Some math for embedded algorithms

- CRC (Cyclic Redundancy Check) is often used to check integrity of transmitted messages
- Hamming codes can *correct* one error
- BCH codes can correct more
- LFSRs (Linear Feedback Shift Registers) are used to generate pseudo-random sequences, used in communications and other apps

What do these algorithms have in common?

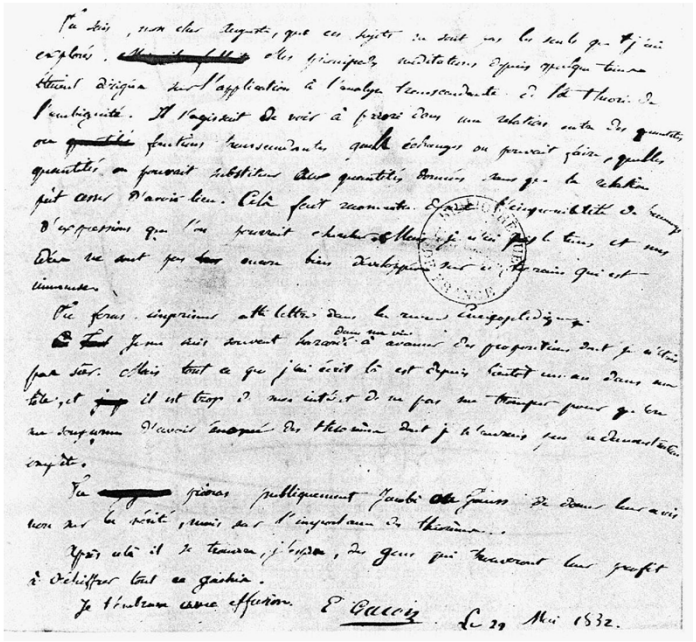
They are all based on ***finite field arithmetic***, specifically Galois Fields, which use polynomial algebra

Historical interlude

Evariste Galois

From his last letter:

"Ask Jacobi or Gauss publicly to give their opinion, not as to the truth, but as to the importance of these theorems. Later there will be, I hope, some people who will find it to their advantage to decipher all this mess…"



Killed in a duel at age 20

Galois theory addresses the question of which polynomial equations can be solved by formulas with radicals (there are quadratic, cubic, quartic formulas; quintic, only sometimes). Also shows which polygons can be constructed with ruler and compass.

The machinery of polynomial algebra and "Galois Fields" turns out to be very useful for certain embedded algorithms!

See Galois Theory, 3rd Ed., I. Stewart

Why the *&^!@ does everyone talk about polynomials whenever codes come up?

- ***Finite fields,*** are a good idea for encoding digital data (e.g. for implementing linear codes): inputs and outputs to/from a computer or channel are discrete & finite; linear codes are defined in terms of addition and multiplication operations
- Informally, a ***field*** is a set that supports addition, multiplication, and the distributive property
- And why do we need Galois Fields (based on polynomial algebra) for coding?
- What's wrong with integer arithmetic mod N? That's linear and finite. Shouldn't that work for coding?
- Let's see what happens!
  - Consider a bunch of addition and multiplication operations on random data, mod 8!

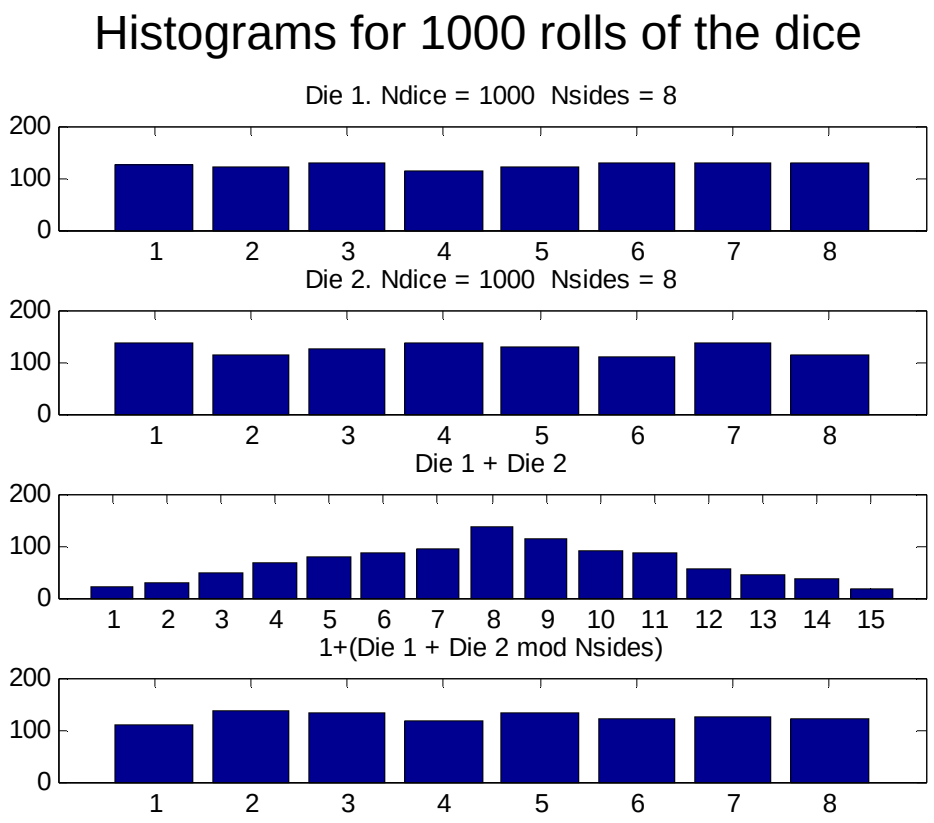
Adding a pair of 8-sided dice, mod 8

Die 1

Die 2

Die 1 + Die 2

(Die 1 + Die 2) mod 8



Pretty Uniform!

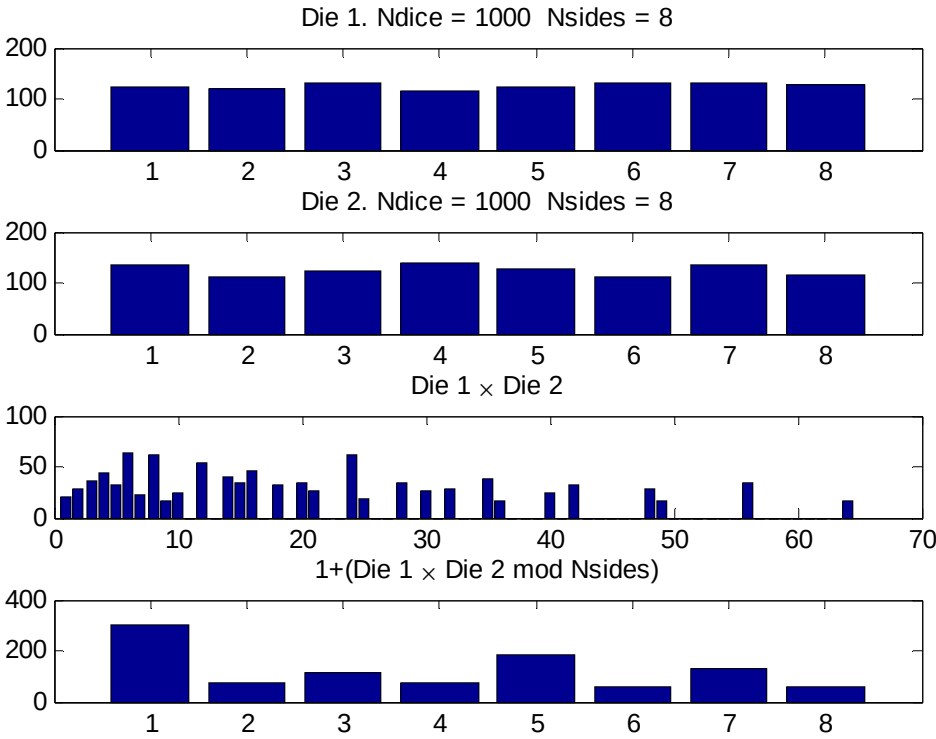
Multiplying a pair of 8-sided dice, mod 8

Die 1

Die 2

Die 1 x Die 2

(Die 1 x Die 2) mod 8



yuck! Big mess!

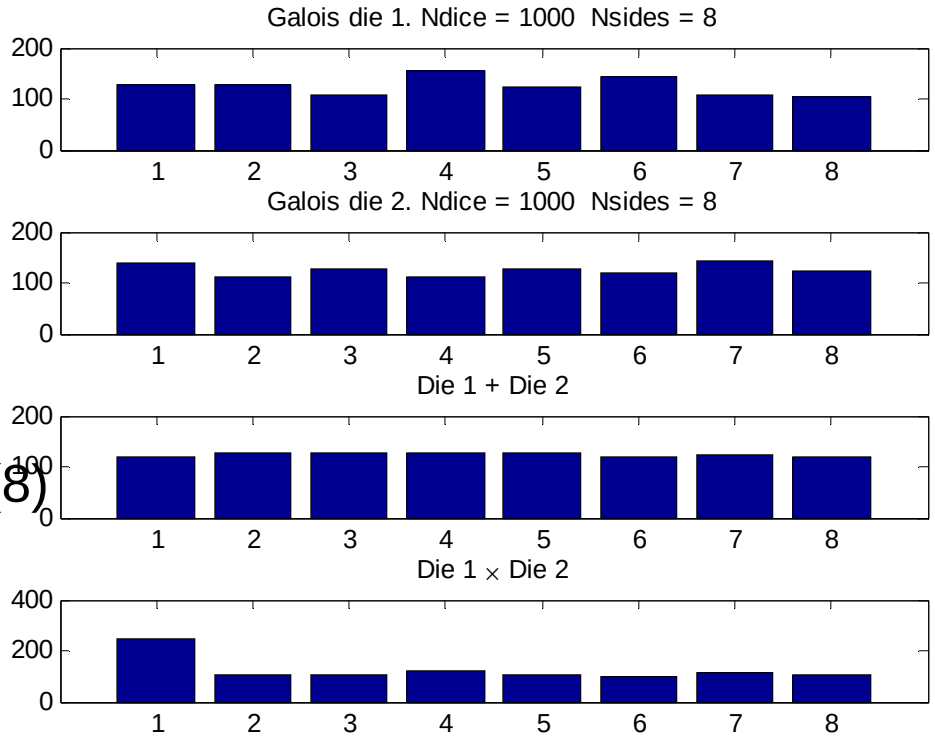
Adding & Multiplying a pair of 8-sided dice, in GF(8)

Die 1

Die 2

Die 1 + Die 2 (GF(8))

(Die 1 x Die 2) in GF(8)



Wow! Very Uniform!

Comparing the multiplication tables

Mod 8

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

Some rows have repeated elements

GF(8)

X	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

Each # appears once per row

What is the rule for constructing this magic multiplication table?

- Interpret the bits as coefficients (1 or 0) of a polynomial
- Interpret coefficients as mod 2
- Multiply the polynomials
- Take result mod x^3+x+1

X	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

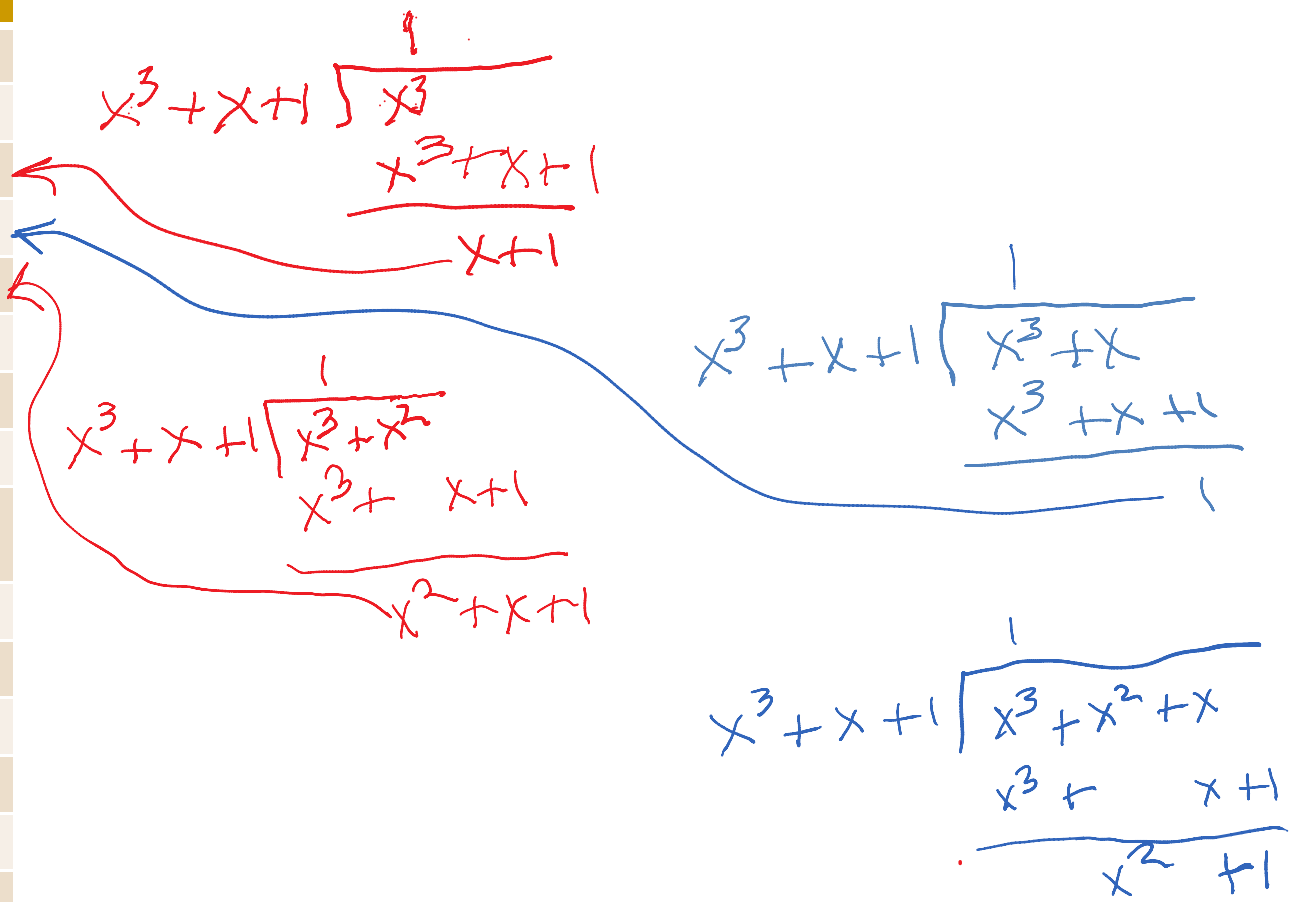
Elt.	Binary Rep.	Polynomial
0	000	0
1	001	1
2	010	x
3	011	x+1
4	100	x^2
5	101	x^2+1
6	110	x^2+x
7	111	x^2+x+1

Check the table!

Elt.	Polynomial	Raw product	Coefs mod 2
2*2	x*x	x^2	x^2
2*3	x*(x+1)	x^2+x	x^2+x
2*4	x*x^2	x^3	x^3
2*5	x*(x^2+1)	x^3+x	x^3+x
2*6	x*(x^2+x)	x^3+x^2	x^3+x^2
2*7	x*(x^2+x+1)	x^3+x^2+x	x^3+x^2+x
3*3	(x+1)*(x+1)	x^2+2x+1	x^2+1
3*4	(x+1)*x^2	x^3+x^2	x^3+x^2
3*5	(x+1)*(x^2+1)	x^3+x^2+x+1	x^3+x^2+x+1
3*6	(x+1)*(x^2+x)	x^3+2x^2+x	x^3+x
3*7	(x+1)*(x^2+x+1)	x^3+2x^2+2x+1	x^3+1
4*4	x^2*x^2	x^4	x^4
4*5	x^2*(x^2+1)	x^4+x^2	x^4+x^2
4*6	x^2*(x^2+x)	x^4+x^3	x^4+x^3
4*7	x^2*(x^2+x+1)	$x^4+x^3+x^2$	$x^4+x^3+x^2$
5*5	(x^2+1)*(x^2+1)	x^4+2x^2+1	x^4+1
5*6	(x^2+1)*(x^2+x)	$x^4+x^3+x^2+x$	$x^4+x^3+x^2+x$
5*7	(x^2+1)*(x^2+x+1)	$x^4+x^3+2x^2+x+1$	x^4+x^3+x+1
6*6	(x^2+x)*(x^2+x)	$x^4+2x^3+x^2$	x^4+x^2
6*7	(x^2+x)*(x^2+x+1)	$x^4+2x^3+2x^2+x$	x^4+x
7*7	(x^2+x+1)*(x^2+x+1)	$x^4+2x^3+3x^2+2x+1$	x^4+x^2+1

Check the table!

Elt.	Coefs mod 2	Mod x^3+x+1
2*2	x^2	x^2
2*3	x^2+x	x^2+x
2*4	x^3	$x+1$
2*5	x^3+x	1
2*6	x^3+x^2	x^2+x+1
2*7	x^3+x^2+x	x^2+1
3*3	x^2+1	
3*4	x^3+x^2	
3*5	x^3+x^2+x+1	
3*6	x^3+x	
3*7	x^3+1	
4*4	x^4	
4*5	x^4+x^2	
4*6	x^4+x^3	
4*7	$x^4+x^3+x^2$	
5*5	x^4+1	
5*6	$x^4+x^3+x^2+x$	
5*7	x^4+x^3+x+1	
6*6	x^4+x^2	
6*7	x^4+x	



Check the table!

Elt.	Coefs mod 2	Mod x^3+x+1	= Elt.
2*2	x^2	x^2	4
2*3	x^2+x	x^2+x	6
2*4	x^3	$x+1$	3
2*5	x^3+x	1	1
2*6	x^3+x^2	x^2+x+1	7
2*7	x^3+x^2+x	x^2+1	5
3*3	x^2+1		
3*4	x^3+x^2		
3*5	x^3+x^2+x+1		
3*6	x^3+x		
3*7	x^3+1		
4*4	x^4		
4*5	x^4+x^2		
4*6	x^4+x^3		
4*7	$x^4+x^3+x^2$		
5*5	x^4+1		
5*6	$x^4+x^3+x^2+x$		
5*7	x^4+x^3+x+1		
6*6	x^4+x^2		
6*7	x^4+x		

Elt.	Binary Rep.	Polynomial
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

X	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

Discussion

- Finite fields exist for sets of size p^n where p is prime
- We are very lucky that 2 is a prime number! Otherwise this wouldn't make sense.

To be continued soon…

With how Galois Field arithmetic used in embedded systems for CRC, LFSR, and other algorithms…

More info

C

Some Mathematics

C.1 Finite field theory

Most linear codes are expressed in the language of Galois theory

Why are Galois fields an appropriate language for linear codes? First, a definition and some examples.

A field F is a set $F = \{0, F'\}$ such that

1. F forms an Abelian group under an addition operation '+', with 0 being the identity; [Abelian means all elements commute, i.e., satisfy $a + b = b + a$.]
2. F' forms an Abelian group under a multiplication operation '·'; multiplication of any element by 0 yields 0;
3. these operations satisfy the distributive rule $(a + b) \cdot c = a \cdot c + b \cdot c$.

For example, the real numbers form a field, with '+' and '·' denoting ordinary addition and multiplication.

A Galois field $GF(q)$ is a field with a finite number of elements q.

A unique Galois field exists for any $q = p^m$, where p is a prime number and m is a positive integer; there are no other finite fields.

$GF(2)$. The addition and multiplication tables for $GF(2)$ are shown in ta-

+	0	1
0	0	1
1	1	0

·	0	1
0	0	0
1	0	1

Table C.1. Addition and multiplication tables for $GF(2)$.

+	0	1	A	B
0	0	1	A	B
1	1	0	B	A
A	A	B	0	1
B	B	A	1	0

·	0	1	A	B
0	0	0	0	0

Try the math dept, or

D.J.C. MacKay
"Information Theory, Inference, and Learning Algorithms"
Cambridge U. Press, 2003
Appendix C,
Some Mathematics

I. Stewart
Galois Theory, 3rd Ed
Chapman & Hall / CRC, 2004

CRC example

Data	CRC-CCITT Poly: 0x1021	CRC-16 Poly: 0x8005	CRC-32 Poly: 0x4C11DB7
The quick brown fox jumped	0x1C20	0x10C2	0x7E9D4E99
The quick brown fox jumped!	0xC7FE	0x8951	0x17C6497D
The quick fox brown jumped	0xB5B9	0x7A20	0x114792B3

Computed using
http://zorc.breitbandkatze.de/crc.html

CRC

- CRC is a set of *u* parity bits appended to a message
- Treat data bits as polynomial, *d(x)* ("data")
- Pad data with *u* zeros (multiply data poly by x^u)
- Our magic checking polynomial is *g(x)* ("generator")
- CRC *r(x)* ("remainder") defined implicitly by

$$m(x) = \frac{d(x)x^u + r(x)}{g(x)}$$

r(x) added to padded data makes it divisible by *g(x).* We don't care about *m(x)*

- Solve above equation to get explicit expression for *r(x)*

$$r(x) = g(x)m(x) - d(x)x^u = d(x)x^u - g(x)m(x)$$

Common CRC polynomials

- CCITT CRC-16 $x^{16}+x^{12}+x^5+1$
- CRC-5-USB $x^5 + x^2 + 1$ (USB token packets)
- CRC-8-CCITT $x^8 + x^2 + x + 1$ ISDN Header Error Control
- CRC-32 $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
  Ethernet, SATA, MPEG-2, Gzip, PKZIP, PNG
- CRC-40-GSM $x^{40} + x^{26} + x^{23} + x^{17} + x^3 + 1$
  GSM Control Channel

And many more…see
http://en.wikipedia.org/wiki/Cyclic_redundancy_check

CRC example

```
11010011101100 000 <---input, padded 3 bits
1011                <--- divisor
01100011101100 000 <--- result
 1011               <--- divisor
00111011101100 000
  1011
00010111101100 000
   1011
00000001101100 000
       1011
00000000110100 000
        1011
00000000011000 000
         1011
00000000001110 000
          1011
00000000000101 000
           101 1
00000000000000 100 <---remainder (3 bits)
```