Computational Mathematics: Handout 06

Curtis Bright

October 3, 2022

1 Finite Fields and Reed–Solomon Codes

This handout will provide a brief introduction to *finite fields* and how they can be applied to Reed–Solomon codes. Reed–Solomon codes are a popular source of codes used in coding theory; for example, they are used in DVDs and Blu-ray discs. The parameters of the coding scheme can be chosen in order to allow correcting an arbitrarily large number of errors.

1.1 Finite Fields

Recall a *field* is a ring that supports division by nonzero elements. A field is *finite* if it contains a finite number of elements. For example, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ are fields, but they all have an infinite number of elements. Clearly $\mathbb{Z}$ is *not* a field (since only ± 1 are have inverses) but the integers modulo m (denoted $\mathbb{Z}_m$ or $\mathbb{Z}/m\mathbb{Z}$) may or may not be a field.

We've seen that an inverse of $x \in \mathbb{Z}_m$ exists exactly when $\gcd(x, m) = 1$, i.e., x and m are coprime. If $m = p$ is a prime then the only x with $\gcd(x, p) \neq 1$ are the multiples of p, i.e., the x that satisfy $x \equiv 0 \pmod{p}$. It follows that an inverse exists for any $x \in \{1, 2, \ldots, p-1\} = \mathbb{Z}_p \setminus \{0\}$ and thus that $\mathbb{Z}_p$ is a field when p is prime. It has p elements and is often denoted by the complete system of representatives $\{0, 1, \ldots, p-1\}$.

On the other hand, if m is *not* a prime, then there must be some $x \in \{1, 2, \ldots, p-1\}$ that is a divisor of p and it follows that $\gcd(x, p) \neq 1$, i.e., that $x^{-1} \bmod p$ does not exist. Thus, $\mathbb{Z}_m$ is *not* a field when m is not prime.

1.1.1 Takeaway

Thus, a finite field with p elements exists for any prime p and is commonly denoted $\mathbb{Z}_p$ or $\mathbb{F}_p$. We've also seen that these are the *only* fields of the form $\mathbb{Z}_m$.

But it is conceivable that some other kind of finite fields exist that are not of the form $\mathbb{Z}_m$. This is a very interesting question: Do any other finite fields exist? The answer is yes. To see how they can be constructed, we review the construction of $\mathbb{C}$ from $\mathbb{R}$.

1.2 From the Real to the Complex

How can the complex numbers be constructed from the real numbers? The standard construction is to add a new element i that satisfies $i^2 = -1$, i.e., a root of the $\mathbb{R}$-unsolvable equation $x^2 + 1 = 0$. Alternatively, complex numbers can be defined as tuples (r_1, i_1) that can be multiplied via the rule

$$(r_1, i_1) \cdot (r_2, i_2) := (r_1 r_2 - i_1 i_2, r_1 i_2 + i_1 r_2)$$

which follows after simplifying $(r_1 + i_1 i)(r_2 + i_2 i) = r_1 r_2 + (r_1 i_2 + i_1 r_2)i + i_1 i_2 i^2$ with $i^2 = -1$.

There is another way of viewing this that will be particularly useful when constructing general finite fields.

Consider the ring of polynomials with real coefficients ($\mathbb{R}[x]$) but peform all computations "modulo $x^2 + 1$". A polynomial mod $x^2 + 1$ will always produce a remainder polynomial of degree 0 or 1, i.e., a polynomial of the form $a + bx$. This construction is known as a "quotient ring" and is commonly denoted by $\mathbb{R}[x]/\langle x^2 + 1\rangle$. Performing computations mod $x^2 + 1$ is equivalent to subtracting off multiples of $x^2 + 1$, or equivalently treating any multiples of $x^2 + 1$ as **zero**. In other words, setting $x^2 + 1 = 0$. Look familiar? This is equivalent to $x^2 = -1$ which looks suspiciously like the definition of i. In fact, the two methods are equivalent; we have that

$$\mathbb{C} \cong \mathbb{R}[x]/\langle x^2 + 1\rangle$$

where $\cong$ denotes the two structures are isomorphic; any expression in the left structure has an equivalent expression in the right structure. The only difference is that the imaginary unit i on the left has been "renamed" to x on the right. Otherwise the arithmetic in both is exactly the same.

1.3 From $\mathbb{F}_p$ to $\mathbb{F}_{p^2}$

The quotient construction used to construct the complex numbers from the reals is very similar to how we can construct larger finite fields starting from the known finite fields $\mathbb{F}_p$. Instead of $x^2 + 1$ we consider an *irreducible* polynomial of degree 2, i.e., one that does not factor into linear factors. Over the real numbers $x^2 + 1$ is irreducible, but over $\mathbb{F}_p$ this polynomial may or may not be irreducible.

For example, in $\mathbb{F}_2$ we have that $x^2 + 1 = (x + 1)^2$. (The "freshman's dream" comes true.) Similarly, in $\mathbb{F}_5$ we have $x^2 + 1 = (x + 2)(x + 3)$, but in $\mathbb{F}_3$ the polynomial $x^2 + 1$ does not factor any farther so it is irreducible.

Consider the quotient ring $\mathbb{F}_p[x]/\langle f\rangle$ where f has degree 2 and is irreducible over $\mathbb{F}_p$. By performing all computations modulo f we can reduce any polynomial in $\mathbb{F}_p[x]$ to one of degree 0 or 1 by using the division algorithm (which always works in a field). That is, we have

$$\mathbb{F}_p[x]/\langle f\rangle \cong \{a + bx : a, b \in \mathbb{F}_p\}$$

and thus $\mathbb{F}_p[x]/\langle f\rangle$ has p^2 elements. Is it actually a field, though? For that to be the case, any nonzero polynomial $g = a + bx$ would need to have an inverse, i.e., the following congruence must be solvable:

$$gh \equiv 1 \pmod{f}.$$

In other words, there must exist $h, k \in \mathbb{F}_p[x]$ such that $gh + kf = 1$. Using the extended Euclidean algorithm on g and f will solve this equation, assuming that $\gcd(g, f) = 1$. However, since f is irreducible and g is nonzero and of lower degree than f we do in fact have $\gcd(g, f) = 1$.

Thus, we have constructed a finite field $\mathbb{F}_p[x]/\langle f\rangle$ with p^2 elements. It is typically denoted $\mathbb{F}_{p^2}$. This might seem strange (you'd think f would have to be part of the notation) but in fact one can show the surprising fact that there is *at most one* finite field of each order up to isomorphism. In other words, if f and g are two irreducible polynomials of degree 2 then $\mathbb{F}_p[x]/\langle f\rangle \cong \mathbb{F}_p[x]/\langle g\rangle$ even though the arithmetic in these two rings will *look* different, any element $a + bx$ in the former can be mapped onto some other $c + dx$ in the latter that will have the exact same algebraic behaviour.

1.4 All finite fields

This construction also works if f is an irreducible polynomial of degree n for any $n \geq 2$. The only difference is that the construction will produce

$$\mathbb{F}_p[x]/\langle f \rangle \cong \left\{ a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} : a_i \in \mathbb{F}_p \right\}$$

and so in this case $\mathbb{F}_p[x]/\langle f \rangle$ has p^n elements. Again, up to ismorphism there is one finite field with p^n elements and it is commonly denoted $\mathbb{F}_{p^n}$.

One can also show that no other finite fields exist. For example, there is no finite field with 6 elements, since 6 is not a prime power.

Thus, we have a complete classification of the finite fields. If the field size is a prime power $q = p^n$ then a finite field exists and is isomorphic to $\mathbb{F}_q$.

Warning: It is important to realize that $\mathbb{Z}_q$ also has q elements but is **not** a field (except in the case $q = p$) since it contains nonzero elements without an inverse. So $\mathbb{Z}_q = \mathbb{F}_q$ when q is prime and $\mathbb{Z}_q \neq \mathbb{F}_q$ when $q = p^n$ for $n \geq 2$.

1.5 Primitive elements

One last fact about finite fields will be useful: they always have an element α for which any nonzero element of the field can be written as a power of α.

In other words, in $\mathbb{F}_{p^n}$ there is some element α so that the list

$$\alpha, \alpha^2, \ldots, \alpha^{p^n - 1}$$

consists of all the nonzero elements of $\mathbb{F}_{p^n}$. The set of nonzero elements of $\mathbb{F}_{p^n}$ is commonly denoted $\mathbb{F}_{p^n}^*$ and α is said to *generate* $\mathbb{F}_{p^n}^*$.

1.6 Finite Fields in Sage

Finite fields of order q can be defined in Sage via `GF(q)`. By default Sage will use an appropriate irreducible polynomial in the construction but you can also explicitly set the irreducible polynomial to use with the `modulus` argument. The elements of the field will be written as polynomials in a variable `a` which can be set by the `name` argument or alternatively with the notation `F.<a> = GF(q)`.

[1]:
```
F.<a> = GF(2^4) # F is defined to be a finite field of order 2^8
alpha = F.primitive_element() # Get a primitive element
[alpha^i for i in (1..2^4-1)] # List all nonzero elements
```

[1]:
```
[a,
 a^2,
 a^3,
 a + 1,
 a^2 + a,
 a^3 + a^2,
 a^3 + a + 1,
 a^2 + 1,
```

```
a^3 + a,
a^2 + a + 1,
a^3 + a^2 + a,
a^3 + a^2 + a + 1,
a^3 + a^2 + 1,
a^3 + 1,
1]
```

Equivalently, the elements of the finite field $\mathbb{F}_{p^n}$ (polynomials of degree strictly less than n) can be considered as vectors of length n over $\mathbb{F}_p$, i.e.,

$$\mathbb{F}_{p^n} \cong \mathbb{F}_p^n.$$

For example, $1 + a + a^3$ in $\mathbb{F}_{2^4}$ would be represented by the vector $[1, 1, 0, 1]$.

[2]:
```
V = F.vector_space(map=False) # A vector space of dimension 10 over GF(2)
[V(alpha^i) for i in (1..2^4-1)] # List all nonzero elements of GF(2^10) as␣
↪vectors
```

[2]:
```
[(0, 1, 0, 0),
 (0, 0, 1, 0),
 (0, 0, 0, 1),
 (1, 1, 0, 0),
 (0, 1, 1, 0),
 (0, 0, 1, 1),
 (1, 1, 0, 1),
 (1, 0, 1, 0),
 (0, 1, 0, 1),
 (1, 1, 1, 0),
 (0, 1, 1, 1),
 (1, 1, 1, 1),
 (1, 0, 1, 1),
 (1, 0, 0, 1),
 (1, 0, 0, 0)]
```

1.7 Reed–Solomon and BCH Codes

Reed–Solomon codes are based on performing computations in finite fields. We will outline the Bose–Chaudhuri–Hocquenghem codes (BCH) codes which are a kind of Reed–Solomon code.

In a BCH code the message you want to send is encoded in the coefficients of a polynomial. For example, to send $[1, 1, 0, 1, 0, 1]$ you could represent this as the polynomial $1 + y + y^3 + y^5$ in $\mathbb{F}_2[y]$.

To "encode" a message you multiply it by a "generator" polynomial g. The recipient of the message can then divide the polynomial that they recieve by g. If they get a remainder of 0 this indicates that the message was almost certainly recieved correctly. If they get a nonzero remainder this indicates that at least one coefficient of the recieved message was corrupted. We will see how the recipient can determine which of the coefficients were corrupted (assuming there were at most t corrupted coefficients, where t is a constant chosen in advance).

First, fix an n which will be an upper bound on the degree of the polynomials in the code. A finite field should be chosen that has primitive nth roots of unity. For example, we can take $n = 2^{10} - 1 = 1023$ because a primitive element α of $\mathbb{F}_{2^{10}}$ will be a primitive nth root of unity.

1.7.1 Generator polynomial

Let's say we want to correct at most t errors. The generator polynomial g will be chosen so that α, $\alpha^2, \ldots, \alpha^{2t}$ are all roots of g. (Recall α is a primitive nth root of unity.)

The Sage method `minpoly` returns a polynomial (with coefficients in the base ring) of minimal degree that has a given element as a root. For example, `alpha.minpoly()` will give the smallest polynomial with coefficients in $\mathbb{F}_2$ for which α is a root.

The smallest polynomial in $\mathbb{F}_2[y]$ that has all $\alpha, \ldots, \alpha^{2t}$ as roots will be the least common multiple (lcm) of the minimal polynomial of each of $\alpha, \ldots, \alpha^{2t}$.

[3]:
```
n = 1023
F.<a> = GF(2^10)
alpha = F.primitive_element()
assert(alpha^n == 1)
t = 1
g = lcm([(alpha^i).minpoly('y') for i in (1..2*t)])
print(alpha)
print(g)
```

```
a
y^10 + y^6 + y^5 + y^3 + y^2 + y + 1
```

1.7.2 Encoding a message via multiplication

Suppose m is a message of 256 bits encoded via the coefficients of a polynomial in $R := \mathbb{F}_2[y]$.

[4]:
```
R.<y> = GF(2)[] # The message will be sent as a polynomial in a new variable y
m = R.random_element(degree=255) # Generate a random message with 256 bits
print(m) # Message as a polynomial
print(m.list()) # Message as a list
```

```
y^255 + y^254 + y^251 + y^248 + y^247 + y^246 + y^245 + y^244 + y^241 + y^235 +
y^234 + y^232 + y^230 + y^229 + y^228 + y^227 + y^226 + y^218 + y^217 + y^208 +
y^207 + y^203 + y^202 + y^199 + y^198 + y^196 + y^195 + y^191 + y^190 + y^189 +
y^187 + y^186 + y^185 + y^179 + y^177 + y^171 + y^168 + y^167 + y^163 + y^160 +
y^158 + y^157 + y^152 + y^150 + y^149 + y^148 + y^147 + y^139 + y^137 + y^135 +
y^133 + y^132 + y^130 + y^129 + y^125 + y^123 + y^122 + y^120 + y^117 + y^116 +
y^115 + y^113 + y^108 + y^106 + y^104 + y^102 + y^100 + y^99 + y^98 + y^97 +
y^95 + y^92 + y^91 + y^89 + y^87 + y^86 + y^85 + y^84 + y^81 + y^80 + y^78 +
y^76 + y^74 + y^73 + y^70 + y^69 + y^68 + y^67 + y^65 + y^64 + y^62 + y^59 +
y^53 + y^50 + y^48 + y^46 + y^42 + y^41 + y^39 + y^36 + y^34 + y^33 + y^29 +
y^28 + y^27 + y^26 + y^24 + y^23 + y^21 + y^20 + y^16 + y^13 + y^12 + y^11 + y^9
+ y^8 + y^6 + y^5
```

```
[0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0,
1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0,
1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0,
1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1,
0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1,
0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1,
0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1,
0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1,
0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1]
```

To encode the message it is multplied by the generator polynomial g. The sent message is then the polynomial $s = m \cdot g$ which is an element in the polynomial ring $\mathbb{F}_2[y]$.

[5]:
```
s = m*g
print(s.list())
```

```
[0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1,
0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1,
1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1,
0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1,
0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0,
1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0,
0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0,
1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1,
0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0,
0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1]
```

The polynomial s is now sent to the recipient across a potentially noisy channel. If the received polynomial r is the same as s then the receiver can recover r by dividing by g:

[6]:
```
r = s
r_quo, r_rem = r.quo_rem(g)
assert(r_rem == 0)
print("No errors; the original message r/g was:")
print(r_quo.list())
```

```
No errors; the original message r/g was:
[0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0,
1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0,
1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0,
1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1,
0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1,
0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1,
0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1,
0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1,
```

```
0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1]
```

However, if the received message r was corrupted then r mod g will not be zero and so r/g does not recover the original message:

[7]:
```
r = s + y^101
r_quo, r_rem = r.quo_rem(g)
print(r_rem.list())
```

```
[1, 1, 0, 0, 1, 0, 0, 1, 1, 1]
```

1.7.3 Recovering the original message

How can the original message be recovered? Assuming there were at most t errors this is possible.

First, note that evaluating r at the points α^i for $1 \leq i \leq 2t$ gives the same exact value as evaluating the "error polynomial" $e = r - s$ at the points α^i for $1 \leq i \leq 2t$. This is a simple consequence of the fact that $\alpha, \ldots, \alpha^{2t}$ are all roots of g, and therefore

$$e(\alpha^i) = r(\alpha^i) - s(\alpha^i) = r(\alpha^i) - m(\alpha^i)g(\alpha^i) = r(\alpha^i) \qquad \text{for } i = 1, \ldots, 2t.$$

Therefore the receiver can compute $e(\alpha^i)$ for $i = 1, \ldots, 2t$. These by themselves do not reveal the error polynomial $e = e_0 + e_1 y + \cdots + e_{n-1} y^{n-1}$, but from the values $e(\alpha), \ldots, e(\alpha^{2t})$ the receiver wishes to recover the coefficients $e_0, \ldots, e_{n-1}$ of the error polynomial e. This is known as an *interpolation* problem. Later in the course we will see that uniquely recovering a degree $n-1$ polynomial requires n evaluations of the polynomial. In this case we only have $2t$ evaluations, but we've also assumed that at most t coefficients of e are nonzero.

The error locator polynomial Let M be the locations at which the errors occured, i.e., $M := \{\, i : e_i \neq 0 \,\}$. The *error locator polynomial* is

$$u(y) := \prod_{i \in M} (1 - \alpha^i y)$$

and clearly u has zeros $y = \alpha^{-i}$ where i is the location of an error. Thus, if we can determine u and find its roots then we can find the error polynomial e (at least when its coefficients are in $\mathbb{F}_2$ and have only two possible values; we merely need to flip the bits at the location of an error). We also define a polynomial v related to u by

$$v(y) := \sum_{i \in M} e_i \alpha^i y \prod_{\substack{j \in M \\ j \neq i}} (1 - \alpha^j y).$$

What a mess! How can this conceivably be useful, given that the receiver doesn't know M (or e_i) and therefore can't construct either u or v?

The reason why u and v are useful is that some incredible simplification occurs when we divide v by u, as shown below.

$$\frac{v}{u} = \sum_{i \in M} \frac{e_i \alpha^i y}{1 - \alpha^i y} = \sum_{i \in M} e_i \sum_{k=1}^{\infty} (\alpha^i y)^k = \sum_{k=1}^{\infty} y^k \sum_{i \in M} e_i (\alpha^k)^i = \sum_{k=1}^{\infty} y^k e(\alpha^k).$$

The second equality uses the formal infinite generating series $\sum_{k=1}^{\infty} x^k = \frac{x}{1-x}$. Of course, we won't actually be using all terms appearing in this representation, since there are an infinite number of them. We'll cut it off after $2t+1$ terms, i.e., use it in the form $\frac{y}{1-y} \equiv \sum_{k=1}^{2t} y^k \pmod{y^{2t+1}}$. In other words, we have

$$\frac{v}{u} \equiv \sum_{k=1}^{2t} e(\alpha^k) y^k \pmod{y^{2t+1}} \qquad (*)$$

and the receiver **does** know the right-hand side, since they can compute $e(\alpha), \ldots, e(\alpha^{2t})$.

Thus the problem becomes to find v/u that satisfies $(*)$ where $\deg(v) \leq t$ and $\deg(u) \leq t$. This is known as a "rational reconstruction" problem. Given the polynomial $\sum_{k=1}^{2t} e(\alpha^k) y^k$ find a rational function that is equivalent to it (modulo a power of y).

Rational reconstruction Let $w(y) := \sum_{k=1}^{2t} e(\alpha^k) y^k$ be the right-hand side of $(*)$. Then rational reconstruction problem is to solve the congruence

$$uw \equiv v \pmod{y^{2t+1}} \qquad \deg(u) \leq t, \deg(v) \leq t$$

for u and v. The congruence can be solved using the Extended Euclidean algorithm on w and y^{2t+1}, as recall that the jth row in the EEA contains the coefficients (t_j, s_j) and the linear combination r_j which satisfy $t_j w + s_j y^{2t+1} = r_j$, so $u := t_j$ and $v := r_j$ satisfies the congruence.

Though all rows of the EEA will produce solutions (u, v) of the congruence, not all rows will produce solutions that satisfy the degree bounds. However, suppose that j is the first row for which $\deg(r_j) \leq t$. One can show that in fact $\deg(t_j) = \deg(r_0) - \deg(r_{j-1}) \leq 2t + 1 - (t+1) = t$ (Lemma 3.10, Modern Computer Algebra). Thus, the EEA can solve the congruence even with the degree bounds.

There is a built-in method in Sage `rational_reconstruct` that performs rational reconstruction on a given polynomial. Given $w, m \in \mathbb{F}_2[y]$, `w.rational_reconstruct(m, dv, du)` solves $wu \equiv v \pmod{m}$ with the degree of $v \in \mathbb{F}_2[y]$ at most `dv` and the degree of $u \in \mathbb{F}_2[y]$ at most `du`.

[8]:
```
w = add([r(alpha^i)*y^i for i in (1..2*t)]) # Construct polynomial w
print(w)
```

```
(a^9 + a^8 + a^6 + a^5 + a^3 + a^2 + a)*y^2 + (a^9 + a^8 + a^7 + a^4 + a + 1)*y
```

[9]:
```
# Perform rational rational reconstruction on w
v, u = w.rational_reconstruct(y^(2*t+1), t, t)
print(u)
```

```
y + a^9 + a^7 + a^3 + a^2 + 1
```

[10]:
```
# Determine the locations of the errors from the error locator polynomial u
# and construct the corrected encoded message c
c = r
for root in u.roots():
    i = (-root[0].log(alpha) % n)
```

```
    print("u has a root alpha^(-{0}), so {0} is the location of an error".
 ↪format(i))
    c = c + y^i

# Determine the original message from the encoded message c
c_quo, c_rem = c.quo_rem(g)
assert(c_rem == 0)
print("")
print("The original message was")
print(c_quo.list())
assert(c_quo == m)
```

```
u has a root alpha^(-101), so 101 is the location of an error

The original message was
[0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0,
1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0,
1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0,
1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1,
0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1,
0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1,
0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1,
0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1,
0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1]
```

1.7.4 An example of correcting multiple errors

The same method works when t is larger than one; the main difference is that the generator polynomial will have higher degree for larger t, since $g(\alpha^i) = 0$ for $i = 1, \ldots, 2t$.

[11]:
```
t = 3
g = lcm([(alpha^i).minpoly('y') for i in (1..2*t)])
print("The generator polynomial g is")
print(g)
s = m*g # Polynomial to send
r = s + y^123 + y^25 + y^201 # Polynomial received with three terms corrupted
w = add([r(alpha^i)*y^i for i in (1..2*t)])
print("The polynomial w to rationally reconstruct is")
print(w)
v, u = w.rational_reconstruct(y^(2*t+1), t, t)
print("The rational reconstruction v/u is")
print(v/u)

# Determine the locations of the errors from the error locator polynomial u
# and construct the corrected encoded message c
```

```
c = r
for root in u.roots():
    i = (-root[0].log(alpha) % n)
    print("u has a root alpha^(-{0}), so {0} is the location of an error".
 ↪format(i))
    c = c + y^i

# Determine the original message from the encoded message c
c_quo, c_rem = c.quo_rem(g)
assert(c_rem == 0)
assert(c_quo == m)
```

```
The generator polynomial g is
y^30 + y^29 + y^28 + y^26 + y^25 + y^24 + y^23 + y^22 + y^20 + y^19 + y^18 +
y^16 + y^14 + y^12 + y^10 + y^9 + y^8 + y^7 + y^6 + y^4 + 1
The polynomial w to rationally reconstruct is
(a^6 + a^4 + a^3 + a + 1)*y^6 + (a^9 + 1)*y^5 + (a^9 + a^7 + a^5)*y^4 + (a^9 +
a^7 + a^6 + a^5 + a^4 + a^3 + 1)*y^3 + (a^7 + a^5 + a^3 + a^2)*y^2 + (a^6 + a^4
+ a^2)*y
The rational reconstruction v/u is
(y^3 + (a^9 + a^7 + a^5 + a + 1)*y)/(y^3 + (a^8 + a^7 + a^6 + a^4 + a^3 + a^2 +
1)*y^2 + (a^9 + a^7 + a^5 + a + 1)*y + a^9 + 1)
u has a root alpha^(-201), so 201 is the location of an error
u has a root alpha^(-25), so 25 is the location of an error
u has a root alpha^(-123), so 123 is the location of an error
```

1.8 Correcting errors over nonbinary fields

If the base field is not $\mathbb{F}_2$ then merely finding the *locations* of the errors is not enough; you also need to be able to recover the original coefficients of the sent polynomial. This is possible by using both the numerator v and the denominator u of the rational reconstruction of w. Interestingly, the *derivative* of the demoninator u is also useful. (Note that it is easy to compute the derivative of a polynomial once you know the coefficients of the polynomial.)

By the product rule from calculus we have

$$u'(\alpha^{-k}) = \frac{\mathrm{d}u}{\mathrm{d}y}\bigg|_{y=\alpha^{-k}} = \sum_{i\in M}(-\alpha^i)\prod_{\substack{j\in M\\ j\neq i}}(1-\alpha^{j-k}).$$

Now suppose $k \in M$. When $k \in M$ the product in this expression becomes zero (except when $k = i$). Thus the summation index expression $i \in M$ reduces to the trivial $i \in \{k\}$ and we have

$$u'(\alpha^{-k}) = -\alpha^k \prod_{\substack{j\in M\\ j\neq k}}(1-\alpha^{j-k}).$$

Now, consider evaluating v at α^{-k}. Similarly, the product in this expression becomes zero except

when $k = i$ and we have

$$v(\alpha^{-k}) = e_k \prod_{\substack{j \in M \\ j \neq k}} (1 - \alpha^{j-k}) = e_k \cdot \frac{u'(\alpha^{-k})}{-\alpha^k}.$$

Putting it all together, the recipient can compute e_k for $k \in M$ via the expression

$$e_k = -\alpha^k \cdot \frac{v(\alpha^{-k})}{u'(\alpha^{-k})}.$$

[12]:
```
# Example using the base field GF(3)

n = 728
F.<a> = GF(3^6)
alpha = F.primitive_element()
assert(alpha^n == 1)
t = 3
g = lcm([(alpha^i).minpoly('y') for i in (1..2*t)])
print("The generator polynomial g is")
print(g)

R.<y> = GF(3)[] # The message will be sent as a polynomial in a new variable y
m = R.random_element(degree=255) # Generate a random message with 256 trits
print("The message m to send is")
print(m.list()) # Message as a list

s = m*g # Polynomial to send
r = s + y^123 + 2*y^25 + y^201 # Polynomial received with three terms corrupted

w = add([r(alpha^i)*y^i for i in (1..2*t)])
print("The polynomial w to rationally reconstruct is")
print(w)
v, u = w.rational_reconstruct(y^(2*t+1), t, t)
print("The rational reconstruction v/u is")
print(v/u)

# Determine the locations of the errors from the error locator polynomial u
# and construct the corrected encoded message c
c = r
for root in u.roots():
    k = (-root[0].log(alpha) % n)
    udiff = u.diff()
    ek = -alpha^k * v(alpha^(-k))/udiff(alpha^(-k))
    c = c - ek*y^k
    print("u has a root alpha^(-{0}), and {0} is the location of an error of␣
 ↪magnitude {1}".format(k, ek))

# Determine the original message from the encoded message c
```

```
c_quo, c_rem = c.quo_rem(g)
assert(c_rem == 0)
assert(c_quo == m)
```

```
The generator polynomial g is
y^24 + y^23 + y^21 + y^17 + y^15 + y^14 + 2*y^13 + y^12 + y^11 + y^10 + 2*y^8 +
y^7 + 2*y^6 + 2*y^3 + y + 1
The message m to send is
[2, 2, 1, 0, 1, 1, 1, 1, 1, 2, 1, 0, 1, 1, 1, 0, 2, 0, 0, 2, 2, 2, 2, 2, 0, 2,
2, 1, 0, 0, 0, 1, 0, 2, 2, 2, 1, 0, 0, 0, 0, 2, 2, 2, 1, 2, 0, 2, 2, 2, 1, 1, 1,
0, 0, 0, 2, 0, 1, 0, 1, 2, 2, 1, 1, 1, 1, 0, 2, 2, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0,
0, 1, 1, 0, 0, 1, 0, 2, 2, 1, 1, 0, 2, 0, 0, 1, 2, 0, 0, 2, 2, 1, 1, 0, 0, 2, 2,
2, 0, 0, 1, 2, 1, 1, 1, 0, 2, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1,
0, 1, 1, 0, 0, 2, 1, 2, 1, 2, 1, 2, 0, 2, 1, 1, 0, 0, 2, 1, 2, 1, 2, 2, 1, 0, 1,
2, 0, 0, 1, 0, 2, 2, 2, 2, 1, 0, 0, 0, 2, 0, 2, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 2,
0, 1, 0, 2, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 2, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1,
0, 0, 2, 0, 0, 2, 2, 0, 2, 2, 1, 0, 2, 0, 2, 0, 0, 1, 1, 2, 1, 0, 0, 1, 0, 0, 0,
0, 2, 1, 2, 2, 1, 1, 2, 1, 1, 2, 1, 0, 2]
The polynomial w to rationally reconstruct is
(2*a^5 + a^4 + a^3 + 2*a^2 + 2)*y^6 + (2*a^4 + 2*a^3 + a^2 + 1)*y^5 + (a^5 +
2*a^4 + 2*a^3 + 1)*y^4 + (2*a^5 + 2*a^4 + a^3 + 2*a^2)*y^3 + (a^5 + 2*a^3 +
2*a^2 + 2)*y^2 + (2*a^5 + 2*a^4 + a^3 + 2*a^2 + a + 2)*y
The rational reconstruction v/u is
(2*y^3 + (a^5 + 2*a^4 + 2*a^2 + 2)*y^2 + (a^5 + a^3 + a + 2)*y)/(y^3 + (a^5 +
a)*y^2 + (a^3 + 2*a^2 + 2*a + 2)*y + 2*a^5 + a^3 + 2*a^2 + 2*a + 1)
u has a root alpha^(-201), and 201 is the location of an error of magnitude 1
u has a root alpha^(-123), and 123 is the location of an error of magnitude 1
u has a root alpha^(-25), and 25 is the location of an error of magnitude 2
```

1.9 Takeaway

Reed–Solomon codes are widely used in practice. For example, data on DVDs are split up into 192×172 matrices over $\mathbb{F}_{256}$ and then both the rows and columns are encoded using a Reed–Solomon code.

The rows of the matrices are specified using polynomials in $\mathbb{F}_{256}[y]$ of degree less than 172 and then encoded as a polynomial of degree up to 182 by multiplying the row polynomial by $g_1(y) := \prod_{i=0}^{10}(y - \alpha^i) \in \mathbb{F}_{256}[y]$. The decoding scheme presented above can then correct up to 5 errors.

For additional redundency, the columns of the matrices are specified using polynomials in $\mathbb{F}_{256}[y]$ of degree less than 192 and then encoded as a polynomial of degree up to 208 by multiplying the column polynomial by $g_2(y) := \prod_{i=0}^{16}(y - \alpha^i) \in \mathbb{F}_{256}[y]$ so that up to 8 errors can be corrected.

This is the first usage of non-prime finite fields in this course. Although we may not use them again in this course, they arise frequently in mathematics and computer science and are used in many more applications that stretch beyond coding theory.