Applications of Finite Sets

Texas A&M Final Project
Jeremy Knight
March 29, 2012

Introduction

In many mathematical applications, it is valuable to consider finite sets to control outcomes, allow approximation, and provide valuable structure to processes and algorithms. The goal of this paper is to explore the impact of finite structures in the study and application of cryptography, partial differential equations (PDEs) and statistics. We will begin by examining the important role that finite fields have in the Advanced Encryption Standard (AES): Rijndael, and error-correcting codes. In the application of PDEs, we often seek a numerical solution through the use of finite approximations when analytic solutions are not available. The field of Differential Galois Theory also employs the use of finite fields extensively. Finally, we will see the application of finite groups in the construction of statistical studies using 2^k factorial designs.

Galois Fields and Cryptography

What are Finite fields and Galois Fields?

A field is a set that

1. is associative, commutative, and distributive for addition and multiplication,
2. contains an additive identity element (zero) and multiplicative identity element (unity),
3. contains an additive inverse for all elements, and
4. contains a multiplicative inverse for all non-zero elements.

A *finite field* is field that has a finite number of elements. The number of elements in a field is called the order.

The order of a finite field must be of the form p^n for some prime number p and integer $n > 1$ [TW]. The standard notation for a field with p^n elements is $GF(p^n)$ where the "GF" represents "Galois Field" in honor of Evariste Galois who was instrumental in the foundations of group theory and Galois theory. In cryptographic systems, it is common to apply the field $GF(2^n)$ and work modulo 2.

We will define $\boldsymbol{Z}_p[X]$ to be the set of polynomials with coefficients mod p. We will typically work with polynomials in $\boldsymbol{Z}_2[X]$. When working with these polynomials mod 2, we will often represent it in binary notation. For example,

$$\mathrm{X}^8 + \mathrm{X}^4 + \mathrm{X}^3 + \mathrm{X} + 1 \rightarrow 100010011$$

is the polynomial notation and binary notation of an important polynomial for the Advanced Encryption Standard (AES). The binary digits $b_8b_7b_6b_5b_4b_3b_2b_1b_0$ are the coefficients of $b_8X^8 + \cdots + b_1X^1 + b_0$.

Arithmetic of $GF(2^m)$

Operations in $Z_2[X]$

Addition and Subtraction

Addition is the XOR operation, denoted with the symbol $\oplus$, modulo 2 giving us

$$1 \oplus 1 = 0, \qquad 1 \oplus 0 = 1, \qquad 0 \oplus 0 = 0.$$

Example Add $(X^8 + X^4 + X^3 + X + 1) + (X^8 + X^7 + X^3 + 1)$ as a polynomial and in binary notation.

$$(X^8 + X^4 + X^3 + X + 1) + (X^8 + X^7 + X^3 + 1) = X^7 + X^4 + X$$

where the X^8, X^3, and1 terms have vanished since the coefficients are $2 \equiv 0 (\text{mod } 2)$.
In binary notation this sum is

$$\{100011011\} \oplus \{110001001\} = \{010010010\}.$$

Furthermore, subtraction of polynomials in $Z_2[X]$ is equivalent to addition since $-1 \equiv 1 (\text{mod } 2)$. Hence,

$$a - b \equiv a + (-1)b \equiv a + b \ (\text{mod } 2)$$

for all $a, b \equiv 0 \ or \ 1 \ (\text{mod } 2)$.

MATLAB code can be found in the Appendix for binxor.m which will compute the XOR of two binary string inputs.

Multiplication

Multiplication of polynomials in $Z_2[X]$ is done in the normal manner applying distribution. However, some powers of X will vanish due to the fact that $1 + 1 \equiv 0 \ (\text{mod } 2)$.

Example Compute $(X^2 + X + 1)(X + 1)$ as a polynomial and in binary notation.

$$(X^2 + X + 1)(X + 1) = (X^3 + X^2 + X) + (X^2 + X + 1) = X^3 + 1,$$

or in binary notation this is

$$\{0111\} \cdot \{0011\} = \{1110\} \oplus \{0111\} = \{1001\}.$$

It is useful to observe that in binary notation, multiplying by a 1 in the b_i digit shifts the bits to the left i decimal places and fills in i zeros to the right. Hence, binary multiplication becomes of series of bit shifts followed by XOR addition.

Example Compute the product of the 8-bit binary numbers $\{10011011\} \cdot \{00100101\}$

$\{10011011\} \cdot \{00100101\}$

$= \{10011011\} \cdot \{00100000\} \oplus \{10011011\} \cdot \{00000100\} \oplus \{10011011\} \cdot \{00000001\}$
$= \{1001101100000\}$
$\oplus \{0001001101100\}$
$\oplus \{0000010011011\}$
$= \{1000110010111\}$

MATLAB code can be found in the Appendix for binmult.m which will compute the binary product of two binary string inputs.

Division

Division of polynomials in $Z_2[X]$ can be done by long division.
Example Use long division to divide $X^4 + 1$ by $X^2 + X + 1$

$$
\begin{array}{r}
X^2 + X \\
X^2 + X + 1 \;\big)\; X^4 + 1 \\
\underline{X^4 + X^3 + X^2} \\
X^3 + X^2 + 1 \\
\underline{X^3 + X^2 + X} \\
X + 1
\end{array}
$$

Thus, we see that $X^4 + 1 = (X^2 + X)(X^2 + X + 1) + (X + 1)$, where $(X + 1)$ is the remainder. Equivalently, we can say that

$$X^4 + 1 \equiv X + 1 (\text{mod}\, X^2 + X + 1).$$

An alternative method for division is to find the inverse of the polynomial and multiply

Example Use binary notation to divide $X^4 + 1$ by $X^2 + X + 1$

$$
\begin{aligned}
\frac{10001}{111} &= \frac{11100 \oplus 1101}{111} \\
&= 100 \oplus \frac{1101}{111} \\
&= 100 \oplus \frac{1110 \oplus 11}{111} \\
&= 100 \oplus 10 \oplus \frac{11}{111} \\
&= 110 \oplus \frac{11}{111}
\end{aligned}
$$

We see that the binary number $110\frac{11}{111}$ corresponds to the polynomial $X^2 + X + \frac{X+1}{X^2+X+1}$. In a binary modulus, we could say $10001 \equiv 11 (\text{mod}\, 111)$.

MATLAB code can be found in the Appendix for bindiv.m which will compute the binary product of two binary string inputs.

Irreducible Polynomials

The most common method for constructing a finite field with p^n elements for prime p and integer $n \geq 1$ is to create a finite set of polynomials mod $P(X)$ where $P(X)$ is an irreducible polynomials. An irreducible polynomial is one that is a member of $\boldsymbol{Z}_p[X]$ that does not factor into polynomials of lower degree mod p. Consider the possible 2nd degree polynomials in $\boldsymbol{Z}_2[X]$. These are

$$X^2, \qquad X^2 + 1, \qquad X^2 + X, \qquad X^2 + X + 1.$$

Three of these can be factored into polynomials in $\boldsymbol{Z}_2[X]$ as

$$
\begin{aligned}
X \cdot X &= X^2 \\
X \cdot (X + 1) &= X^2 + X \\
(X + 1) \cdot (X + 1) &= X^2 + 1
\end{aligned}
$$

For small values of n, we can check all products of polynomials in $Z_{n-1}[X]$ to find a polynomial that is irreducible.

To demonstrate, we will seek an irreducible polynomial of $GF(2^3)$ of degree 3. We first consider the nonzero elements of $GF(2^3)$

$$X^2, \qquad X^2 + X, \qquad X^2 + 1, \qquad X^2 + X + 1, \qquad X, \qquad X + 1, \qquad 1.$$

We will check all products that produce a polynomial of degree 3.

$$X^2(X) = X^3$$
$$X^2(X+1) = X^3 + X$$
$$(X^2+X)(X) = X^3 + X^2$$
$$(X^2+X)(X+1) = X^3 + X^2 + X^2 + X = X^3 + X$$
$$(X^2+1)(X) = X^3 + X$$
$$(X^2+1)(X+1) = X^3 + X^2 + X + 1$$
$$(X^2+X+1)(X) = X^3 + X^2 + X$$
$$(X^2+X+1)(X+1) = X^3 + X^2 + X + X^2 + X + 1 = X^3 + 1$$

We observe that the only $Z_2[X]$ polynomials of degree 3 that are not produced above are

$$f(X) = X^3 + X^2 + 1,$$

and

$$f(X) = X^3 + X + 1.$$

Thus, these are irreducible polynomials in $GF(2^3)$.

Multiplicative Inverse

When working with $GF(2^m)$ modulo an irreducible polynomial, all polynomials have a multiplicative inverse. That is, for $a(X) \in GF(2^m)$ and irreducible polynomial $m(X) \in GF(2^m)$, the Chinese Remainder Theorem tells us that there exists polynomials $b(X), c(X) \in GF(2^m)$ such that

$$a(X)b(X) + m(X)c(X) = 1.$$

This gives us

$$a(X)b(X) \equiv 1 \ \big(\mathrm{mod}\, m(X)\big),$$

so

$$a^{-1}(X) = b(X) \ \big(\mathrm{mod}\, m(X)\big).$$

To find this inverse we will use the extended Euclidean algorithm. Consider

$$GF(2^3) = Z_2[X] \ (\mathrm{mod}\, X^3 + X + 1)$$

using the irreducible polynomial $m(X) = X^3 + X + 1$ we found previously. For example, we will find the inverse of $a(X) = X^2 + X + 1$ in $GF(2^3)$. To find the inverse, we first use the Euclidean Algorithm to find the greatest common denominator (which should be 1).

$$X^3 + X + 1 = (X+1)(X^2 + X + 1) + (X)$$
$$X^2 + X + 1 = (X+1)(X) + 1$$

The last remainder is 1, which tells us that the greatest common divisor is 1. This gives us quotients $\{(X+1), (X+1)\}$. Working backwards, we get

$$\begin{aligned} 1 &= (X^2 + X + 1) + (X+1)(X) \\ &= (X^2 + X + 1) + (X+1)\big(X^3 + X + 1 + (X+1)(X^2 + X + 1)\big) \\ &= (1 + (X+1)^2)(X^2 + X + 1) + (X+1)(X^3 + X + 1) \\ &= (X^2)(X^2 + X + 1) + (X+1)(X^3 + X + 1). \end{aligned}$$

Hence,

$$(X^2)(X^2 + X + 1) \equiv 1 \ (\mathrm{mod}\, X^3 + X + 1)$$

and

$$a^{-1}(X) \equiv X^2 \ (\mathrm{mod}\, X^3 + X + 1).$$

We can confirm this using the algorithms binmult.m and bindiv.m described above. First we multiply the binary forms of $a(X)$ and $a^{-1}(X)$

```
>> binmult('100','111')
        ans = 11100
```

We then use bindiv.m to check for a remainder of 1 when divided by $X^3 + X + 1 \rightarrow 1011$

```
>> bindiv('11100','1011',1)
11100/1011=
0+(10110+1010)/1011
10+1010/1011
10+(1011+1)/1011
11+1/1011
Polynomial form:
        (x^4+x^3+x^2)/(x^3+x^1+1) =x^1+1  r (1)
```

Applying $GF(2^m)$ to the Advanced Encryption Standard: Rijndael

In 2002, the National Institute of Standards and Technology (NIST) adopted the Advanced Encryption Standard (AES) also known as Rijndael. This is currently the standard encryption algorithm that is designed to be used by Federal departments and agencies have information that requires encryption [NIST]. The algorithm accepts a 128 bit sequence of plaintext information and cycles through four layers to produce the ciphertext which is also a 128 bit sequence of data.

The first step in the Rijndael algorithm is to group the 128 bit input into 16 bytes of 8 bits and arrange them into a 4 × 4 array of input bytes.

Input bytes

$$\begin{pmatrix} in_1 & in_5 & in_9 & in_{13} \\ in_2 & in_6 & in_{10} & in_{14} \\ in_3 & in_7 & in_{11} & in_{15} \\ in_4 & in_8 & in_{12} & in_{16} \end{pmatrix} \Rightarrow$$

State Array

$$\begin{pmatrix} s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \\ s_4 & s_8 & s_{12} & s_{16} \end{pmatrix} \Rightarrow$$

Output bytes

$$\begin{pmatrix} out_1 & out_5 & out_9 & out_{13} \\ out_2 & out_6 & out_{10} & out_{14} \\ out_3 & out_7 & out_{11} & out_{15} \\ out_4 & out_8 & out_{12} & out_{16} \end{pmatrix}$$

The input array is then manipulated by the 4-layer algorithm in 10, 12, or 14 rounds for key lengths of 128, 192, or 256 bits. We will now describe the specifics of each encryption layer as prescribed by [NIST].

ByteSub (BS)

The ByteSub routine is a non-linear byte substitution that operates on each byte with bits $\{b_0, b_1 \dots, b_7\}$ using the affine transformation

S-Box Values																	
S(rs)	**s**																
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
R	**0**	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84

	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \end{pmatrix}.$$

To simplify matters, we can compute this transformation on all possible bytes $\boldsymbol{b}$ in $GF(2^8)$ and place them in a look up table called an S-box. Below is a copy of the AES S-box in hexadecimal format.

In the ByteSub layer of the algorithm, we take a byte such as {10011011} and use the first four digits $\{1001\} = \{9\}$ which tell us to look in row 9, and the second four digits $\{1011\} = \{b\}$ which gives us column b. Thus,

$$\{10011011\} = \{9b\} \rightarrow \{81\} = \{10000001\}.$$

ShiftRow (SR)

The ShiftRow layer offsets the bytes cyclically by 0, 1, 2, and 3 columns in rows 1, 2, 3, and 4 respectively. This gives us

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} \Rightarrow \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{pmatrix}$$

MixColumns (MC)

In the MixColumn layer, we consider each column as a 4-term polynomial in $GF(2^8)$ such as

$$b(x) = b_3X^3 + b_2X^2 + b_1X + b_0$$

where b_0, b_1, b_2, and b_3 are bytes. We multiply this polynomial modulo $(X^4 + 1)$ by the fixed polynomial

$$a(X) = \{0011\}X^3 + \{0001\}X^2 + \{0001\}X + \{0010\} \pmod{X^4 + 1}.$$

Now, since $X^4 + 1$ is not irreducible, a polynomial does not necessarily have an inverse in this modulo. However, $a(X)$ is a primitive polynomial modulo $X^4 + 1$, so the inverse does exist. Namely,

$$a^{-1}(X) = \{1011\}X^3 + \{1101\}X^2 + \{1001\}X + \{1110\} \pmod{X^4 + 1}.$$

It is crucial that any cryptographic system be reversible to be able to decrypt the ciphertext.

Let's analyze this product $a(X)b(X)$ with

$$a(X) = \sum_{i=0}^{3} a_i X^i, \qquad b(X) = \sum_{j=0}^{3} b_j X^j.$$

We note that when working (mod $X^4 + 1$) we have

$$X^i \equiv X^{i \bmod 4} (\text{mod } X^4 + 1).$$

Multiplying and collecting like terms, we get

$$c(X) = a(X) \cdot b(X) = \sum_{n=i+j=0}^{6} c_n X^n = \sum_{n=i+j=0 \bmod 4}^{3 \bmod 4} d_n X^n,$$

where

$$d_n = \sum_{i+j=0 \bmod 4}^{3 \bmod 4} a_i \cdot b_j.$$

This gives us

$$\begin{aligned} d_0 &= a_0 \cdot b_0 \oplus a_3 \cdot b_1 \oplus a_2 \cdot b_2 \oplus a_1 \cdot b_3 \\ d_1 &= a_1 \cdot b_0 \oplus a_0 \cdot b_1 \oplus a_3 \cdot b_2 \oplus a_2 \cdot b_3 \\ d_2 &= a_2 \cdot b_0 \oplus a_1 \cdot b_1 \oplus a_0 \cdot b_2 \oplus a_3 \cdot b_3 \\ d_3 &= a_3 \cdot b_0 \oplus a_2 \cdot b_1 \oplus a_1 \cdot b_2 \oplus a_0 \cdot b_3 \end{aligned}$$

or in matrix form

$$\begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix}.$$

This matrix sufficiently defines the MixColumn transformation when applied to each column $(b_0, b_1, b_2, b_3)^T$ of the shift matrix.

AddRoundKey (ARK)

In the AddRoundKey layer, we XOR the shift matrix with the round key matrix as defined by the key schedule which is again defined by operations in $GF(2^8)$.

Key Schedule

The Rijndael system is designed to work with a key of 128, 192, or 256. We will describe the key schedule for the 128-bit algorithm. Larger keys are analogous resulting in more rounds which use the extra keys. This is a private-key that will needed to decrypt the ciphertext.

The key schedule matrix is designed in the following steps:

1) Arrange the 128-bit key into a 4×4 matrix of bytes as done previously.
2) Add 40 columns to the matrix as follows:
    a. Designate the first 4 columns $W(0), W(1), W(2), W(3)$.
    b. For successive columns i
    If $i > 0 \bmod 4$, then
    $$W(i) = W(i-4) \oplus W(i-1)$$
    If $i \equiv 0 \bmod 4$, then
    $$W(i) = W(i-4) \oplus T(W(i-1))$$
    Where $T(W)$ is a transformation in which we:
        i. Shift elements cyclically in column $W(i-1)$.
        i.e. A row $[a, b, c, d]^T \rightarrow [b, c, d, a]^T$
        ii. Replace bytes with the corresponding element in the S-box to get $[e, f, g, h]^T$.
        iii. Compute round constant
        $$r(i) = 00000010^{(i-4)/4}$$
        in $GF(2^8)$ and compute
        $$T(W) = (e + r(i), f, g, h).$$

Rijndael Encryption Summary

For the 128-bit key, we encryption includes

1) ARK with the 0th round key,
2) Nine rounds of BS, SR, MC, and ARK, using keys 1 to 9, and
3) Tenth round of BS, SR, and ARK using the 10th key.

Rijndael Decryption Steps

Now, the key feature of any cryptographic system is that it is reversible, which is true about each step of Rijndael. Hence, we can use inverse operations IARK, ISR, IBS, and IMC in reverse order to decrypt the ciphertext. However, we would like a decryption algorithm that looks similar to the encryption algorithm. In the end, we get the steps

1) ARK with 10th round key,
2) Nine rounds of IBS, ISR, IMC, IARK using keys 9 to 1
3) Tenth round of IBS, ISR, and ARK using the 10th key.

Inverse ByteSub

In this step, we apply the inverse affine transformation to each byte in the shift array and find the multiplicative inverse of the result in $GF(2^8)$. The affine transformation here is again accomplished by another look-up table.

Inverse ShiftRow

In this step, we shift rows to the right instead of the left by 0, 1, 2, and 3 entries respectively. Resulting in the formula

$$s'_{r,(c+shift(r,4)) \bmod 4} = s_{r,c}.$$

Inverse MixColumn

Again, we treat each column as a 4th –degree polynomial modulo $X^4 + 1$ in $GF(2^8)$. The result is the matrix product

$$\begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

where the a_i entries are the coefficients of the equation

$$a^{-1}(X) = \{1011\}X^3 + \{1101\}X^2 + \{1001\}X + \{1110\} \pmod{X^4 + 1}$$

which we apply column-by-column.

Inverse AddRoundKey

Since ARK is simply an XOR of the round key, it is its own inverse.

Now, using these inverse operations, we can perform the layers in exactly reverse order for the middle rounds: ARK, IMC, ISR, IBS. However, we may like a decryption algorithm that is similar to the encryption algorithm. Since ISR and IBS work bit-by-bit, we can reverse these.

ARK and IMC cannot be reversed so easily. We can accomplish this if we define the InverseAddRoundKey (IARK) as the process of XORing

$$[k'_{i,j}] = [m_{i,j}]^{-1}[k_{i,j}],$$

where $[m_{i,j}]^{-1}$ is the inverse of the MixColumn matrix. We can now substitute "ARK, then IMC" for "IMC, then IARK". This leaves us with the core of the decryption computed in the steps IBS, ISR, IMC, IARK. Regrouping gives us the decryption steps mentioned above.

In conclusion, we see that the properties of the finite field $GF(2^8)$ are crucial to the operation and integrity of this cryptographic system. The multiple layers serve to defend against any known methods of linear or differential cryptanalysis. Furthermore, with the ability to use larger keys, the Rijndael system is likely to be effective for many years to come.

Error correction code

Another important application of Galois fields is in error correction codes. When transmitting a cryptographic ciphertext, the corruption of even one bit can make a plaintext message unreadable. Furthermore, any machine that transmits digital data is susceptible to errors from bit reversal due to noise. The goal of error correction codes is to identify the bit or bits that have been altered so they can be returned to their original state.

One technique for error correction is the use of linear codes. A linear code has dimension k and length n over field F if it forms a k-dimensional subspace of F^n. If $F = Z_2$, then we have a binary code of length n and dimension k which is a set of 2^k binary n-tuples which are the codewords. [TW,409]

One class of error correction codes are the Hamming codes. We define an $[n, k]$ block code as one that encodes a k-bit information word to an n-bit codeword. This is done by multiplying the k-bit binary message by a generating matrix. Following transmission, a check matrix identifies any errors present in the n-bit codewords. We use the properties of Galois fields $GF(2^m)$ and primitive polynomials of degree m to create a generator polynomial $G(X)$ the block code. [RT1]

A primitive polynomial is a monic irreducible polynomial whose roots are primitive elements, which are generators for the group $GF^*(p^n)$ of nonzero elements of $GF(p^n)$ for prime p. Such polynomials can be derived using an algorithm such as Hansen and Mullen [HM] to produce tables of primitive polynomials. For small values of n, we can find the powers of its roots to see if it is primitive. We have seen above that $P(X) = X^3 + X^2 + 1$ is an irreducible polynomial for $GF(2^3)$, so we will now check to see if it is a primitive polynomial. To do this, we assume that a is a root of $P(X) = X^3 + X^2 + 1$. Thus,

$$P(a) = a^3 + a^2 + 1 = 0$$

and

$$a^3 = a^2 + 1.$$

We now observe that the powers of a give us all the nonzero elements of $GF(2^4)$.

$a^0 = a^0$	$= 1$		$= 001 = 1$
$a^1 = a^0 \times a$	$= 1 \times a$	$= a$	$= 010 = 2$
$a^2 = a^1 \times a$	$= a \times a$	$= a^2$	$= 100 = 4$
$a^3 = a^2 + 1$	$= a^2 + 1$	$=$	$= 101 = 5$
$a^4 = a^3 \times a$	$= (a^2 + 1) \times a$	$= a^3 + a = a^2 + a + 1$	$= 111 = 7$
$a^5 = a^4 \times a$	$= (a^2 + a + 1) \times a$	$= a^3 + a^2 + a = a^2 + 1 + a^2 + a = a + 1$	$= 011 = 3$
$a^6 = a^5 \times a$	$= (a + 1) \times a$	$= a^2 + a$	$= 110 = 6$
$a^7 = a^6 \times a$	$= (a^2 + a) \times a$	$= a^3 + a^2 = a^2 + 1 + a^2 = 1$	$= 001 = 1$

We see from the power $a^7 = 1$ that we have the cyclic group $GF^*(2^3)$ which are all the possible remainders when working (mod $X^3 + X^2 + 1$).

To design a linear $[n, k]$ hamming code $C \subset GF(2^m)$, we first consider the parity check matrix H which has the property that a vector $v \in GF(2^m)$ is in if and only if $vH^T = 0$. The following theorem from [TW] to aid us in connecting the generator and parity check matrix.

> **Theorem EC-1:** If $G = [I_k, P]$ is the generating matrix for a code C, then $H = [-P^T, I_{n-k}]$ is a parity check matrix for C.

So, once we have the parity check matrix, the generating matrix follows. One method of constructing the parity check matrix described by Tervo [RT1] is to compile the remainders found above into a matrix with columns $[a^6 \quad a^5 \quad a^4 \quad a^3 \quad a^2 \quad a^1 \quad a^0]$. Using the results from above, we get the check matrix

$$H = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}^T.$$

Using Theorem EC-1, we compute the generating matrix

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

Example Suppose we begin with a plaintext word, $p = 1010$. We will encode it with G, alter one bit, then use the parity check matrix H to identify the error.

Step 1. Compute code word $c=pG$

$$c = pG = (1 \quad 0 \quad 1 \quad 0)\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} = (1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1).$$

Observe that the codeword contains the plaintext word in the first four entries, followed by three check bits 001.

Now, we will alter the 4th bit to get $c' = (1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1)$.

Step 2. Compute the check bit product $c'H$.

$$c'H = (1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1)\begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}^T = (1 \quad 0 \quad 1).$$

Now, if the codeword was unaltered, we would have had a zero product when multiplied by the check matrix. However, we received $c'H = (1 \quad 0 \quad 1)$, which is the fourth column. This tells us that the fourth bit has been altered, an easy problem to fix working modulo 2. This gives us

$$cH = (1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1)\begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}^T = (0 \quad 0 \quad 0)$$

Why does it work? Valid 7-bit codewords must have a zero remainder when working in $GF(2^3)$ modulo $P(X) = X^3 + X^2 + 1$. Since the codewords begin with the 4-bit input, there are $2^4 = 16$ possible multiples of the primitive polynomial 1101.

$$\begin{aligned} 0 \times 1011 &= 0000000 \\ 1 \times 1011 &= 0001011 \\ 10 \times 1011 &= 0010110 \\ &\vdots \\ 1111 \times 1011 &= 1101001 \end{aligned}$$

The rows of hamming matrix H is an orthogonal basis for these possible codewords. This results in a zero product when a correct codeword is mulplied by the hamming matrix. When one bit is changed, this alters the XOR process in the exact column of the matrix product which reveals the error.

Finite Elements and Partial Differential Equations

The applications of Partial Differential Equations (PDEs) are endless in the all forms of science and engineering. PDEs give us the ability to accurately describe complex systems in space and/or time from just a few initial conditions and boundary conditions. Many techniques are available for solving linear PDEs in standard forms such as the wave equation, the Laplace equation, and parabolic equations. However, in many applications, the equations become too complex for known analytical methods.

In this situation, we must use numerical methods of approximating solutions to PDEs. One common technique for solving PDEs numerically is with a difference method. These methods (of which there are several variations) approximate derivatives by calculating differences over increasingly small intervals.

Another technique, and the one that we will focus on here, is the Finite Element Method. In this method, a given region is divided into a finite number of geometric sub-regions, called the finite elements. Using initial values and boundary values, we then use a set of basis functions from a chosen function space to extrapolate the values of the solution for each finite element.

The key steps in the Finite Element Method are

1. Define our finite element space V_h and the nature and parameters of the functions v in V_h.
2. Compute the local stiffness matrix and the coefficients of the local basis functions.
3. Compute the values of the global nodes and maps the local nodes associated to an element to global nodes.
4. Compute the global stiffness matrix, S, which is the coefficients of the system which we need to solve.
5. Compute the values of the vector $\mathbf{b} = (b_i)$, where $b_i = \int_\Omega f(x,y)\phi_i(x,y)dx$(We will approximate this integral by quadrature).
6. Finally, we will solve the matrix equation $Sx = \mathbf{b}$.

A careful description of this process and an example of a solution to the Poisson problem:

$$-\Delta u = f \quad in \quad \Omega, \qquad \Omega = [0,1]^2$$
$$u = 0 \quad on \quad \partial\Omega$$

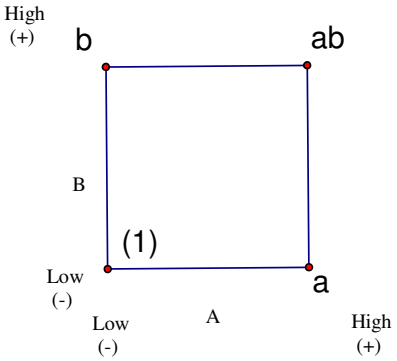
using the finite element method is available at [JK]. The method is implemented using MATLAB algorithms. The algorithm is then tested using a function f that gives us a partial differential equation with a known analytic solution, such as $f = \sin(\pi x)\sin(\pi y)$.

Finite Sets and Statistics

Experiments are an important tool for all areas of science and engineering. To perform effective experiments and interpret the results accurately, it is important to carefully consider the *design* of the experiment. Often a result is affected by multiple factors. To correctly consider all of these factors, it is useful to perform a *factorial experiment*, which is performed at all factor levels.

If we have k factors that can be controlled, we can use a 2^k factorial design. When we conduct experiments with multiple factors, we must study the effects of the individual factors as well as the joint effect of the factors on the response. Suppose we have k factors that have an effect on the response. These can be quantitative or qualitative responses which are studied at only two levels for each factor. This is called a 2^k factorial design since the experiment requires $2 \times 2 \times \ldots \times 2 = 2^k$ observations. [MR]

In a 2^2 factorial design, we have two factors, say A and B. We will observe these factors at two levels, low(-) and high(+), and denote them with these symbols. This design requires $2^2 = 4$ observations as shown in the geometric model below [MR]. Here we let the letters (1), a, b, and ab represent the total of all n observations taken at these levels. By using this design, we can be sure to address all possible factors and interactions.



Treatment	A	B
(1)	-	-
a	+	-
b	-	+
ab	+	+

When we are conducting an experiment with the 2^k factorial design, we see that the combinations of the (+) and (-) symbols mirror the binary representation of the polynomials in $GF(2^k)$. The example below applies a 2^4 factorial design to determine which factors are significant in the yield of a chemical process. While one might be tempted to test each of the factors at many levels, the factorial model gives us a good picture of which factors are significant by testing them at only two levels each.

Example:[MR]

An article in *Analytica Chimica Acta* examined four parameters that affect the sensitivity and detection of the analytical instruments used to measure clinical samples. They optimized the sensor function using EBC samples spiked with acetone, a known clinical biomarker in breath. The following table shows the results for a single replicate of a 2^4 factorial experiment for one of the outputs, the average amplitude of acetone peak over three repetitions.

Configuration	A	B	C	D	Yield
1	+	+	+	+	0.12
2	+	+	+	-	0.1193
3	+	+	-	+	0.1196
4	+	+	-	-	0.1192
5	+	-	+	+	0.1186
6	+	-	+	-	0.1188
7	+	-	-	+	0.1191
8	+	-	-	-	0.1186
9	-	+	+	+	0.121
10	-	+	+	-	0.1195
11	-	+	-	+	0.1196
12	-	+	-	-	0.1191
13	-	-	+	+	0.1192
14	-	-	+	-	0.1194
15	-	-	-	+	0.1188
16	-	-	-	-	0.1188

A: RF voltage of the DMS Sensor (1200 or 1400V)

B: Nitrogen carrier gas flow rate (250 or $500 mL min^{-1}$)

C: Solid phase microextraction filter type (polyacrylate or PDMS-DVB)

D: GC cooling profile (cryogenic and noncryogenic)

Data: A 2^4 factorial experiment on clinical samples
Objective: Factor analysis and interaction of factor using an effects model
Hypotheses: We consider the null hypotheses below with a confidence level of 95%.

Main Effects – $H_0: (\alpha)_i = 0$
2-way Interaction Effects - $H_0: (\alpha\beta)_{ij} = 0$
3-way Interaction Effects – $H_0: (\alpha\beta\gamma)_{ijk} = 0$
(We will ignore 4-way interactions since they are highly unlikely to be significant.)

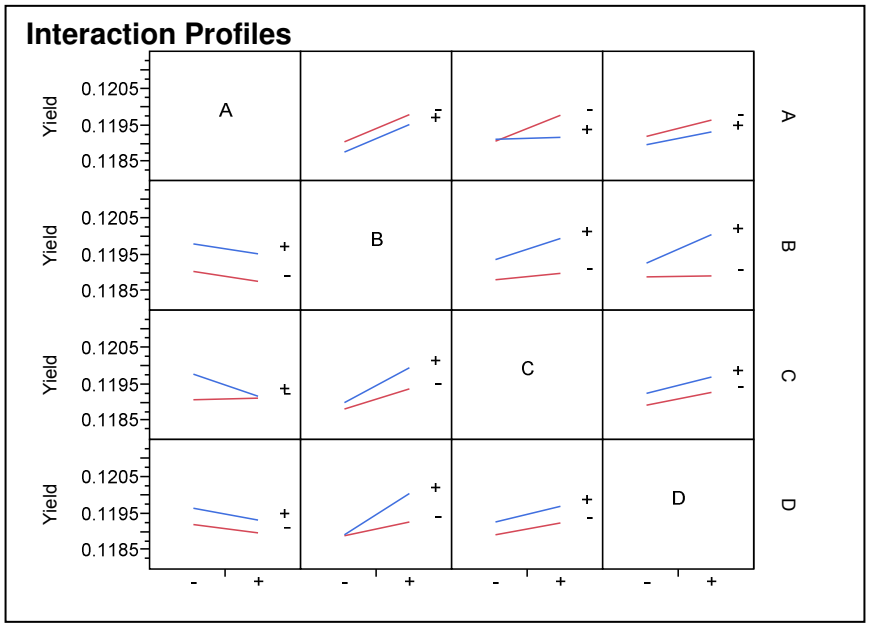
Computing the effects model, it is apparent that the significant main effects are B, C, and D since they give us a *p*-value of less than .05. Likewise, the significant two-way interactions are AC and BD. These interactions can also be seen below in the interaction profile plots since the plots of A,C and B,D intersect.

Effect Tests

Source	Nparm	DF	Sum of Squares	F Ratio	Prob > F
A	1	1	3.025e-7	121.0000	0.0577
B	1	1	0.00000225	900.0000	0.0212*
C	1	1	5.625e-7	225.0000	0.0424*
D	1	1	0.00000064	256.0000	0.0397*
A*B	1	1	4.8148e-35	0.0000	1.0000
A*C	1	1	4.225e-7	169.0000	0.0489*
A*D	1	1	0.00000001	4.0000	0.2952
B*C	1	1	0.00000016	64.0000	0.0792
B*D	1	1	5.625e-7	225.0000	0.0424*
C*D	1	1	0.00000001	4.0000	0.2952
A*B*C	1	1	0	0.0000	1.0000
A*B*D	1	1	1.225e-7	49.0000	0.0903
A*C*D	1	1	0.00000009	36.0000	0.1051
B*C*D	1	1	3.025e-7	121.0000	0.0577

To analyze the fit of the effects model, we look at the ANOVA table and see that the *p*-value for the model fit is .0628 which is too high.

Since A is not a main effect, we will remove this factor and recalculate the model. When we do this, we get a *p*-value of .0150. This is an acceptable value for our goodness of fit.



Analysis of Variance (Factors A, B, C, and D)

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	14	5.435e-6	3.8821e-7	155.2857
Error	1	2.5e-9	2.5e-9	**Prob > F**
C. Total	15	5.4375e-6		0.0628

Summary of Fit

RSquare	0.99954
RSquare Adj	0.993103
Root Mean Square Error	0.00005
Mean of Response	0.119288
Observations (or Sum Wgts)	16

Analysis of Variance (Factors B, C, and D)

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	7	4.4875e-6	6.4107e-7	5.3985
Error	8	0.00000095	1.1875e-7	**Prob > F**
C. Total	15	5.4375e-6		0.0150*

Summary of Fit

RSquare	0.825287
RSquare Adj	0.672414
Root Mean Square Error	0.000345
Mean of Response	0.119288
Observations (or Sum Wgts)	16

MATLAB Code

The following is MATLAB code referred to in this document.

```
function c=binxor(a,b)
%Computes the binary XOR of two binary numbers a and b
%inputs are binary numbers in string format
c=dec2bin(bitxor(bin2dec(a),bin2dec(b)));
```

```
function s=binmult(a,b)
%This function accepts two binary integers a and b as strings
%and returns the product as a binary number in character format
%This is performed by first doing a bitshift for each 1 in the smallest
%factor and then using XOR to sum the results.

%set x to be the shortest binary string and y to be the longest
%we will use a distribution technique to multiply y*x.
if length(a)<length(b)
   x=a;
   y=b;
else
   x=b;
   y=a;
end
n=length(x); %n=length of shortest binary string x
% set initial value for product string
if x(n)=='1'
  s=y;
else
   s='0';
end
for i=(n-1):-1:1
   if x(i)=='1'
      t=dec2bin(bitshift(bin2dec(y),n-i));
      s=dec2bin(bitxor(bin2dec(s),bin2dec(t)));
   end
end
```

```
function [quotient,dd]=bindiv(dividend,divisor,display)
%Divides two binary numbers and returns two binary number strings for
%q=quotient and r=remainder
%display=1 will print intermitent values
if bin2dec(divisor)==0
   disp('Error - Divisor equal to zero')
   quotient='N/A';
   dd='N/A';
   return
end

%remove leading zeros and store dividend and divisor
dd=dec2bin(bin2dec(dividend));
dr=dec2bin(bin2dec(divisor));

ddLength=length(dividend);
drLength=length(divisor);

%Check to see if dividend polynomial has greater degree than or equal to divisor polynomial
if ddLength<drLength
```

```
   quotient=dec2bin(0);
   return
end
disp(strcat(dd,'/',dr,'='));
quotient='0';
% i=0;
while ddLength>=drLength
%    i=i+1;
   if dd(1)=='1'  %each leading 1 in dividend determines a factor of divisor can be divided
      bitplace=(ddLength-drLength);
      qpart=dec2bin(bitshift(bin2dec(dr),bitplace));
      dd=dec2bin(bitxor(bin2dec(qpart),bin2dec(dd)));
      if display==1
         disp(strcat(quotient,'+(',qpart,'+',dd,')/',dr));
      end
      quotient=dec2bin(bitxor(bitshift(1,bitplace),bin2dec(quotient)));
      if display==1
         disp(strcat(quotient,'+',dd,'/',dr));
      end
      ddLength=length(dd);
   end
end
if display==1
   disp('Polynomial form:');
   disp(strcat('(',bin2poly(dividend),')/(',bin2poly(divisor),') = ',bin2poly(quotient),'  r (',bin2poly(dd),')'));
end
```

Sources

[TW] Trappe, Wade, and Washington, Lawrence C. *Introduction to Cryptography with Code Theory. 2nd Edition.* Prentice Hall. 2006.

[MR] Montgomery, Douglas C. and Runger, George C. *Applied Statistics and Probability for Engineers. 5th edition.* John Wiley and Sons. 2011.

[RT1] Tervo, Richard. *Error Control Codes from Galois Fields.* Course notes for EE4253 Digital Communications. University of New Brunswick. http://www.ee.unb.ca/cgi-bin/tervo/galois.pl

[RT2] Tervo, Richard. *The Hamming Code Revisited – A Matrix Approach.* Course notes for EE4253 Digital Communications. University of New Brunswick. http://www.ee.unb.ca/tervo/ee4253/hamming2.shtml

[NW] Wagner, Niel R., The Laws of Cryptography; The Finite Field GF(2^8). http://www.cs.utsa.edu/~wagner/laws/FFM.html http://www.cs.utsa.edu/~wagner/lawsbookcolor/laws.pdf

[NIST] *Specification for the Advanced Encryption Standard (AES).* National Institute of Standards and Technology. November 26, 2001. *http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf*

[HM] Hansen, Tom and Mullen, G.L. *Primitive Polynomials Over Finite Fields.* Mathematics of Computation. American Mathematical Society. 1992. http://www.ams.org/journals/mcom/1992-59-200/S0025-5718-1992-1134730-7/S0025-5718-1992-1134730-7.pdf

[MK] Kyuregyan, Melsik K. *Iterated constructions of irreducible polynomials over finite fields with linearly independent roots.* Science Direct. 2003. http://web.mit.edu/minilek/Public/irreducible.pdf

[CJ] Johnson, Claes. Numerical Solution of Partial Differential Equations by the Finite Element Method. Dover Publications. 2009.

[JK] Knight, Jeremy. *Solving Poisson's Equation with the Finite Element Method.* 2011. http://knightmath.com/tamu/poisson/pages/poisson&FEM.pdf