Galois Fields and Forth

Bill Stoddart and John Goldman

September 25, 2019

Abstract

Galois fields are rich finite algebraic structures with applications in cryptography, error correcting codes, experimental design, constraint programming and pattern recognition. We describe some of these fields and the structures related to them known as Latin Squares, which are used in many applications. We describe and implement examples of the polynomial arithmetic that underlies Galois Fields, we describe the automorphisms between different implementations of the "same" field, and we give an implementation of the field used in the Advanced Encryption Standard. As an example application we consider in detail the construction of a pack for the children's card game Dobble, in which there are cards marked with symbols such that any two cards share exactly one symbol. We include mathematical appendices in which we prove, or show how to prove by comprehensive validation, various important properties satisfied by these fields.

1 Introduction

Galois fields are named in honour of the French mathematician Evariste Galois, an inspiring but tragic figure who died at 20 in a duel yet left a huge legacy. He discovered them whilst investigating the properties of polynomials. These fields are rich finite algebraic structures with applications in cryptography, error correcting codes, experimental design, constraint programming and pattern recognition. Each Galois field consist of values $0..p$ along with two operations which are analogous to multiplication and addition. These operations obey the axioms of a mathematical "field", which are also shared by real numbers, but not, totally, by integers, since every element in a Galois

field has an exact multiplicative inverse. Galois fields also support operations analogous to subtraction and division, and division is perfect, there is never a remainder.

This paper is organised as follows. In section 2 we give the axiomatic properties of a Galois field. In section 3 we show how Galois fields with a prime number of elements are implemented. We define Latin squares, and also what it means for two Latin squares to be orthogonal. We show how orthogonal Latin squares can be used to solve an old playing card puzzle. In section 4 we look at fields of size q^n where q is prime, and in particular we implement the field of size 8. We show there are different implementations of this field which are related by an automorphism. In section 5 we implement the Galois field with 256 elements, which is mandated for use in the Advanced Encryption Standard.

In section 6 we look at our main example problem, which is the allocation of symbols to cards in Dobble packs. In section 7 we consider a Forth implementation for the construction of a Dobble pack. In section 8 we conclude.

In the mathematical appendices we prove that modular arithmetic does not generally yield a Galois Field, and we prove the existence of the mutually orthogonal Latin squares associated with each field. Finally we consider the proof of field axioms by exhaustive verification.

2 Galois Field Properties

A Galois field $\{\mathcal{F}\}$ of size p consists of two operations analogous to addition and multiplication, acting on a set of values 0..p and obeying certain axioms. A Galois field will exist whenever p is the power of a prime., e.g. 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, .. 256 ...

The addition and multiplication on a Galois field of size $p=q^n$ where q is prime are those of "polynomial modular arithmetic", which we will explain in due course. In the case where p is prime this reduces to addition and multiplication modular p and we can define the Galois field operations as:

```
: +0 + p MOD  ;          : *0  *  p  MOD  ;
```

We will write the operations of a Galois field as $+_0$ and $*_0$ to distinguish them from normal addition and multiplication. If considering more than one implementation of the same field we will also use the names $+_1$, $*_1$, $+_\_$, and $*_\_$. The field size will be clear from context.

The axioms of a Galois field are a set of rules that, with one exception, also apply to integers. The exception is that every element has an inverse *within the field*. The integer $\mathtt{n}$ can be said to have inverse $\mathtt{1/n}$, but this inverse is not itself an integer. The inverses of Galois field elements are in the field.

2.1 Axioms

In the following $\mathtt{u}, \mathtt{v}, \mathtt{w}$ are arbitrary values in the field $\mathcal{F}$.

Closure. If $\mathtt{u}$, $\mathtt{v}$ are in the field so are $\mathtt{u} +_0 \mathtt{v}$ and $\mathtt{u} *_0 \mathtt{v}$

Commutativity. $\mathtt{u} +_0 \mathtt{v} = \mathtt{v} +_0 \mathtt{u}$ and $\mathtt{u} *_0 \mathtt{v} = \mathtt{v} *_0 \mathtt{u}$

Associativity. $(\mathtt{u} +_0 \mathtt{v}) +_0 \mathtt{w} = \mathtt{u} +_0 (\mathtt{v} +_0 \mathtt{w})$

Distributivity $\mathtt{u} *_0 (\mathtt{v} +_0 \mathtt{w}) = (\mathtt{u} *_0 \mathtt{v}) + (\mathtt{u} *_0 \mathtt{w})$

Unit property $\mathtt{u} *_0 \mathtt{1} = \mathtt{u}$

Zero properties $\mathtt{u} +_0 \mathtt{0} = \mathtt{u}$ and $\mathtt{u} *_0 \mathtt{0} = \mathtt{0}$

Inverses. If $\mathtt{u}$ is non-zero it has a multiplicative inverse $\mathtt{u}^{-1}$ in $\mathcal{F}$ such that $\mathtt{u} *_0 \mathtt{u}^{-1} = \mathtt{1}$, and has an additive inverse $\sim\mathtt{u}$ in $\mathcal{F}$ such that $\mathtt{u} +_0 \sim\mathtt{u} = \mathtt{0}$

The existence of inverses means division and subtraction may be defined, with $\mathtt{a} /_0 \mathtt{b} \mathrel{\widehat{=}} \mathtt{a} *_0 \mathtt{b}^{-1}$ and $\mathtt{a} -_0 \mathtt{b} \mathrel{\widehat{=}} \mathtt{a} +_0 \sim\mathtt{b}$

3 Latin Squares, orthogonality, and a combinatorial playing card problem

A Latin square is an $\mathtt{n} \times \mathtt{n}$ array containing $\mathtt{n}$ different elements with each of these occurring once in each column and once in each row.

Two Latin squares are orthogonal if the pairs of elements formed from the first and second square are all different.

Here are two orthogonal Latin squares, together with the square of pairs formed from them:

```
0 1 2 3   0 1 2 3   0,0 1,1 2,2 3,3
1 0 3 2   2 3 0 1   1,2 0,3 3,0 2,1
2 3 0 1   3 2 1 0   2,3 3,2 0,1 1,0
3 2 1 0   1 0 3 2   3,1 2,0 1,3 0,2
```

A Galois field of size `n` is associated with `n-1` mutually orthogonal Latin squares, with the element at the j,kth position the ith such square given by $\texttt{i} *_0 \texttt{j} +_0 \texttt{k}$. A mathematical proof is given in the appendix.

As an example application of orthogonal Latin squares, and a justification of the term "orthogonal", consider finding a solution to the problem of arranging the 16 picture cards from a pack in a 4×4 square such that in each row and each column we have one Ace, one King, one Queen, one Jack, one Heart, one Diamond, one Club, and one Spade. We can use two orthogonal Latin squares to split the program into two. We use the first Latin square to allocate positions for aces, kings queens and jacks, and the second to allocate positions to hearts diamonds clubs and spades. Under this interpretation the Latin squares above give the shown solution to our picture card problem.

4 Polynomial arithmetic and fields of size 2^n

To implement a Galois field of size $\texttt{q}^\texttt{n}$ where `q` is prime we use addition and multiplication of polynomials with coefficients taken from the field G(q). To explain this fully we consider fields of size $2^\texttt{n}$.

The Galois field G(2) has elements 0,1 with the multiplication:

$$0 *_0 0 = 0,\ 0 *_0 1 = 0,\ 1 *_0 0 = 0,\ 1 *_0 1 = 1$$

so multiplication in this field is equivalent to logical AND. And with the addition we have:

$$0 +_0 0 = 0,\ 0 +_0 1 = 1,\ 1 +_0 0 = 1,\ 1 +_0 1 = 0$$

so addition in this field is equivalent to logical XOR. Also, subtraction modulo 2 is identical to addition modulo 2 so addition and subtraction in the field are identical.

We now consider the polynomial arithmetic required to implement the field G(8), whose elements are the polynomials of order 2 and below. Given that the coefficients of these polynomials are from G(2), i.e. take only the values 0 and 1, we have 8 such polynomials, which are:

$$0\ ,\ 1\ ,\ x\ ,\ x +_0 1\ ,\ x^2\ ,\ x^2 +_0 1\ ,\ x^2 +_0 x\ ,\ x^2 +_0 x +_0 1$$

and these have coefficients that can be encoded in 3 bits as:

000 , 001 , 010 , 011 , 100 , 101 , 110, 111

these polynomials, or more exactly the encoding of their coefficients, will be the members of our implementation of the field G(8). [1]

An example of polynomial addition in this system is:

$$(x^2 +_0 1) +_0 (x +_0 1) = x^2 +_0 x +_0 1 +_0 1 = x^2 +_0 x$$

The bit encoded form of the same calculation, with $\oplus$ representing exclusive or, is $101 \oplus 011 = 110$.

The Forth definition for addition in field G(8), or any field $G(2^n)$, will be

```
: +0  ( n1 n2 -- n3 , n3 = n1 +0 n2 )  XOR ;
```

An example of polynomial multiplication is

$$(x^2 +_0 x +_0 1) *_0 (x^2 +_0 x) = x^4 +_0 x^3 +_0 x^2 +_0 x^3 +_0 x^2 +_0 x$$
$$= x^4 +_0 x$$

We note that the result $x^4 +_0 x$ isn't one of the 8 polynomials of G(8), so we can't use polynomial multiplication as the multiplication operator $*_0$ of our field. We have to reduce the result my taking its modulus relative to a reduction polynomial. This is analogous to modular arithmetic where our multiplication operator for modulus p arithmetic is defined as:

```
: *0 ( n1 n2 -- n3, n3 = n1 *0 n2) * p MOD ;
```

[1] Mathematical note: we treat polynomials as mathematical objects, rather than as expressions denoting mathematical objects. Considered as expressions, polynomials x and x^2 are identical, since with mod 2 arithmetic $x = x^2$. Under the interpretation in which polynomials are mathematical objects they correspond formally to sequences of coefficients $\langle 0, 1, 0 \rangle$ and $\langle 1, 0, 0 \rangle$ and polynomial arithmetic operations are defined on such sequences of coefficients. Representing such sequences of coefficients as bit sequences is an implementation technique.

Just as, for modular arithmetic to yield a field p must be prime, that is a number without factors, our reduction polynomial must also not have factors. It must also be higher order than the polynomials on the field. A suitable polynomial is $x^3 +_0 x +_0 1$.

The polynomial arithmetic we are performing is analogous to our numeric arithmetic. Indeed, we write our numbers in an abbreviated polynomial form; for example 406 is the value of $4x^2 + 0x + 6$ when $x = 10$. We might calculate the quotient and modulus when dividing 406 by 123 as follows

```
      3
123)406 _
    369          The calculation shows us that
     37          406 = 123 * 3 + 37
```

Our division of $x^4 +_0 1$ by $x^3 +_0 x +_0 1$ can be shown as follows (we explicitly include the zero terms in x, x^2 and x^3)

$$
\begin{array}{r}
\multicolumn{1}{r}{x} \\
\hline
x^3 +_0 x +_0 1 \;)\; x^4 +_0 0 +_0 0 +_0 0 +_0 1 \\
\multicolumn{1}{r}{x^4 +_0 0 +_0 x^2 +_0 x} \\
\hline
\multicolumn{1}{r}{x^2 +_0 x +_0 1}
\end{array}
$$

and shows us that $x^4 +_0 1 = (x^3 +_0 x +_0 1) *_0 x +_0 x^2 +_0 x +_0 1$

The term we are interested in here is the remainder $x^2 +_0 x +_0 1$. this is the result of our Galois field multiplication. In full:

$$
\begin{array}{l}
(x^2 +_0 x +_0 1) *_0 (x^2 +_0 x) = \\
(\,(x^2 +_0 x +_0 1) * (x^2 +_0 x)\,) \bmod (x^3 +_0 x +_0 1) = \\
x^2 +_0 x +_0 1
\end{array}
$$

Multiplication in the field G(8) can be defined in Forth as:

```
: *0 ( a b -- a *0 b, multiplication of elements from G(8) )
   (: a b :) 0 ( the polynomial product a*b will be collected on the
   stack, we note XOR being used for addition since we are using
   modulo 2 arithmetic on our polynomial coefficients )
   1 b AND IF a XOR THEN
   a 2* to a
   2 b AND IF a XOR THEN
   a 2* to a
   4 b AND IF a XOR THEN
   ( modulo 2 polynomial product now on stack, we now divide
     it by our reduction polynomial x^3+x+1 )
```

```
DUP 16 AND IF 22 XOR THEN
DUP   8 AND IF 11 XOR THEN ( 11 ~ 1011₂ ~ x³+x+1 ) ;
```

The reduction polynomial is not unique (except in the case of G(4)). For G(8) an alternative reduction polynomial is x^3+x^2+1, and we will use $+_1$ and $*_1$ as the names for the corresponding field operations. We have the following addition and multiplication tables.

$+_0$	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

$*_0$	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

$+_1$	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

$*_1$	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	5	7	1	3
3	0	3	6	5	1	2	7	4
4	0	4	5	1	7	3	2	6
5	0	5	7	2	3	6	4	1
6	0	6	1	7	2	4	3	5
7	0	7	3	4	6	1	5	2

Although it is accepted usage to refer, in the singular, of "the field G(8)" we clearly have two different Galois field multiplication operators. However, they may be seen as acting on different encodings of the abstract elements of the field G(8) rather than being operations from two different fields. The following permutation relates the encodings used with reduction polynomial $x^3 +_0 x +_0 1$ to those used with reduction polynomial $x^3 +_0 x^2 +_0 1$

$\mathtt{perm} = \{ 0 \mapsto 0,\ 1 \mapsto 1,\ 2 \mapsto 3,\ 3 \mapsto 2,\ 4 \mapsto 5,\ 5 \mapsto 4,\ 6 \mapsto 6,\ 7 \mapsto 7 \}$

Now for any $a, b \in 0..7$ we have the following "automorphism" relating $+_0$ to $+_1$

$\mathtt{perm}(a +_0 b) = \mathtt{perm}(a) +_1 \mathtt{perm}(b)$

and a similar automorphism relating $*_0$ to $*_1$

$\texttt{perm(a}*_0\texttt{ b) = perm(a)}*_1\texttt{ perm(b)}$

In Forth we can implement this permutation function as:

```
HERE 0 C, 1 C, 3 C, 2 C, 5 C, 4 C, 6 C, 7 C,
: PERM ( 8b -- 8b ) LITERAL + C@ ;
```

and the checks we must do to establish the automorphisms are that for arbitrary I and J from the field

```
I J +0 PERM   I PERM J PERM +1  =
I J *0 PERM   I PERM J PERM *1  =   AND
```

5 Galois fields and encryption

All unary Galois field operations (add x, subtract x, multiply by x, divide by x, additive inverse, multiplicative inverse) have inverses. Thus if such operations are used to encode data, we can always rely on there being inverse operations that can decode it. Also, there are unary operations to take any point in the field to any other point in the field. These properties make Galois field operations useful for scrambling bits during encryption.

The field G(256), with reduction polynomial $x^8 + x^4 + x^3 + x + 1$ is mandated for use in the Advanced Encryption Standard, which (Wikipedia tells us) is "currently the only publicly accessible cipher approved by the National Security Agency (NSA) for top secret information when used in an NSA approved cryptographic module".

An implementation of G(256) operations is given below. They follow the same form of polynomial addition and multiplication as we saw for G(8).

```
( Our reduction polynomial x^8+x^4+x^3+x+1 encodes to 100011011_2 )
2 BASE ! 100011011 CONSTANT poly HEX

: +_ XOR ;

: *_ (: a b :) 0
   1 b AND IF a XOR THEN
```

```
   a 2* to a
   2 b AND IF a XOR THEN
   a 2* to a
   4 b AND IF a XOR THEN
   a 2* to a
   8 b AND IF a XOR THEN
   a 2* to a
   10 b AND IF a XOR THEN
   a 2* to a
   20 b AND IF a XOR THEN
   a 2* to a
   40 b AND IF a XOR THEN
   a 2* to a
   80 b AND IF a XOR THEN
  ( modulus 2 polynomial product now on stack, its
      max possible value is < 8000 hex )
   DUP 4000 AND IF [ poly 40 * ] LITERAL XOR THEN
   DUP 2000 AND IF [ poly 20 * ] LITERAL XOR THEN
   DUP 1000 AND IF [ poly 10 * ] LITERAL XOR THEN
   DUP 800 AND IF [ poly 8 * ] LITERAL XOR THEN
   DUP 400 AND IF [ poly 4 * ] LITERAL XOR THEN
   DUP 200 AND IF [ poly 2 * ] LITERAL XOR THEN
   DUP 100 AND IF poly  XOR THEN ;

DECIMAL
```

6 Dobble

Dobble is a card game in which 8 symbols appear on each card, and any two cards have exactly one symbol in common.

We can generalise this to `n` symbols per card, so long as `n-1` is the power of a prime. We illustrate the construction of a Dobble pack using 5 symbols per card, represented by digits 0..9 and letters A , B .. K

We begin by writing down the cards that contain the symbol 0. To see how many such cards there, it will help if we write down at the same time the cards that contain 1. As we write down each 0 card we distribute its symbols between the 1 cards. After we have added the card `05678` we have the following situation, which shows us we need 4 cards that contain 1 (but not

0) in order to distribute the symbols 5, 6 7 and 8 between them, and thus avoid having a card with more than 1 symbol in common with `05678`

```
0 1 2 3 4
0 5 6 7 8   1 5 ? ? ?
            1 6 ? ? ?
            1 7 ? ? ?
            1 8 ? ? ?
```

continuing in this way and adding blocks for cards containing 2, 3 and 4 we arrive at.

```
0 1 2 3 4
0 5 6 7 8   1 5 9 D H   2 ? ? ? ?   3 ? ? ? ?   4 ? ? ? ?
0 9 A B C   1 6 A E I   2 ? ? ? ?   3 ? ? ? ?   4 ? ? ? ?
0 D E F G   1 7 B F J   2 ? ? ? ?   3 ? ? ? ?   4 ? ? ? ?
0 H I J K   1 8 C G K   2 ? ? ? ?   3 ? ? ? ?   4 ? ? ? ?
```

At this point we can see that for a pack with 5 symbols per card we need 21 different symbols and that our pack will contain `5+4*4=21` cards. In general a Dobble pack with `N` symbols per card will use $N+(N-1)^2$ symbols and have $N+(N-1)^2$ cards.

The sub-matrices for the 0 and 1 blocks are complete, and the sub-matrix for the 1 block is the transpose of that for the 0 block (rows of the 0 block have become columns of the 1 block). Now we use the orthogonal Latin squares of the Galois field G(4) to position the elements of the sub-matrix of block 1 in each of the remaining sub-matrices. These Latin squares are:

```
0  1  2  3      0  1  2  3      0  1  2  3
1  0  3  2      2  3  0  1      3  2  1  0
2  3  0  1      3  2  1  0      1  0  3  2
3  2  1  0      1  0  3  2      2  3  0  1
```

The top rows of these Latin squares tell us how to share the row `59DH` of sub-matrix 1 between the sub-matrices 2, 3 and 4.

```
0 1 2 3 4
0 5 6 7 8   1 5 9 D H   2 5 ? ? ?   3 5 ? ? ?   4 5 ? ? ?
```

```
0 9 A B C   1 6 A E I   2 ? 9 ? ?   3 ? 9 ? ?   4 ? 9 ? ?
0 D E F G   1 7 B F J   2 ? ? D ?   3 ? ? D ?   4 ? ? D ?
0 H I J K   1 8 C G K   2 ? ? ? H   3 ? ? ? H   4 ? ? ? H
```

The second rows of the Latin squares tell us how to share the row `6AEI`.

```
0 1 2 3 4
0 5 6 7 8   1 5 9 D H   2 5 A ? ?   3 5 ? E ?   4 5 ? ? I
0 9 A B C   1 6 A E I   2 6 9 ? ?   3 ? 9 ? I   4 ? 9 E ?
0 D E F G   1 7 B F J   2 ? ? D I   3 6 ? D ?   4 ? A D ?
0 H I J K   1 8 C G K   2 ? ? E H   3 ? A ? H   4 6 ? ? H
```

the third and forth rows of the Latin squares tell us how to share the the rows `7BFJ` and `8CGK`. We have now shared the symbols `5` to `K` between the rows of the 2 3 and 4 sub-matrices in such a way that we have one symbol in common between any two rows, and because the 0 sub-matrix is the transpose of the 1 sub-matrix, we also have one symbol in common between the rows of the 0 sub-matrix and the rows of the 2 3 and 4 matrices, and we have completed our solution:

```
0 1 2 3 4
0 5 6 7 8   1 5 9 D H   2 5 A F K   3 5 C E J   4 5 B G I
0 9 A B C   1 6 A E I   2 6 9 G J   3 8 9 F I   4 7 9 E K
0 D E F G   1 7 B F J   2 7 C D I   3 6 B D K   4 8 A D J
0 H I J K   1 8 C G K   2 8 B E H   3 7 A G H   4 6 C F H
```

7 Implementing the construction of a Dobble pack in Forth

Given a Galois field `G(p)`, we consider how to implement the construction a Dobble pack with `N=p+1` symbols per card, as outlined above. We allocate space for the Dobble pack, with 1 byte used to represent each symbol. Each run of `p+1` bytes records the symbols on a particular card. Initially we fill the pack with a value that would print as to show an unassigned symbol. [2]

[2]Though not shown here, for printing a pack we set BASE to 72, and character $\sim$, which is the highest allocated ascii code, corresponds to 71 in this base. This can be verified with: `71 72 BASE ! . DECIMAL`

```
p 1+  CONSTANT N   ( no. of symbols per card )
N  N 1 - DUP * + CONSTANT #PACK ( cards in pack,
and also no. of different symbols used in pack )

CREATE PACK #PACK N * ALLOT
PACK #PACK N * 71 FILL ( fill with a value that will display as ~ )
: CARD ( n -- a, a is addr of card n in PACK ) N * PACK + ;
```

On the same space we impose a different interpretation which allows us to easily access the column vectors and sub-matrices.

```
: V ( i j -- addr ) (: i j :) PACK N +  N N 1- * i * +  j N * + ;
: M ( i j k -- addr,
      addr is address of element (j,k) in matrix i of PACK )
  (: i j k :) PACK  N 1+ +  N N 1- * i * +  j N * +  k + ;
```

We require an implementation of the Latin squares associated with the field.

```
: LAT ( n i j -- k, k is i,jth element of fields nth Latin square)
  >R *0 R> +0 ;
```

We define operations to build the pack. These fill in the first card, store the elements 0 to p onto the cards (completing the constant column vectors), fill in sub-matrix 0, copy the transpose of sub-matrix 0 to matrix 1, and complete the remaining sub matrices using their associated Latin squares.

```
: !CARD0 N 0 DO I PACK I + C! LOOP ;

: !0..p ( store symbols 0 to p onto cards )
  N 0 DO p 0 DO J   J I V C! LOOP  LOOP ;

: !M0 ( --, Set elements of matrix 0 of pack ) N
   N 1- 0 DO N 1- 0 DO
     DUP  0 J I M C!  1+
   LOOP LOOP DROP ;

: M0_Trans_toM1 ( assign transpose of M0 to M1 )
  N 1- 0 DO  N 1- 0 DO
     0 I J M C@   1 J I M C!
```

```
  LOOP LOOP ;

: LATINISE
   N 1- 1 DO
     N 1- 0 DO
       N 1- 0 DO
                    1 I J M C@   K 1+  K I J LAT  J  M  C!
                LOOP
        LOOP
    LOOP ;
```

We now have everything we need to build a pack.

```
: BUILD-PACK !CARD0 !0..p !M0 M0_Trans_toM1 LATINISE ;
```

A quickly implemented debug aid that allows us to look at the memory where the Dobble pack is being constructed is a “Dobble dump", i.e. a `DUMP` modified to use the number of symbols in the pack as its output base.

```
: DDUMP BASE @ >R #PACK BASE !  DUMP R> BASE ! ;
```

To verify our pack we first need an operation to check how many symbols are shared by a pair of cards.

```
: SHARED ( card1 card2 -- n, pre: card1 <> card2
   post: n is no of symbols shared by card1 and card2 of pack )
   (: card1 card2 :) 0
   N 0 DO
     N 0 DO card1 CARD I + C@ card2 CARD J + C@  =
       IF  1+  THEN
     LOOP
   LOOP ;
```

To check the pack we look at every pair of cards and verify that they share exactly one symbol.

```
: CHECK-PACK ( --, report first pair of cards found to differ
by anything other than one symbol )
```

```
0  #PACK 0 DO   #PACK 0 DO
  I J > IF
    I J  SHARED 1  <> IF
     CR ." Cards " I . J . ." differ in"  I J SHARED . ." symbols"
     CR ." Cards are "  I .CARD ." and "  J .CARD  ABORT
   ELSE
     ." ." 1+
   THEN
  THEN
LOOP LOOP CR . ." Checks, pack validated " ;
```

8 Conclusions

Galois fields are rich finite algebraic structures based on modular and polynomial arithmetic and are useful in many applications. Here we have described their properties and shown how they can be implemented. Mathematical properties relating to the existence of such fields and their associated sets of orthogonal Latin squares are are proved in the appendix. We have looked in detail at how the orthogonal Latin squares associated with each Galois field can be used to construct a Dobble pack. Forth proved a versatile programming tool, able to model the complexities of the Dobble pack by overlaying alternative interpretations of the data in a block of memory in a transparent and simple way. Forth's ability to work in a number of bases was made use of: we used binary for representing encodings of polynomials with modulo 2 coefficients, hexadecimal for the compact encoding higher order polynomials, and base 72 output to print the symbols on our cards. This last was useful as it allows us to print the symbols on our cards as numbers, with numbers up to 71 appearing as 71 different single output symbols. This allowed us to concentrate on the theory underlying the implementation, and this is where the beauty of this example lies, rather than in spending time making the representation of cards more attractive. The 71 ascii symbols used were perfectly sufficient to verify our code. We would note, however, that using such a high number base does not work well for input, since it produces conflicts between numbers and the names of Forth operations.

Mathematical Appendix

Modulo p arithmetic where p is not prime.

Arithmetic modulo $\mathtt{p}$ yields a Galois Field if and only if $\mathtt{p}$ is prime. To see why suppose (in order to obtain a contradiction) that modulo $\mathtt{p}$ arithmetic does yield a Galois Field $\mathcal{F}$ when $\mathtt{p}$ is non-prime. When $\mathtt{p}$ is non-prime it has factors, $\mathtt{u}$ and $\mathtt{v}$ say such that $\mathtt{u}, \mathtt{v} \in \mathcal{F}$ and $\mathtt{u} *_0 \mathtt{v} = \mathtt{p}$. Now recall that $\mathtt{u} *_0 \mathtt{v} = (\mathtt{u} * \mathtt{v}) \bmod \mathtt{p}$ and since $\mathtt{u} * \mathtt{v} = \mathtt{p}$ we have $\mathtt{u} *_0 \mathtt{v} = \mathtt{p} \bmod \mathtt{p} = 0$:

But if $\mathtt{u}, \mathtt{v} \in \mathcal{F}$ they have inverses $\mathtt{u}^{-1}$ and $\mathtt{v}^{-1}$ and so:

$(\mathtt{u} *_0 \mathtt{u}^{-1}) *_0 (\mathtt{v} *_0 \mathtt{v}^{-1}) = 1 *_0 1 = 1$

But on the other hand:, by the commutativity and associativity laws of a Galois Field:

$(\mathtt{u} *_0 \mathtt{u}^{-1}) *_0 (\mathtt{v} *_0 \mathtt{v}^{-1})$ = "by comm and ass laws"
$(\mathtt{u} *_0 \mathtt{v}) *_0 (\mathtt{u}^{-1} *_0 \mathtt{v}^{-1})$ = "since u $*_0$ v = 0"
$0 *_0 (\mathtt{u}^{-1} *_0 \mathtt{v}^{-1})$ = "by multiplicative property of 0"
0

and the contradiction is established.

Latin square properties

Given a Galois field $\mathcal{F}$ of size $\mathtt{n}$ with elements $0..\mathtt{n}-1$ there are $\mathtt{n}-1$ mutually orthogonal Latin squares of size $\mathtt{n} \times \mathtt{n}$ such that the value in row $\mathtt{j}$ and column $\mathtt{k}$ of the $\mathtt{ith}$ such square is given by $\mathtt{i} *_0 \mathtt{j} +_0 \mathtt{k}$ where $\mathtt{i} \in 1..\mathtt{n}-1$ and $\mathtt{j}, \mathtt{k} \in 0..\mathtt{n}-1$.

Proof that each square is a Latin square

By closure properties $\mathtt{i} *_0 \mathtt{j} +_0 \mathtt{k} \in \mathcal{F}$ which leaves us to prove that for an arbitrary square $\mathtt{i}$ the same element cannot occur twice in any row, or twice in any column.

To show an element cannot occur twice in a row, note that the element at row $\mathtt{j}$ column $\mathtt{k}$ of square $\mathtt{i}$ is $\mathtt{i} *_0 \mathtt{j} +_0 \mathtt{k}$. Assume that the element at row $\mathtt{j}'$ in the same column has the same value, and assume $\mathtt{j} \neq \mathtt{j}'$. Then

$\mathtt{i} *_o \mathtt{j} +_o \mathtt{k} = \mathtt{i} *_o \mathtt{j}' +_o \mathtt{k} \equiv$ "subtracting $\mathtt{k}$ from both sides"
$\mathtt{i} *_o \mathtt{j} = \mathtt{i} *_o \mathtt{j}' \equiv$ "dividing both sides by $\mathtt{i}$"
$\mathtt{j} = \mathtt{j}'$ "contradicting our assumption"

So the only way the element at row $\mathtt{j}$ and col $\mathtt{k}$ of an arbitrary square can be equal to the element at row $\mathtt{j}$ and col $\mathtt{k}'$ of that square is if $\mathtt{j} = \mathtt{j}'$, i.e. if they are the same square.

To show an element cannot occur twice in a column, assume that the element at row $\mathtt{j}$ and col $\mathtt{k}$ of square $\mathtt{i}$ is the same as that at row$\mathtt{j}$ and col $\mathtt{k}'$, and assume $\mathtt{k} \neq \mathtt{k}'$. Then

$\mathtt{i} *_o \mathtt{j} +_o \mathtt{k} = \mathtt{i} *_o \mathtt{j} +_o \mathtt{k}' \equiv$ "subtracting $\mathtt{i} *_o \mathtt{j}$ from both sides"
$\mathtt{k} = \mathtt{k}'$ "contradicting our assumption"

Proof that any two distinct squares are orthogonal.

For the squares $\mathtt{i}$ and $\mathtt{i}'$ to be orthogonal we need the pair of values:

$(\mathtt{i} *_o \mathtt{j} +_o \mathtt{k}, \mathtt{i}' *_o \mathtt{j} +_o \mathtt{k})$

to be distinct from

$(\mathtt{i} *_o \mathtt{j}' +_o \mathtt{k}', \mathtt{i}' *_o \mathtt{j}' +_o \mathtt{k}')$.

To obtain a contradiction we will assume these pairs of values are not distinct, i.e.

$\mathtt{i} *_o \mathtt{j} +_o \mathtt{k} = \mathtt{i} *_o \mathtt{j}' +_o \mathtt{k}'$ (1) and $\mathtt{i}' *_o \mathtt{j} +_o \mathtt{k} = \mathtt{i}' *_o \mathtt{j}' +_o \mathtt{k}'$ (2).
Then from (1)

$\mathtt{i} *_o \mathtt{j} +_o \mathtt{k} -_o \mathtt{i} *_o \mathtt{j}' -_o \mathtt{k}' = \mathtt{0} \equiv$ "subtracting $\mathtt{i} *_o \mathtt{j}' +_o \mathtt{k}'$ from each side of (1)"
$\mathtt{i} *_o (\mathtt{j} -_o \mathtt{j}') +_o \mathtt{k} -_o \mathtt{k}' = \mathtt{0}$ (3)

"also similarly from (2)"
$\mathtt{i}' *_o (\mathtt{j} -_o \mathtt{j}') +_o \mathtt{k} -_o \mathtt{k}' = \mathtt{0}$ (4)
"Now subtracting (4) from (3) and factorising"
$(\mathtt{i} -_o \mathtt{i}') *_o (\mathtt{j} -_o \mathtt{j}') = \mathtt{0} \equiv$ "since $\mathtt{i} \neq \mathtt{i}'$"
$\mathtt{j} = \mathtt{j}'$

Now substituting $\mathtt{j}$ for $\mathtt{j}'$ in (1) we obtain

$\mathtt{i} *_o \mathtt{j} +_o \mathtt{k} = \mathtt{i} *_o \mathtt{j} +_o \mathtt{k}' \equiv$ "subtracting $\mathtt{i} *_o \mathtt{j}$ from each side
$\mathtt{k} = \mathtt{k}'$

Square Roots

In section 4 we give multiplication tables for possible implementations of G(8), and it can be seen from these that every element has an exact square root.

More generally this is true for any field $G(2^n)$, and this comes from the observation that since, in modulo 2 arithmetic, addition is synonymous with subtraction, this property will also hold for polynomial arithmetic on polynomials with modulo 2 coeffiecients.

To ensure that every element of such a field has a square root it is sufficient to show that for any p, q in the field: $p^2 = q^2 \Rightarrow p = q$

Proof. We begin with the assumption that $p^2 = q^2$

$p^2 = q^2 \equiv$ "subtracting q^2 from each size"
$p^2 -_o q^2 = 0 \equiv$ "factorising"
$(p -_o q) *_o (p +_o q) = 0 \equiv$ "since $p +_o q = p -_o q$"
$(p -_o q) *_o (p -_o q) = 0 \equiv$ "property of additive inverse"
$p = q$ □

Proof of field properties

We don't attempt a mathematical proof that the polynomial arithmetic we have described yields the operations of a Galois field. However, for any particular field that we implement we can prove the axioms hold by verifying them for all combinations of values in the field. The code for this verification just needs a single parameter, p, which gives the size of the field.

For these tests we use +_ and *_ as our operation names. The names of our actual field operations might be $+_0$ and $*_0$ or $+_1$ and $*_1$, so we use these to define +_ and *_ before loading the field tests.

The most complex tests are to establish the existence and uniqueness of inverses. Here is the code for checking the multiplicative inverse axiom.

```
: (*INV) ( n -- f, pre: n ∈ 1 .. p-1
 post: f true iff n has a unique  multiplicative inverse)
 (: n :) 0 ( count of inverses )
  p 1 DO  I  n   *_   1 = IF 1+ THEN LOOP  1 = ;

: *INV-TEST  ( -- f, true if every element of 1 .. p-1 has a unique
```

```
multiplicative inverse in arithmetic modular p)
  TRUE  p 1 DO
    I  (*INV) NOT IF  DROP FALSE LEAVE THEN
  LOOP  ;
```

Given that we already know, from established mathematical results, that the field axioms will hold, the real benefit of verifying them is as a check that our field operations have been correctly implemented.