An Introduction to Galois Fields and Reed-Solomon Coding

James Westall
James Martin

School of Computing
Clemson University
Clemson, SC 29634-1906

October 4, 2010

1 Fields

A field is a set of elements on which the operations of addition and multiplication are defined. The operations are commutative ($ab = ba$ and $a+b = b+a$), associative ($a(bc) = (ab)c$, and $a + (b + c) = (a + b) + c$) and closed. Closure impliles that the sum and product of any two elements in the field are also elements of the field. A distributive law relates multiplication and addition: $a(b + c) = ab + ac$.

A field also has additive and multiplicative identities (0 and 1) such that $a + 0 = a$ and $1a = a$ for any element in the field. Elements of a field must have additive and multiplicative inverses. The additive inverse of a is an element b such that $a+b = 0$ and the multiplicative inverse of a is an element c such that $ac = 1$.

The existence of these inverses implicitly defines the operations of subtraction and division. The value of $a - c$ is $a + (-c)$ where $-c$ is the additive inverse of c. Similarly, the value of a/c is $a \times c^{-1}$ where c^{-1} is the multiplicative inverse of c.

Division by 0, the additive identity, is not defined. This implies that the additive and multiplicative identities are not the same ($0 \neq 1$), and also that $ab = 0$ if and only if either $a = 0$ or $b = 0$.

1.1 Finite fields

Well known fields having an infinite number of elements include the real numbers, $\mathbb{R}$, the complex numbers $\mathbb{C}$, and the rational numbers $\mathbb{Q}$. However, the integers under the usual arithmetic, $\mathbb{Z}$, do not constitute a field because only +1 and -1 have multiplicative inverses.

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Table 1: Addition table for $\mathbb{Z}_3$

×	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

Table 2: Multiplication table for $\mathbb{Z}_3$

Although the real, complex, and rational fields all have an infinite number of elements finite fields also exist. The symbol $\mathbb{Z}_p$ refers the integers $\{0, 1, .., p-1\}$ using modulo p arithmetic. $\mathbb{Z}_p$ is a field if and only if p is a prime number.

Regardless of whether or not p is prime each element x has an additive inverse with the value $p - x$. This follows from the fact that $(x + p - x = p = 0 \ \ mod \ \ p)$. If p is prime then each element x also has a multiplicative inverse y whose value is chosen so that $xy = 1 \ \ mod \ \ p$. There is no simple formula for computing y. For small p, a simple $O(p)$ algorithm is to multiply a by $2, 3, ...p - 1$ halting when the result is $modp$ is 1. For large values of p a variant of Euclid's greatest common divisor algorithm is more efficient.

If p is not a prime number, then it is possible to factor p as $p = ab$ where $1 < a, b < p$. Furthermore the product $ab = 0$ mod p. In this case a and b are called divisors of zero. Fields satisfy a cancellation law: $ac = ad$ implies $c = d$, and the following argument shows that a fields cannot have divisors of zero. Suppose $ab = 0$ for $a \neq 0$. Since $a0 = 0$ we can rewrite $ab = 0$ as $ab = a0$ and thus by the cancellation law $b = 0$. This shows that in any field if $ab = 0$, then either $a = 0$ or $b = 0$. Therefore, $\mathbb{Z}_p$ for p not prime is not a field.

Tables 1, 2, 3, and 4 illustrate addition and multiplication in $\mathbb{Z}_3$ and $\mathbb{Z}_7$.

For any such finite field it will always be the case the each row of the addition table

+	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

Table 3: Addition table for $\mathbb{Z}_7$

×	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

Table 4: Multiplication table for $\mathbb{Z}_7$

is a permutation of the values $\{0, 1, ..., p-1\}$ and each row of the multiplication table except the first row will also be a permutation of the elements of the field. As noted previously, a value of 1 in the multiplication table identifies a pair of multiplicative inverses.

1.2 Galois fields

If p is a prime number, then it is also possible to define a field with p^m elements for any m. These fields are named for the great French algebraist Evariste Galois who was killed in a duel at age 20. They have many applications in coding theory.

The fields, denoted $GF(p^m)$, are comprised of the polynomials of degree $m-1$ over the field $\mathbb{Z}_p$. These polynomials are expressed as $a_{m-1}x^{m-1} + ... + a_1x^1 + a_0x^0$ where the coefficients a_i take on values in the set $\{0, 1, ..., p-1\}$.

When employed in coding applications p is commonly 2 and thus the coefficients $\{a_0, ..., a_{m-1}\}$ are taken from the binary digits $\{0, 1\}$. In coding applications, for $m \leq 32$, it is common to represent an entire polynomial in $GF(2^m)$ as a single integer value in which individual bits of the integer represent the coefficients of the polynomial. The least significant bit of the integer represents the a_0 coefficient.

1.2.1 Polynomial arithmetic in $GF(2^m)$

Addition and multiplication of polynomial coeffcients, *but not the polynomials themselves* in the field $GF(2^m)$ are defined by the rules of $\mathbb{Z}_2$. These are shown in tables 5 and 6. It can be observed that addition is defined by the *exclusive or* operation and multiplication by the *and* operation.

+	0	1
0	0	1
1	1	0

Table 5: Addition table for $\mathbb{Z}_2$

×	0	1
0	0	0
1	0	1

Table 6: Multiplication table for $\mathbb{Z}_2$

These operations are used in manipulating the coefficients during multiplication and addition of polynomials, but the basic algorithms used in adding and multiplying polynomials over the integers remain applicable.

1.2.2 Polynomial addition in $GF(2^m)$

To add two or more polynomials, for each power of x present in the summands, just add the corresponding coefficients modulo 2. If a particular power appears an odd number of times in the summands it will have a coefficient of 1 in the sum. If it appears an even number of times or does not appear at all, it will have a coefficient of 0 in the sum. For example,

$$(x^2 + 1) + (x + 1) + (x^2 + x + 1) = 1.$$

Similarly,

$$(x^2 + x + 1)(x + 1) = x^3 + x^2 + x + x^2 + x + 1 = x^3 + 1.$$

Note that the polynomials of degree $m - 1$ are closed under polynomial addition. The sum is always a polynomial of degree no more than degree $m-1$. Furthermore, because of the *xor* method of addition, each polynomial is its own additive inverse.

1.2.3 The generating polynomial of $GF(2^m)$

The polynomials of degree $m-1$ are not closed under multiplication. For example, x^{m-1} times x^{m-1} is x^{2m-2}. Thus for all $m > 1$, the degree of the product may exceed than $m-1$.

Our objective is to build a field of 2^m elements in which the operations of addition and multiplication are based upon polynomial addition and multiplication. Thus, we need a mechanism for ensuring that multiplication is closed. To do this we resort again to modular arithmetic.

A generating polynomial for $GF(p^m)$ is a degree m polynomial that is irreducible over $\mathbb{Z}_p$. This simply means that it cannot be factored. For example x^3+1 is not irreducible over $\mathbb{Z}_2$ because it can be factored as $(x^2+x+1)(x+1)$. Note that this factorization works only over $\mathbb{Z}_2$ and not $\mathbb{Z}$.

1.2.4 Polynomial addition and multiplication in $GF(2^3)$

If an irreducible polynomial $g(x)$ can be found, then polynomial multiplication can be defined as standard polynomial multiplication modulo $g(x)$. That is, to compute the product $a(x)b(x)$ first compute $p(x) = a(x)b(x)$ and then transform $p(x)$ back into the set of polynomials of degree $m-1$ by taking the remainder when $p(x)$ is divided by $g(x)$. If $p(x)$ already has degree no larger than $m-1$, then the remainder is simply $p(x)$, but if this is not the case, the remainder is guaranteed to have degree no higher than $m-1$.

Note that the requirement that $g(x)$ be irreducible is implicit in this definition of multiplication. Suppose $g(x)$ is not irreducible. Then there exist two polynmials $a(x)$ and $b(x)$ such that $g(x) = a(x)b(x)$. However, $g(x) = 0 mod g(x)$. Hence $a(x)$ and $b(x)$ are divisors of zero, and it has previously been shown that fields may not contain zero divisors.

It is the case that both x^3+x+1 and x^3+x^2+1 are irreducible over $\mathbb{Z}_2$. Therefore, either one can be used to generate a field of 8 elements representing polynomials of degree 2. The mapping of coefficients to numbers for x^3+x+1 is given in the table 7.

We will consider $g(x) = x^3+x+1$ in the following examples. Our objective is to generate addition and multiplication tables for $GF(2^3)$ that are analogous to

Dec	Bin	Poly
0	000	0
1	001	1
2	010	x
3	011	$x+1$
4	100	x^2
5	101	x^2+1
6	110	x^2+x
7	111	x^2+x+1

Table 7: Polynomial representation

those we developed for the field $\mathbb{Z}_7$. Addition can be performed by inspection. For example $3 + 5$ represents $(x + 1) + (x^2 + 1) = x^2 + x$ which is 6. So $3 + 5 = 6$. An equivalent approach is to remember that polynomial addition over $\mathbb{Z}_2$ is defined by the xor operation. So $3 + 5 = 011\ xor\ 101 = 110 = 6$. Code to implement addition over $GF(2^m)$ for m less than or equal to the word size of the computer is trival.

```
int gf_add(
int v1,
int v2)
{
   return(v1 ^ v2);
}
```

The $gf\_add()$ function was used to produce the following addition table.

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

Table 8: Addition table for $GF(2^3)$

Multiplication cannot be represented so simply. Consider the problem of multiplying 5×6. This is $(x^2 + 1)(x^2 + x) = x^4 + x^3 + x^2 + x$. Since this result has terms of higher order than 2, it is necessary to reduce the result modulo $g(x)$. This can be done via long division as shown. The reduction of $x^4 + x^3 + x^2 + x$ is $x + 1$.

```
                1 1
          ----------------
1 0 1 1 | 1 1 1 1 0
          1 0 1 1
          -------
            1 0 0 0
            1 0 1 1
            -------
              0 1 1  <-- (x + 1)
```

We can consider problem of modular reduction of degree from an alternate but equivalent perspective. Suppose we apply the modular mapping to x^3 itself. That is, we wish to know what is the remainder when x^3 is divided by $x^3 + x + 1$.

```
                1
          ----------------
1 0 1 1 | 1 0 0 0
          1 0 1 1
          -------
            0 1 1  (x + 1)
```

Here we see that the remainder is $(x + 1)$. In general the remainder when the high order term of the generator is divided by the generator itself will always be comprised of the generator absent its high order term.

This also follows from the fact that $g(x) = q(x) + r(x)$. The quotient with the high order term is divided by the generator is always the high order term, and the remainder is thus necesssarily the remainder of the terms.

Thus, when an x^3 term appears in the product in may be replaced by $x + 1$. Since $x^4 = x(x^3) = x(x+1) = x^2+x$, an x^4 term in the product can be replaced by x^2+x. This reduction strategy may be used manually in place of long division. Applying the strategy to the answer we found above we observe that $x^4 + x^3 + x^2 + x = x^2 + x + x + 1 + x^2 + x = x + 1$.

The full process of multiplication followed by reduction can be coded reasonably succinctly as:

```
int gf_mult(
```

```
int m,       // GF(2 ^ M)
int poly,    // low order terms of g(x)
int v1,
int v2)
{

   int prod = 0;
   int k;
   int mask;

/* Multiply phase */

   for (k = 0; k < m; k++)
   {
      if (v1 & 1)
      {
         prod ^=  (v2 << k);
      }
      v1 >>= 1;
      if (v1 == 0)
         break;
   }

/* Reduce phase */

   mask = 1 << m;
   mask <<= m - 2;

   for (k = m - 2; k >= 0; k--)
   {
      if (prod & mask)
      {
         prod &= ~mask;
         prod ^= (poly << k);
      }
      mask >>= 1;
   }
```

```
   return(prod);
}
```

The multiply phase models manual polynomial multiplication by performing successive adds of shifted terms of the multiplicand if the corresponding coefficient in of the multiplier is not 0. The reduce phase models the long division algorithm, or, equivalently, the replacement algorithm. The $gf\_mult()$ function was used to generate the following multiplication table.

×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	3	1	7	5
3	0	3	6	5	7	4	1	2
4	0	4	3	7	6	2	5	1
5	0	5	1	4	2	7	3	6
6	0	6	7	1	5	3	2	4
7	0	7	5	2	1	6	4	3

Table 9: Multiplication table for $GF(2^3)$

1.3 Fast multiplication using logarithms

Any element of $GF(2^m)$ may be used as a logarithmic base. We will use base 2 in the following example in $GF(2^3)$. As usual $log_2(k) = j$ if and only if $2^j = k$, and then it is also true that $log_2^{-1}(j) = k$. It is also the case that, as usual, $log_2(0)$ is undefined and for a more subtle reason so is $log_2^{-1}(2^m - 1)$. The latter follows from the fact that in $GF(2^m)$ $j^{2^m-1} = 1$ for any non-zero j and $log_2(1)$ has already been defined to be 0.

The following simple algorithm can be used to construct the table of logarithms shown below. As expected, the inverse logarithm table is simply a table of powers of 2.

```
/* Construct the log and inverse log table */

   gf->ilog[0] = 1;
   prod = 2;
   for (i = 1; i < gf->size - 1; i++)
   {
      gf->ilog[i] = prod;
      gf->log[prod] = i;
```

```
    prod = gf_mult(gf, prod, 2);
}
```

i	0	1	2	3	4	5	6	7
$log_2(i)$	-	0	1	3	2	6	4	5
$log_2^{-1}(i)$	1	2	4	3	6	7	5	-

Table 10: Logarithm tables for $GF(2^3)$

The following examples show how to use the tables to perform multiplication. As usual $ab = log_2^{-1}(log_2(a) + log_2(b))$. Thus, to multiply 3×4, we first see that $log_2(3) = 3$ and $log_2(4) = 2$. To add the logarithms, we use normal integer addition *not* xor because we are operating on exponents and not elements of the field, and we find $2 + 3 = 5$. Finally, we see $log_2^{-1}(5) = 7$. We can verify from table 9 that 7 is the correct answer.

Since we are using integer arithmetic to add the exponents, we can get a value too large to use as an index into the inverse logarithm table. As previously noted $n^{2^m-1} = 1$ for all values, n, in the Galois field. Thus successive exponentiation repeats itself cyclically with a period of $2^m - 1$. Therefore, when the sum of exponents is $\geq 2^m - 1$, then $2^m - 1$ is subtracted from the sum. This maps it into the correct range to be used as a table lookup key.

For example, in computing 5×6 we obtain logarithms 6 and 4 whose sum is 10. Subtracting 7 from 10 yields 3. The inverse log of 3 is 3 which is the correct product.

2 Reed-Solomon Codes

Reed-Solomon codes can be used to perform a form of forward error correction *FEC* in computer networks. The specific type of correction is called *erasure correction*. The objective of erasure correction is to recover from loss of entire packets. They can be used in conjunction with traditional error detecting cyclic codes by simply treating packets with errors as lost.

The encoding and decoding employs arithmetic in the domain $GF(2^m)$. We will use $m = 3$ as an example. $GF(2^3)$ consists of 8 elements. Each element is a polynomial of degree 2 and is encoded in an 3 bit value. We will use the generator $g(x) = x^3 + x + 1$.

The value, m, is the *word size* of the encoding. Each packet is subdivided into words of length m bits and check values must be computed for each word. Therefore, in the *real world* word sizes of 8, 16, or 32 instead of 3 would normally be used.

The packet stream that is actually transmitted consists of FEC groups containing both data packets and check packets used in reconstructing lost data packets. Data packets are fixed size and check packets have the same length as data packets.

A FEC group consists of n data and k check packets where $n+k <= 2^m$. Successful receipt of any n packets from the combination data and check packets is sufficient to permit reconstruction of the n data packets.

The algorithm requires an encoding/decoding matrix of $n+k$ rows and n columns having the properties that (1) the first n rows constitute an $n \times n$ identity matrix, I, and (2) any n of the $n+k$ rows are linearly independent. Property (2) ensures that any collection of exactly n rows constitutes an invertible $n \times n$ matrix.

The required matrix is derived from Vandermonde matrix which is known to possess property (2). For arbitrary, n and m this matrix has the following form. The fact that the largest value in $GF(2^m)$ is $2^m - 1$ necessarily constrains the number of rows and columns in the matrix $(n+k)$ to be less than or equal to 2^m.

$$\begin{bmatrix} 0^0 & 0^1 & 0^2 & ... & 0^{(n-1)} \\ 1^0 & 1^1 & 1^2 & ... & 1^{(n-1)} \\ 2^0 & 2^1 & 2^2 & ... & 2^{(n-1)} \\ ... & ... & ... & ... & ... \\ (2^m-1)^0 & (2^m-1)^1 & (2^m-1)^2 & ... & (2^m-1)^{(n-1)} \end{bmatrix}$$

For the specific choices of $m = 3$, $n = 3$ and $k = 5$, the Vandermonde matrix has the following form.

$$V = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 5 \\ 1 & 4 & 6 \\ 1 & 5 & 7 \\ 1 & 6 & 2 \\ 1 & 7 & 3 \end{bmatrix}$$

This matrix is known to possess property(2) but clearly does not possess property (1). Nevertheless, by a series of linear transformations in which a multiple of one column is added to another, we can obtain property (1) while preserving property (2). For example, if we add column 1 to column 0 and column 2, then row 1 is transformed to have the desired values 0 1 0. To fix up row 2 of the matrix we multiply column 2 by the multiplicative inverse of 4 and then add column 2 to 1 and 2× column 2 to column 1. This process can be continued until the first n rows constitute an identity matrix. We will call this transformed matrix D.

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 6 \\ 4 & 3 & 2 \\ 5 & 2 & 2 \\ 5 & 3 & 4 \\ 4 & 2 & 4 \end{bmatrix}$$

2.1 The encoding algorithm

For simplicity we will assume that each data packet consists of only one 3 bit word. If an actual packet consisted of 12,000 bits, the process described below would have to be repeated 4,000 times, once for each data word in the packet.

The bottom k rows of the transformed Vandermonde matrix, D, constitute the encoding matrix E that is used to create the k check packets.

$$E = \begin{bmatrix} 1 & 1 & 6 \\ 4 & 3 & 2 \\ 5 & 2 & 2 \\ 5 & 3 & 4 \\ 4 & 2 & 4 \end{bmatrix}$$

The matrix E has dimension $k \times n$ where k is the number of check packets and n is the number of data packets. When the $n \times 1$ vector of data words is multiplied by E as shown an $k \times 1$ vector of check values is produced. In this example we assume that the 3 data words have the values $\{4, 5, 6\}$.

$$\begin{bmatrix} 1 & 1 & 6 \\ 4 & 3 & 2 \\ 5 & 2 & 2 \\ 5 & 3 & 4 \\ 4 & 2 & 4 \end{bmatrix} \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \\ 4 \\ 3 \\ 2 \end{bmatrix}$$

Each of the 8 "packets" must also carry an identifier that allows the recipient to determine exactly which packets of a FEC group have been received. The values contained in the 8 "packets" that are sent are thus:

$$\{(0, 4), (1, 5), (2, 6), (3, 3), (4, 5), (5, 4), (6, 3), (7, 2)\}.$$

2.2 The error correction algorithm

The correction algorithm is then straightforward. Suppose a collection of n packets including both data and check packets have been received. We extract the n rows of the D matrix corresponding the n recieved packets. We call this $n \times n$ matrix D' and by property (2) it is invertible over $G(2^m)$. We invert D' to obtain D'^{-1}. Then the product of D'^{-1} and the received mixture of data words and check words recovers the data words $d_0, ...d_{n-1}$.

Continuing with our numeric example, suppose only packets 3, 4, and 5 containing the first three check values $\{3, 5, 4\}$ are received. The receiver doesn't know what the data values are so we will call them d_0, d_1, and d_2. Nevertheless the receiver *must* know what values of m, n and k are in use. Thus, the receiver can construct a matrix D' consisting of rows of the modified Vandermonde matrix that correspond to the packets that were received.

$$D' = \begin{bmatrix} 1 & 1 & 6 \\ 4 & 3 & 2 \\ 5 & 2 & 2 \end{bmatrix}$$

Because the receiver also knows the algorithm by which the check packets were constructed, the receiver knows that:

$$\begin{bmatrix} 1 & 1 & 6 \\ 4 & 3 & 2 \\ 5 & 2 & 2 \end{bmatrix} \times \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \\ 4 \end{bmatrix}$$

Inverting D' yields D'^{-1}

$$D'^{-1} = \begin{bmatrix} 5 & 6 & 2 \\ 5 & 7 & 3 \\ 3 & 3 & 3 \end{bmatrix}$$

Multiplying the received check values by D'^{-1} recovers the original values.

$$\begin{bmatrix} 5 & 6 & 2 \\ 5 & 7 & 3 \\ 3 & 3 & 3 \end{bmatrix} \times \begin{bmatrix} 3 \\ 5 \\ 4 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$$

2.3 Analysis of coding overhead

Three parameters determine the cost of the FEC procedure. The value of m defines the Galois field as $GF(2^m)$ and also defines the word size of the encoding to be m.

For example, a packet having 2^{10} bits, consists of 256 words over $GF(2^4)$ but only 64 words of $GF(2^{16})$.

The size of a FEC group is $n+k$ where n is the number of data packets in a group and k is the number of check packets. The matrix D consists of $n+k$ rows and n columns. The encoding matrix E that is used to generate the checksums has dimension k rows and n columns.

2.3.1 Check packet generation

Thus the generation of each word of the checksums requires multiplication of an $k \times n$ matrix times a vector of length n. This requires n multiplies and $n-1$ sums per row or a total of kn multiplies and $kn-1$ sums *per packet word*.

If multiplication is implemented as logarithm table lookups followed by addition and inverse logarithm lookup, multiplication can be done in constant time for values of M for which tables of size 2^m are practical. In this case each multiplication can be done as one or two integer additions and two table lookups.

Note that doubling of both k and n which keeps the fraction of FEC packets constant *quadruples* the cost of generating the checksums. This is a strong motivator for the use of an FEC group interleaver as opposed to a large value of n for for additional robustness against error bursts.

2.3.2 Recovery from errors

Error recovery begins with the inversion of the $n \times n$ matrix comprised of rows from the D matrix corresponding to the n packets in the FEC group that were actually received. In the worse case this inversion is $O(n^3)$. In practice the cost of the inversion is proportional to the number of check rows that must replace identity rows in the decode matrix. When the inversion has been completed, it is necessary to multiply the received vector of n words by the $n \times n$ inverse matrix for each word in the packet. As noted above this requires $O(n^2)$ multiplies and sums.

2.4 Example

Assuming $GF(2^{16})$, $n = 6$, and $k = 2$, generation of the 2 check packets requires $64 \times 2 \times 6$ multiplies and a similar number of additions. Decoding requires no overhead if no packet is lost. Furthermore, since there are only 2 check words in each each FEC group of 8 words, at most two rows need be transformed by column operations in the inversion process.