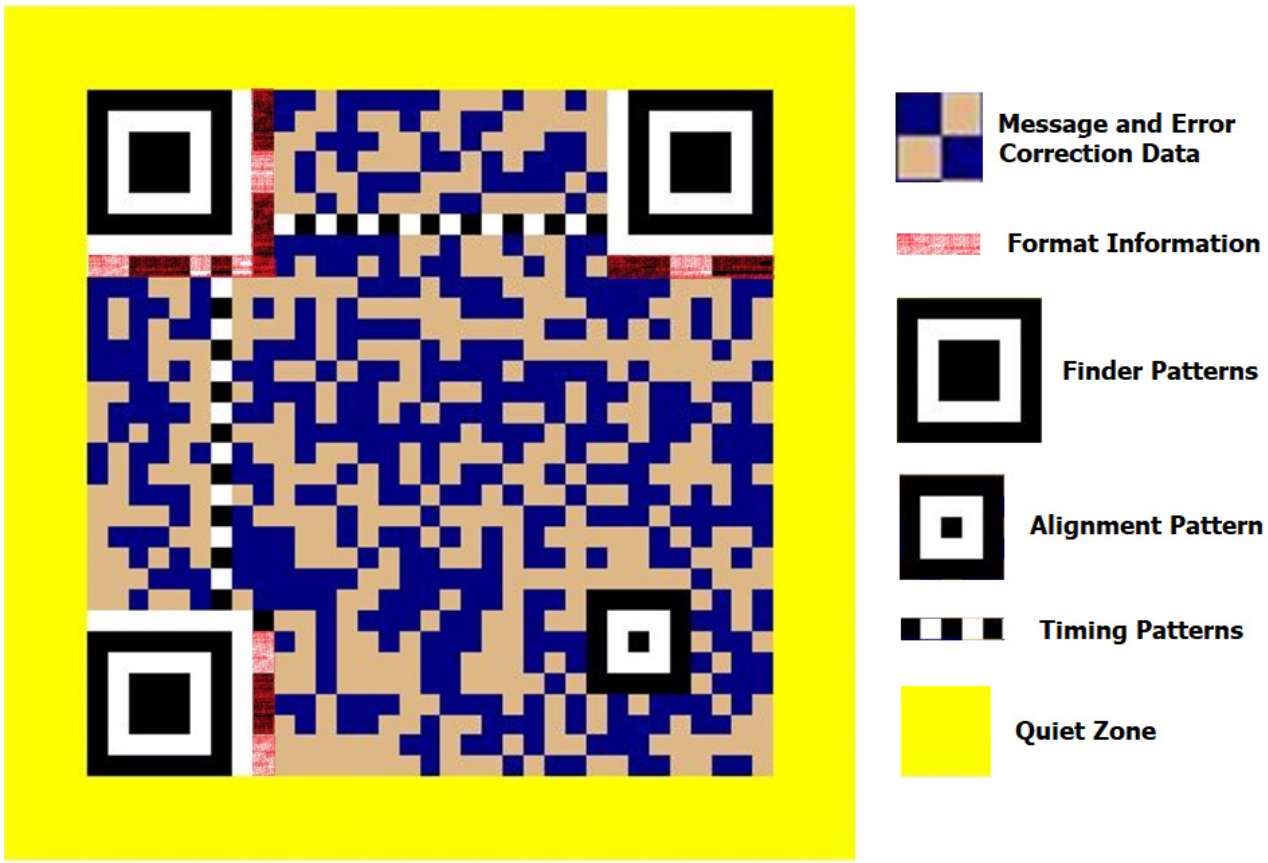

Figure 5: Component summary for a QR code of version 4.

In Figure 5, 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 bit, for which normally a 1 is represented by a darker color (dark blue, in this case), and a 0 is represented by a lighter color (light brown, in

this case). The data is placed in the image starting in the lower right corner. The regions shaded red in Figure 5 are reserved for the *format information*, which is a string of bits giving, among other things, the error correction level for the code (L, M, Q, or H). The format information also gives the *mask pattern*, which we will describe 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 array. *Separators*, which are shown as solid white rectangular strips in Figure 5, 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 are able to correctly identify and orient an image to be decoded. Finally, a *quiet zone* is included, which is indicated in yellow in Figure 5, and which is an area of lighter color designed to enclose the main QR code array. QR codes of versions 7–40 require more alignment patterns and an additional 18-bit block placed in two parts of the array containing information about the QR code version. More information about all of this can be found in [6].

For the message `https://atcm.mathandtech.org/`, Figure 6 shows how the data given by (4) is placed in a QR code of version 4.

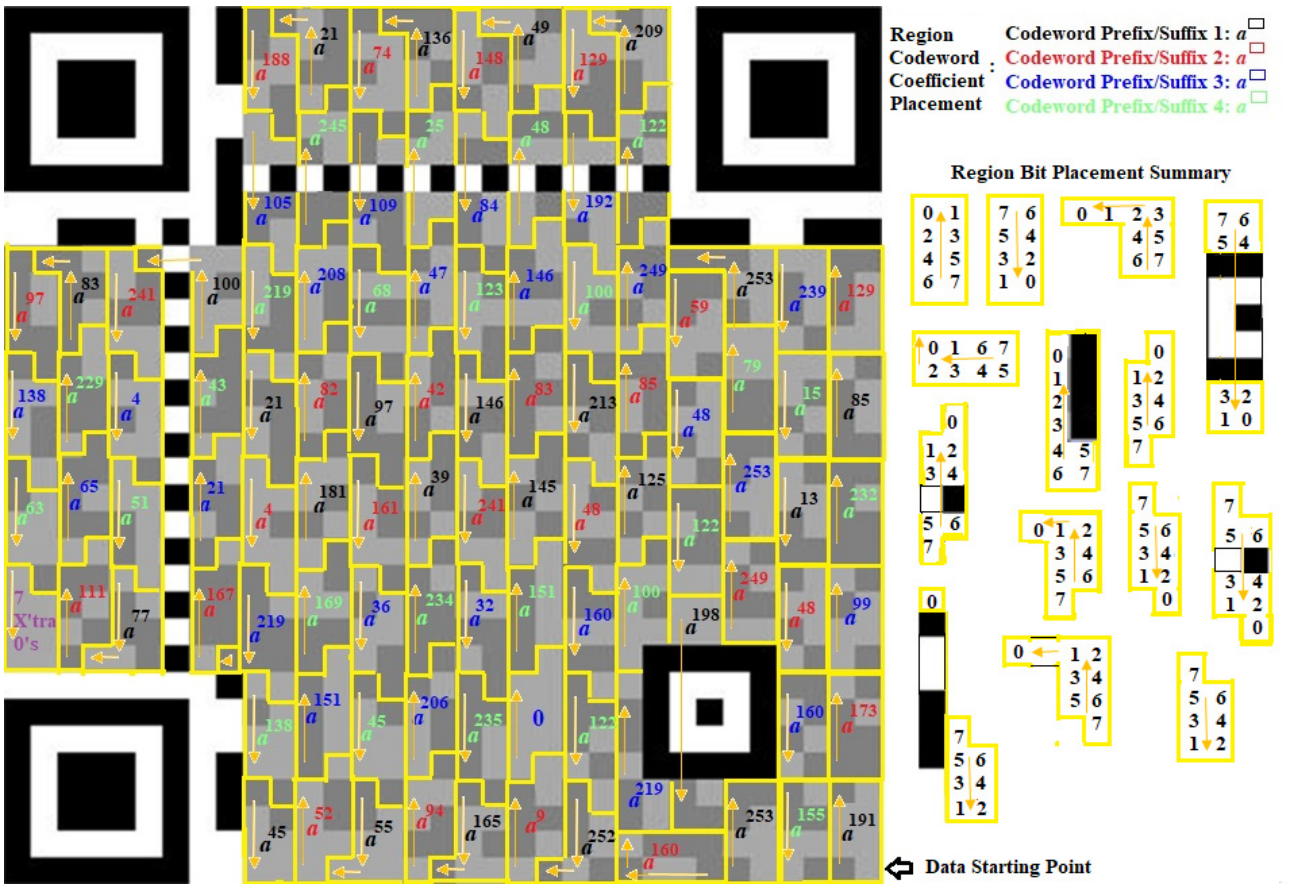

Figure 6: QR code data placement for the message `https://atcm.mathandtech.org/`.

Starting in the lower right corner of a QR code array, the data for each codeword prefix/suffix is placed in blocks in an up-and-down pattern. Each primitive power element in Figure 6 is color-coded to illustrate the codeword prefix/suffix with which it is associated, as well as how the interleaving spreads each codeword throughout the array. For each individual block, the eight bits representing the primitive power element are placed in a zig-zag pattern. The region bit placement summary section in Figure 6 indicates the order in which bits are placed within similarly shaped blocks, with the coefficients of each polynomial of maximum 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 in the region bit placement summary section in Figure 6.

After the data is placed into the QR code array, to make the data as readable as possible by QR code scanners, the process of *masking* is applied. When data in a QR code is masked, particular data squares, depending on their location in the array, 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 not flipped. 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 array. The different mask patterns and how the penalty scores are calculated for each are summarized in [6].

For the QR code of version 4 representing `https://atcm.mathandtech.org/`, the penalty score results in mask pattern 2. Numbering the 33 columns in this array from $0,1,\ldots,32$, mask pattern 2 flips the data squares in all columns with (column number) mod $3=0$. Figure 7 illustrates this QR code before and after masking, with the column numbers labeled where the masking occurs.

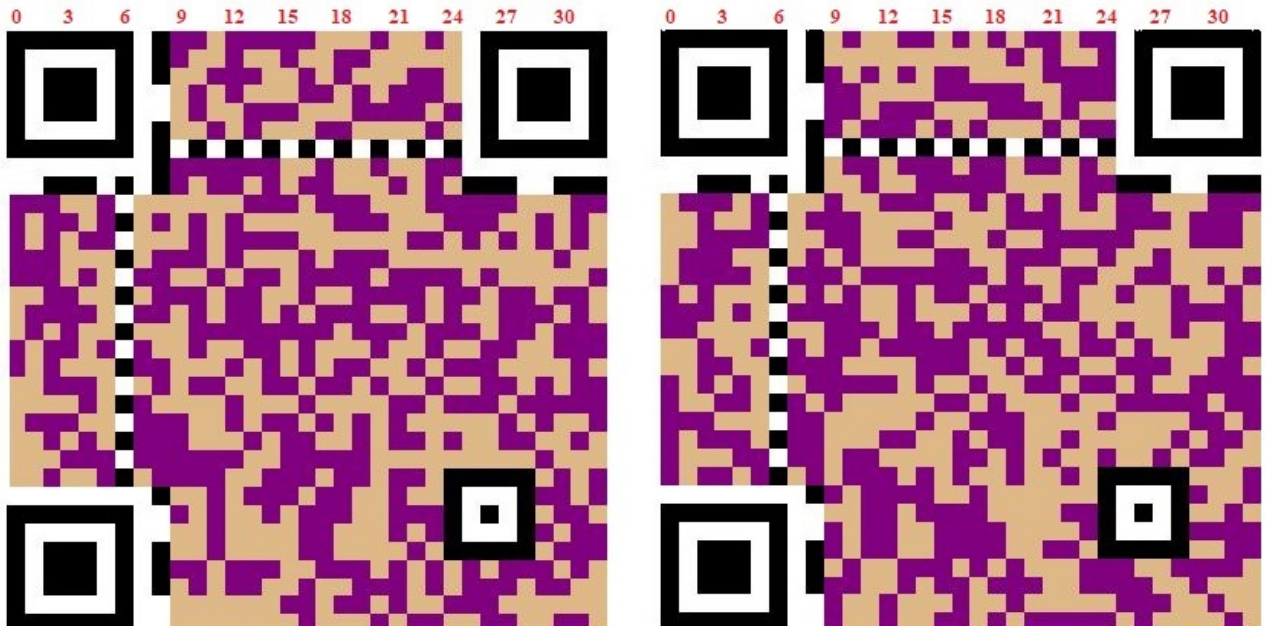

Figure 7: QR code example before (left) and after (right) masking.

After masking, a QR code is ready to be output. For generating QR codes, we will use another Maplet written by the authors, which is available for download at the link labeled [S5] in Section 6. This Maplet allows a user to input a message to be included, select an error correction level, and then generate the resulting QR code. The Maplet automatically generates the code with a specific version based on how much and what type of data is entered and the error correction level selected. Figure 8 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/`.

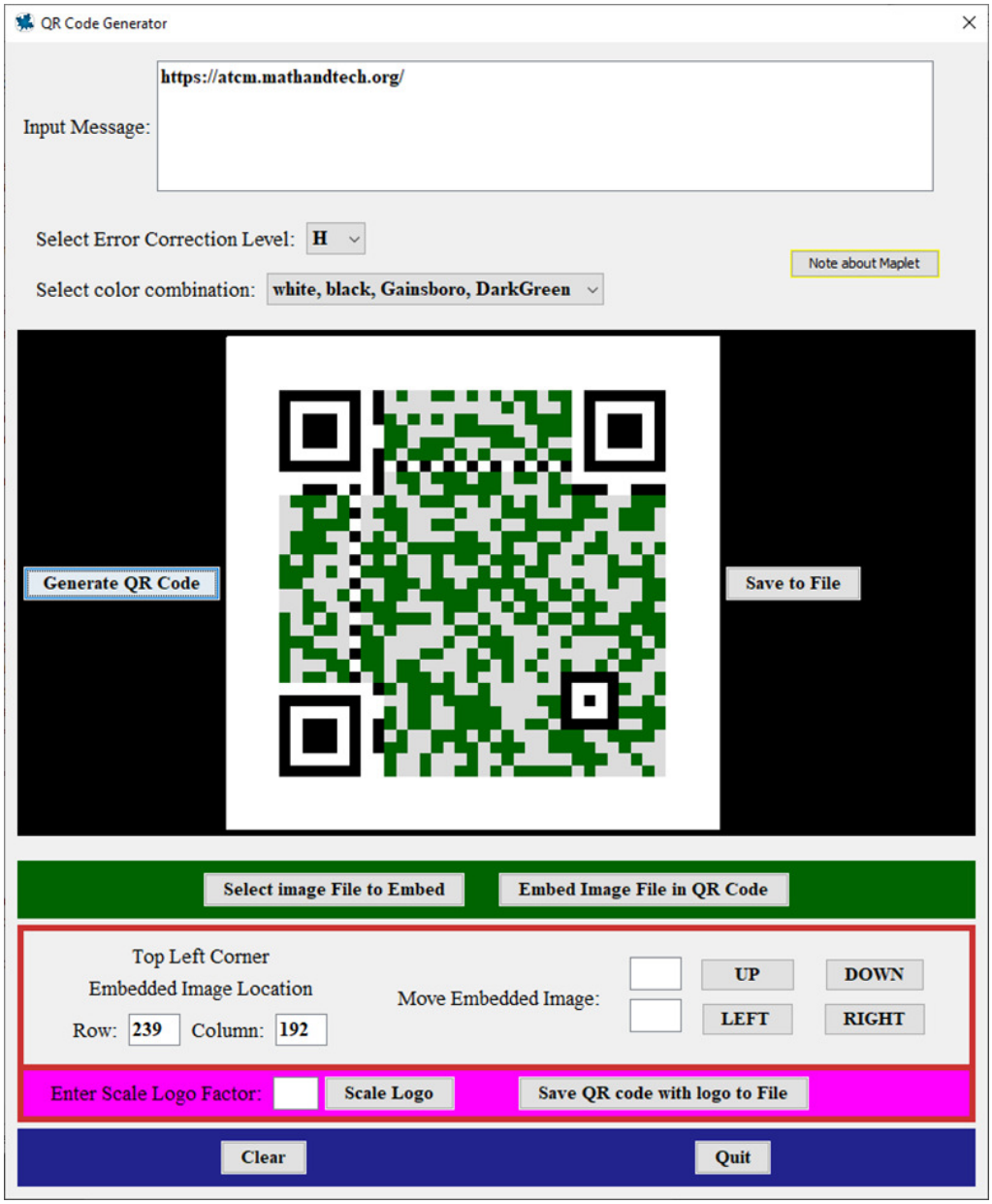

Figure 8: QR code example for the message `https://atcm.mathandtech.org/`.

Thanks to error correction, an image up to a certain size can be overlaid onto a QR code to help users identify its message contents. With error correction level H, even with 30% of the data squares

covered by an image, a QR code could still scan correctly. The Maplet illustrated in Figure 8 can accept an image file of a user's choice, and overlay it onto a generated QR code, with a placement and scaled size that can be designated by the user directly within the Maplet. Figure 9 shows the result with a logo overlaid onto a code that would still scan to `https://atcm.mathandtech.org/`.

Figure 9: QR code with an overlaid image.

5 Conclusion

In this paper, we presented some details about how information is formatted and placed into QR codes, with error correction provided via Reed-Solomon codes. Maplets written by the authors specifically for this paper were used to demonstrate this. These Maplets are available for download at [5]. The QR code generated in this paper was for specific information with a particular QR code version and error correction level. Additional explanations for how other information could be formatted and placed into QR codes with other versions and error correction levels can be found in [6].

This paper still leaves questions about QR codes unanswered though. For example, why does masking make codes easier for scanners to process? And does error correction in codes depend on where errors occur? More concretely, would the code shown in Figure 9 still scan to the correct information if the logo were placed at the bottom rather than in the middle, or increased in size by 20%? We will leave these and other questions about QR codes for the reader to explore, as our purposes for this paper were only to demonstrate a hands-on method for the use of non-trivial mathematics in a practical real-life application in which realistic examples could be produced, and to motivate readers to consider exploring the possibility of creating QR codes using their own favorite software system.

6 Supplementary Electronic Materials

[S1] Maplet that can be used to generate the elements in a finite field, with nonzero elements expressed as polynomials in a of maximum degree seven and with coefficients that are all either 0 or 1, as illustrated in Figure 1:
https://www.appstate.edu/~klimare/FiniteFieldGenerator.mw.

[S2] Maplet that can be used to convert bytes into their corresponding powers of a in the finite field used with QR codes, as illustrated in Figure 2:
https://www.appstate.edu/~klimare/FiniteFieldConversion.mw.

[S3] Maplet that can be used to divide a prefix by a generating polynomial $g(x)$ and form the prefix/suffix codeword pair in Reed-Solomon codes used in QR codes, as illustrated in Figure 3:
https://www.appstate.edu/~klimare/ReedSolomonCodewordGenerator.mw.

[S4] Maplet that can be used to correct errors in Reed-Solomon codes used in QR codes, as illustrated in Figure 4:
https://www.appstate.edu/~klimare/ReedSolomonErrorCorrector.mw.

[S5] Maplet that can be used to generate QR codes, as illustrated in Figure 8:
https://www.appstate.edu/~klimare/QRCode.mw.

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] Maplesoft, 2021. Maplesoft Media Release, March 10, 2021. Available at: https://www.maplesoft.com/company/news/releases/2021/2021-03-10-maple-2021-provides-even-more-tools-to-help-students-learn-math.aspx?L=E.

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

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