# Computation in finite fields

John Kerl
Arizona State University and Avnet Design Services

March 26, 2007

## Abstract

Elementary techniques for computation in finite fields are presented, at a level appropriate for an engineering, computer science or mathematical audience.

Second, elementary Galois theory for finite fields is presented at a level appropriate for undergraduate or beginning graduate students in mathematics. For the latter, illustrative properties of a novel *logarithmic root chart* are developed.

Third, linear feedback shift registers are presented in a manner which connects mathematical theory and engineering practice.

Fourth, some techniques for software implementation are outlined: using a computer algebra system for the general case, and in C for the $p = 2$ case.

# Contents

# Chapter 1

# Introduction

## 1.1 Purpose

Finite fields are often given a brief treatment in algebra courses, and when they are discussed, they are discussed in the abstract. This is in fact a testament to the power of theory: Many properties may be deduced about finite fields using no other information than that they are fields and that they are finite. However, specific computations are interesting as well as useful. This paper will present elementary techniques that are readily implementable with pencil and paper, software or electronic circuitry.

## 1.2 Context

Why do we care about finite fields in the first place? Among their applications are the following:

- They are beautiful and interesting in their own right, which is sufficient justification for a pure mathematician to care about them.
- They have applications in number theory, i.e. pure math as applied to other parts of pure math.
- As we will see, finite fields have size $p^n$ for some prime integer $p$. Finite fields with large $p$ and $n = 1$ are crucial in many cryptosystems, and of course cryptography is important in today's electronic commerce as well as other electronic activities requiring privacy. For another example, the AES cryptosystem [1] is built entirely out of a finite field with $p = 2$ and $n = 8$ (hence its natural operation on bytes): even the S-box is just reciprocation in the finite field.

- Finite fields with $p = 2$ are essential in coding theory [3], [5], which is the study of the efficient detection and correction of errors in a signal. This is the kind of thing you use (without even realizing it) whenever you talk on a cell phone or use a data network.

## 1.3 Scope

Whereas finite fields may be presented in the abstract, this paper will take the opposite approach by first constructing systems which are fields and which are finite, and then illustrating how the resulting fields have properties which the theory told us they would have. As a result, few proofs will be given; the emphasis is on computation.

Not only will I eschew proofs in favor of computation, but moreover I will defer some computational methods to the appendices, to keep everything simple and smoothly flowing in the main discourse.

Since this paper cites many well-known theorems without proof, it is not by any means a standalone treatise on finite fields. For the intentionally omitted theory, please consult any graduate-level text on abstract algebra. In particular I recommend Dummit and Foote [4]. Also, Lidl and Niederreiter [10] is the standard (and excellent) reference for finite fields. (Note: Any theorem or algorithm in this paper which I've not attributed to a specific reference either is my own or, more often, is something I consider to be general knowledge.) For the software-implementation section, I assume only basic familiarity with C.

## 1.4 Audience

My intended audience is non-specialists and fellow students. For the first few sections, which discuss how to add, subtract, multiply and divide, I will presume a general familiarity with abstract algebra, e.g. 444 at Arizona State. For the later sections on Galois theory, I will of course presume some knowledge of Galois theory. This entails a jump in the level of prerequisite material, so these latter sections may be omitted. (However, my treatment of Galois theory is gentle; a reader without previoius exposure with Galois theory should be able to read these sections.)

My particular concern is for (a) engineers who wish to have a better understanding of what's going on, and (b) mathematicians who want be able to perform computations, e.g. for testing out hypotheses.

## 1.5 Rings and fields; notation

I will assume familiarity with ring and field axioms. Fields discussed in this document are almost always the finite ones, with occasional mention of the rationals $\mathbb{Q}$, reals $\mathbb{R}$ and complexes $\mathbb{C}$. The only rings discussed are the integers $\mathbb{Z}$, as well as the ring of polynomials with coefficients in a field. For a field $F$, the latter is written $F[x]$. The ideal generated by a ring element $m$ will be written $\langle m \rangle$.

Recall that a ring of polynomials whose coefficients are in a field is a Euclidean domain, hence a principal ideal domain, hence an integral domain. (Remember *FEPUI* [feh, pooey!]: fields $\subsetneq$ Euclidean domains $\subsetneq$ PIDs $\subsetneq$ UFDs $\subsetneq$ integral domains.) This means that we can factor our integers or polynomials, that an ideal is simply the set of all multiples of some particular integer or polynomial, and that we can divide one integer or polynomial by another non-zero integer or polynomial to get a quotient and remainder. Furthermore, the remainder is smaller than the divisor in the usual way in the integers; for polynomials, the remainder has smaller degree, or is the zero polynomial.

# Part I

# Arithmetic

# Chapter 2

# The field $\mathbb{F}_p$

One of the many finite-field theorems which this paper will reference (without proof) is that all finite fields of a given size are isomorphic to one another. This means that I may construct certain finite fields with the confidence that any others will look something like them. As the canonical representative of a first family of finite fields, I will select the integers modulo a prime integer $p$. These are named $\mathbb{F}_p$, where the font is intended to indicate that they are of equal importance with $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, etc. (An older notation still occasionally used in the literature is $\mathrm{GF}(p)$, which stands for **Galois field**.)

Not all fields are accounted for by the $\mathbb{F}_p$'s, but later we will use the $\mathbb{F}_p$'s to construct all the remaining finite fields.

## 2.1 Elements of $\mathbb{F}_p$

Formally, elements of $\mathbb{F}_p$ are equivalence classes of integers modulo $p$, where we say that two integers are equivalent mod $p$, written $a \equiv b \,(\mathrm{mod}\, p)$ or simply $a \equiv b(p)$, iff $a - b \,|\, p$. For example, if $p = 5$, we have a set with five elements:

$$\begin{aligned}\mathbb{F}_5 = \mathbb{Z}/5\mathbb{Z} \quad = \quad & \{\{\ldots, -10, -5, 0, 5, 10, \ldots\} \\ & \{\ldots, -9, -4, 1, 6, 11, \ldots\} \\ & \{\ldots, -8, -3, 2, 7, 12, \ldots\} \\ & \{\ldots, -7, -2, 3, 8, 13, \ldots\} \\ & \{\ldots, -6, -1, 4, 9, 14, \ldots\}\}\end{aligned}$$

Since the integers $\mathbb{Z}$ are a Euclidean domain, we may divide any integer $a$ by $p$ to obtain a quotient $q$ (which we don't care about) and a remainder $r$ such that $0 \leq r < p$. Clearly, any two elements of the same equivalence class have the same remainder $r$, so we take the $r$'s to be the canonical representatives of the equivalence classes. (It is easy to show that reducing mod $p$ is a ring homomorphism from $\mathbb{Z}$ to $\mathbb{Z}/p\mathbb{Z}$.) Then, the above five-element set may be more compactly written as

$$\mathbb{F}_5 = \{\overline{0}, \overline{1}, \overline{2}, \overline{3}, \overline{4}\}$$

where the overline refers to an equivalence class.

Then, the following two statements are equivalent:

$$\begin{aligned} 7 &\equiv 2 \pmod 5 \\ \overline{7} &= \overline{2} \end{aligned}$$

where the former statement involves integers and the latter involves equivalence classes. However, I will sloppily omit the overlines and write

$$\mathbb{F}_5 = \{0, 1, 2, 3, 4\}$$

where it is now implicit that we are working mod 5. From a formal point of view, this is sloppy; however, it is tolerable and helps avoid a forest of overline bric-a-brac.

## 2.2 Arithmetic in $\mathbb{F}_p$

Addition, subtraction and multiplication are as you would expect. Examples, again with $p = 5$:

- $3 + 4 = 7$, which is 2 mod 5.
- $3 - 4 = -1$, but remember we want our canonical representatives to be between 0 and $p - 1$, inclusive, so we say $3 - 4 = 4$.
- $3 \cdot 4 = 12$, which is 2 mod 5.

Simple enough. In fact, addition, subtraction and multiplication work like this in $\mathbb{Z}/m\mathbb{Z}$ for any integer $m$, prime or not. As we'll see below, though, to have a field (i.e. to have reciprocals for all non-zero elements) we need the modulus to be prime.

How do we divide? Clearly, it suffices to know how to reciprocate, e.g. if we can compute $1/b$, then we can use multiplication (which we already know how to do) to compute $a/b = a \cdot (1/b)$. But how do we reciprocate? For example, what is $1/2$ mod 5? There are four methods.

## 2.3 First reciprocation method in $\mathbb{F}_p$

One answer is to search for a reciprocal. Since finite fields are finite, it is possible to enumerate all the elements of the field and see which one works: 2 times 0 is 0, nope; 2 times 1 is 2, nope; 2 times 2 is 4, nope; 2 times 3 is 1, aha, so $1/2 = 3$. The search method for reciprocation is acceptable for small $p$, but if, say, you wanted to compute 1/7 mod 1009 this way, you'd probably get tired of looking before reaching 865.

## 2.4 Second reciprocation method in $\mathbb{F}_p$

You might wonder, how do you know there will *be* an inverse — which is tantamount to asking, how do we know $\mathbb{F}_p$ is really a field? (All the other field axioms besides the existence of multiplicative inverses, i.e. reciprocals, are easily verified. That is, the integers mod $m$ form a commutative ring with identity regardless of whether $m$ is prime.) The proof of the existence of reciprocals mod $p$ is a constructive proof, and so gives our second method for reciprocation.

I said above that the homomorphism from $\mathbb{Z}$ to $\mathbb{F}_p$ is the **mod** operation. The opposite operation is called **lifting**. This is not an inverse function, of course, since the mod operation is not one-to-one. Nonetheless, given an element of $\mathbb{F}_p$, sometimes we want to send it to some element of $\mathbb{Z}$ — usually to the canonical representative which is between 0 and $p - 1$.

Given an element $a$ of $\mathbb{F}_p$, erase the imaginary overline which I am not writing, i.e. lift $a$ to the integers. If $a$ is 0 (or is a multiple of $p$, which means $a$ is 0 mod $p$), then we expect no reciprocal. But if $a$ is not a multiple of $p$, then it is relatively prime to $p$. This means its GCD with $p$ is 1. Using the **extended GCD**, also known as the *GCD trick*, we can always write the output of the GCD algorithm as a linear combination of the two inputs, i.e.

$$1 = as + pt$$

for some integers $s$ and $t$. But $pt$ is a multiple of $p$ and so is zero mod $p$, so

$$1 \equiv as \pmod{p}$$

which means the equivalence class containing $s$ is the reciprocal of the equivalence class containing $a$. Above we lifted to $\mathbb{Z}$ to do the GCD; now we mod back to $\mathbb{F}_p$ by restoring the imaginary overline, and say that $s$ is the reciprocal of $a$ in $\mathbb{F}_p$.

Now, this is great for proving existence of reciprocals in $\mathbb{F}_p$, but the details of pencil-and-papering the extended GCD algorithm (details which I've omitted) are tedious. (Even using Blankinship's algorithm for extended GCD, appendix A on page 84, it's better but still tedious). There are two more pleasant ways, as we will see.

## 2.5 Third reciprocation method in $\mathbb{F}_p$

The third method uses a little group theory. Namely, recall that in any field $F$, the non-zero elements form a group under multiplication — this is called the field's **multiplicative group**, and is written $F^\times$, or occasionally $F^*$. If $\mathbb{F}_p$ has $p$ elements, then there are $p-1$ non-zero elements in $\mathbb{F}_p^\times$.

Recall Lagrange's theorem, which is your first real theorem in group theory: the order of a subgroup divides the order of the parent group. Also, given any element $a$ of a finite group, you can form the cyclic subgroup generated by the element, simply by taking powers of $a$ until you get the identity $e$. (You *will* get the identity eventually: you will eventually run out of elements and get a loop wherein $a^i = a^j$ for some $i$ and $j$, and then you'll have $a^{i-j} = e$.) Also recall that the order of an element, that is, the least positive integer $k$ such that $a^k = e$, is the same as the order of the cyclic subgroup generated by $a$.

So if the order of $a$ is $k$, and since by Lagrange's theorem $k \bigm| |G|$, we have $a^{|G|} = e$ for all $a$ in $G$. Now, a smaller power might send a particular $a$ to $e$, and in fact for non-cyclic groups a smaller power might suffice to send *all* elements to $e$ (e.g. the Klein four-group has order 4 but all elements have order 1 or 2). But nevertheless we know that $a^{|G|}$ will always be $e$, for any element $a$ of any finite group $G$.

For the case $G = \mathbb{F}_p^\times$, where $|G| = p-1$ and the identity is 1, we have

$$a^{p-1} = 1$$

for all non-zero elements of $\mathbb{F}_p$. (Hence a quick proof of Fermat's little theorem.) This has two direct consequences. First, we can write

$$a^{p-2} = a^{-1}$$

for all non-zero elements, which gives us our third algorithm for reciprocation, the $p-2$ algorithm. Second, we can make this true for all elements of $\mathbb{F}_p$ (including zero) by writing

$$a^p = a$$

or

$$x^p - x = 0$$

which is a polynomial satisfied by all elements of $\mathbb{F}_p$. This sounds like it might be a useful piece of information to remember.

So, given the $p-2$ algorithm, we can easily reciprocate in $\mathbb{F}_p$. What is $1/2$ mod 5? Since $5-2=3$, we can invert any element simply by cubing it. E.g. $2^3 = 8$ which is 3. So $1/2 = 3$ mod 5. And we can verify this by writing $2 \cdot 3 = 1$ which checks out.

## 2.6 Repeated-squaring algorithm for exponentiation

Raising an element $a$ to the $p-2$ power by performing $p-3$ multiplications is OK for small $p$, but not so good for larger $p$. (E.g. to find 1/7 mod 1009, 1006 multiplications by 7 is no fun at all.) However, repeated squaring makes it a breeze. To see this by example, let $p = 23$. Then we can reciprocate any element $a$ of $\mathbb{F}_{23}$ by raising $a$ to the 21st power. Write out 21 as a sum of powers of 2, e.g. $21 = 16 + 4 + 1$. Then $a^{21} = a^{16}a^4a^1$. Then given $a$ we can square $a$ to get $a^2$, square that to get $a^4$, square that to get $a^8$, and square one more time to get $a^{16}$. Then just multiply the 1st, 4th and 16th powers and discard the 2nd and 8th powers.

Instead of requiring $p-3$ multiplications, this requires only approximately $\log_2(p-2)$ multiplications. So you can raise something to the millionth power with only 20 multiplications. On paper this isn't too bad, and in a computer it takes only an instant.

## 2.7 Fourth reciprocation method in $\mathbb{F}_p$; logarithms

One of the finite-field theorems which this paper doesn't prove is that the multiplicative group of a finite field is *cyclic*. This goes back at least to Gauss, who proved that there are always **primitive elements** mod $p$, i.e. always at least one element of $\mathbb{F}_p^\times$ such that all other elements are powers of it.

In group-theoretic terms, this means there is an element $g$ ($g$ for *generator*) such that for all $a \in \mathbb{F}_p^\times$ there is a $k$, with $0 \le k < p-1$, such that $a = g^k$. Since $a^{p-1} = 1$, these exponents are actually taken mod $p-1$. If $a = g^k$, then $a^{-1} = g^{-k}$, or equivalently $a^{-1} = g^{p-1-k}$.

This is an even quicker way to reciprocate in $\mathbb{F}_p$, but it comes at a little expense. The $p-2$ method requires you only to know what $p$ is; this method requires a little precomputation in order to also find a generator $g$. How do you find a generator (i.e. primitive element) mod $p$? Some slightly more sophisticated techniques are discussed in appendix C on page 88, but a simple and quite sufficient method is just to search for one. It turns out that either 2 or 3 works most of the time, and if not, then try 5, etc. You will find one.

For example, with $p = 5$, powers of 2 are

| $k$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $2^k$ | 2 | 4 | 3 | 1 |

which runs through all of $\mathbb{F}_5^\times$, so 2 is primitive mod 5.

With $p = 7$, powers of 2 are

| $k$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $2^k$ | 2 | 4 | 1 | 2 | 4 | 1 |

which doesn't run through all of $\mathbb{F}_7^\times$. But, powers of 3 are

| $k$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $3^k$ | 3 | 2 | 6 | 4 | 5 | 1 |

so 3 is primitive mod 7.

You can similarly show (just by trying it) that 2 is primitive mod 11. I'll use 11 as an example of log reciprocation, since 11 is big enough to be a little more interesting.

We saw above that since $\mathbb{F}_p^\times$ is cyclic, we can write any non-zero $a$ as

$$a = g^k.$$

Then we define the logarithm base $g$ of $a$ to be

$$\log_g(a) = k$$

(This is sometimes called a **discrete log** since it takes only integer values.)

Now I can tabulate field elements and their logs, sorting by field element, to obtain a **log table** (Lidl and Niederreiter call this an *ind table*, for *index*) as follows:

| $\log_2(a) = k$ | 0 | 1 | 8 | 2 | 4 | 9 | 7 | 3 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|
| $a = 2^k$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Also, I can sort the log table by log value to obtain an antilog table:

| $\log_2(a) = k$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $a = 2^k$ | 1 | 2 | 4 | 8 | 5 | 10 | 9 | 7 | 3 | 6 |

Since $a^{-1} = g^{-k} = g^{p-1-k}$, it's easy to reciprocate just by looking at the antilog table. E.g. $8 = 2^3$ (three places from the left), so $1/8 = 2^{10-3} = 2^7 = 7$ (three places from the right).

You can multiply using the log and antilog tables by the familiar rule $\log_g(ab) = \log_g(a) + \log_g(b)$, where you must remember to do the addition mod $p - 1$:

$$ab = g^{\log_g(ab)} = g^{\log_g(a)+\log_g(b)}$$

Likewise you can do division in a single step (i.e. instead of finding the reciprocal, then multiplying by the reciprocal) using the familiar $\log_g(a/b) = \log_g(a) - \log_g(b)$:

$$a/b = g^{\log_g(a/b)} = g^{\log_g(a)-\log_g(b)}$$

See also appendix B on page 86 for an efficient way to build large log tables (e.g. in a software implementation).

# Chapter 3

# The field $\mathbb{F}_{p^n}$

That's about it for $\mathbb{F}_p$: we know how to add, subtract, multiply and divide, and we have the polynomial $x^p - x = 0$ of which all elements of $\mathbb{F}_p$ are roots. Now we can look at all the other finite fields.

I went into some detail with $\mathbb{F}_p$, which may have seemed overkill — after all, $\mathbb{F}_p$ is just integers mod $p$, which are familiar, and who really needs four different techniques for reciprocation when any one of them would suffice? While $\mathbb{F}_{p^n}$ may be a little less familiar, much of what we for $\mathbb{F}_{p^n}$ will be in direct analogy to things in $\mathbb{F}_p$, and those analogies will be instructive. We'll see another reason in the next section.

## 3.1 Field characteristic and prime subfield

Consider the order of 1 in the additive group of a finite field $F$. This order is called the **characteristic** of the field. It cannot be infinite (we would run out of elements) so let $M$ be the order of 1. Then 1 added to itself $M$ times is 0, i.e. $M \cdot 1 = 0$. If $M$ is composite, with non-trivial factors $a$ and $b$, then we get $(a \cdot 1)(b \cdot 1) = 0$ which gives zero divisors which are not present in a field. So, $M$ must be prime and without further ado I rename $M$ to be $p$.

Furthermore, it's easy to show that the set of distinct repeated sums of 1 in $F$ (i.e. 1, $1 + 1$, etc.) forms a subfield which is isomorphic to $\mathbb{F}_p$.

## 3.2 Existence of $\mathbb{F}_{p^n}$

Whenever we have a smaller field $K$ (finite or not) and a bigger field $F$ with $K$ a subset of $F$, we say that $K$ is a **subfield** of $F$, and that $F$ is an **extension field** of $K$. If you write down the axioms for a vector space $V$ over a field $K$ (i.e. scalars are elements of $K$), then replace the $V$'s with $F$'s (i.e. vectors are elements of $F$), you can simply check off all the axioms and see that they're all satisfied. This means that $F$ is necessarily a *vector space* over $K$, and in particular, any finite field is a vector space over $\mathbb{F}_p$ for some $p$. Since the vector space is finite, the dimension of that vector space is finite; I'll call it $n$. This forces the size of any finite field $F$ to be $p^n$ for some prime $p$ and some positive integer $n$. So if we construct a finite field, we know it will have prime-power order.

Another finite-field theorem tells us that all finite fields of a given size are isomorphic to one another. This means that if, given $p$ and $n$, I can construct even a single finite field of order $p^n$, I'll in some sense have constructed them all. It remains only to actually construct such a field.

To do that, we start by looking at the ring of polynomials with coefficients in the field $\mathbb{F}_p$, which we call $\mathbb{F}_p[x]$. While each coefficient has only $p$ possible values, the degree can be arbitrarily large — in particular, even though the coefficients are taken mod $p$, the exponents are not, so you can have $x^{100}$, $x^{1000}$, etc.

Also notice that there are $p-1$ possible degree-0 polynomials (excluding 0 itself), $(p-1)p$ of degree 1 (leading coefficient must be non-zero for the polynomial to be linear and the rest can be anything mod $p$), $(p-1)p^2$ of degree 2, etc. and in general $(p-1)p^n$ of degree $n$. So although $\mathbb{F}_p[x]$ has infinitely many polynomials in it, there are only *finitely many* polynomials of a given degree $n$, and likewise only finitely many polynomials of degree less than $n$.

Second, since $\mathbb{F}_p$ is a field, $\mathbb{F}_p[x]$ is a Euclidean domain (hence a PID and an integral domain) so it makes sense to talk about *irreducible polynomials* in $\mathbb{F}_p[x]$, and for simplicity we will take our irreducibles to be **monic**, i.e. with 1 for the leading coefficient[1].

Given a monic irreducible $r(x)$ (for this to be sensible we require $\deg(r) \geq 1$), we can form the ideal $\langle r(x) \rangle$ generated by it. Since $\mathbb{F}_p[x]$ is a commutative ring with identity, this ideal is nothing more than the set of all multiples of $r(x)$, that is, all those polynomials in $\mathbb{F}_p[x]$ which, when factored into irreducibles, have $r(x)$ as one of their factors. This is in exact analogy to $\langle p \rangle$ being an ideal of $\mathbb{Z}$: the ideal $\langle p \rangle$ is the set of all multiples of $p$ in $\mathbb{Z}$; the ideal $\langle r(x) \rangle$ is the set of all multiples of $r(x)$ in $\mathbb{F}_p[x]$. Also, since $r(x)$ is irreducible, the ideal $\langle r(x) \rangle$ is maximal in $\mathbb{F}_p[x]$.

Now[2] we simply take the quotient ring $\mathbb{F}_p[x]/\langle r(x) \rangle$. Since $\mathbb{F}_p[x]$ is commutative and $\langle r(x) \rangle$

[1]There are technical distinctions between *prime* and *irreducible*, which disappear for PIDs. I will nonetheless choose to reserve the term *prime* for integers $p$ which don't factor any more than into $\pm p$ and $\pm 1$, and I will reserve the term *irreducible* for polynomials which don't factor into the product of polynomials of lesser degree. Over $\mathbb{F}_p$, or any field for that matter, we can always play with the leading coefficient and pull out a scalar factor — but I won't consider this to be an interesting factorization. That is, over a field, polynomial factorization is interesting only up to scalars, which is why we choose monics as our standard irreducibles.

[2]In the general theory, one allows $\mathbb{F}_q$, with $q = p^m$, to be the base field and studies $\mathbb{F}_{q^n}$ as an extension.

is maximal, $\mathbb{F}_p[x]/\langle r(x)\rangle$ is a field — this is what the theory guarantees for us. Let $n$ be the degree of $r(x)$. Then we write

$$\mathbb{F}_{p^n} = \mathbb{F}_p[x]/\langle r(x)\rangle$$

(When we formed $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$, we saw that we got a ring whether or not $p$ was prime, but needed $p$ prime to get a field. Here too, $\mathbb{F}_p[x]/\langle r(x)\rangle$ is a ring regardless of whether $r(x)$ is irreducible, but we'll need it to be irreducible so that we can reciprocate all non-zero elements.)

You might ask if there is a monic irreducible for all $p$ and $n$, and you might also ask what happens if there is more than one — how can $\mathbb{F}_{p^n}$ be uniquely defined? The short answer is that the finite-field theorems guarantee there is in fact always at least one monic irreducible (see section 5.8) for all $p$ and $n$, and if there is more than one (which generally there is) then all such fields are isomorphic to one another. Later on in this paper we'll look at those isomorphisms in more detail.

So what does one of these $\mathbb{F}_{p^n}$'s look like? How do we compute in it? Where's the beef?

When we constructed $\mathbb{F}_p$, we took the infinite set $\mathbb{Z}$, then divided by $p$, discarded the quotient and kept the remainder to get a finite set of possible remainders 0 through $p-1$. We defined elements of $\mathbb{F}_p$ to be the set of equivalence classes where two integers were equivalent if they had the same remainder.

Here we can do the same thing: $\mathbb{F}_p[x]$ is a Euclidean domain so we can divide any polynomial by $r(x)$ to get a remainder. Now, $\mathbb{F}_p[x]$ is not totally ordered, but it is partially ordered by degree. When we divide by $r(x)$, we get a quotient which we again discard, and we get a remainder which is zero, or whose degree is less than $r(x)$'s degree. To see how that works, I'll do a few more computational examples.

## 3.3 Computation in $\mathbb{F}_p[x]$

Since $\mathbb{F}_p[x]$ is a ring, we can add, subtract and multiply. Since it's a Euclidean domain, we can divide and get a quotient and a remainder. It's important to remember this distinction: e.g. in $\mathbb{Z}$, 9/4 has quotient 2 and remainder 1, whereas in $\mathbb{Q}$ it's exactly 9/4 with never any remainder. Here we're doing ring division, with quotient and remainder [3].

All these operations are done in precisely the same way you learned to do arithmetic in

However, $\mathbb{F}_q$ is precisely what we're still learning to construct, so I'll defer use of this idiom.

[3] $\mathbb{F}_p[x]$ does have a field of fractions, as $\mathbb{Z}$ has $\mathbb{Q}$, which we call $\mathbb{F}_p(x)$: the field of rational functions in $x$ with coefficients in $\mathbb{F}_p$. This is an infinite field of characteristic $p$, which makes it a little interesting. However, we won't deal at all with the field $\mathbb{F}_p(x)$ in this paper. To dispel another frequent misconception while we're at it: $\mathbb{F}_{p^n}$ is not $\mathbb{Z}/p^n\mathbb{Z}$, except for $n = 1$: in particular $\mathbb{Z}/p^n\mathbb{Z}$ has zero divisor $p$ whenever $n > 1$ so it cannot be a field.

elementary school — with one small exception.

First, for brevity of notation I'll often strip off the $x$'s, exponents and plus signs from elements of $\mathbb{F}_p[x]$. E.g. $x^3 + x^2 + 1$ will be simply 1101. (If $p$ were greater than 10, which it won't be from here on out, I would write $1, 1, 0, 1$ to avoid ambiguity.) This **compact notation** will be familiar from synthetic division in high school. Then $x^3 + x^2 + 1$ is an example of what I will refer to as **full notation**.

Now, addition is just like integer addition in elementary school (and similar to synthetic addition of polynomials in high school) with the one small exception that there are no carries. E.g. with $p = 5$,

```
  1044
+  231
------
  1220
```

Likewise,

```
  1044
-  231
------
  1313
```

We simply operate coefficientwise, using $\mathbb{F}_p$ arithmetic which we already know how to do. Remember that since the coefficients are in $\mathbb{F}_p$, we always want coefficients between 0 and $p - 1$, so if a coefficient goes negative on subtraction, add $p$ to the difference.

Multiplication is more of the same: form partial products by multiplying scalars times polynomials, then add the partial products. Again with $p = 5$:

```
  1044
*  231
------
  1044
 3022
2033
------
234014
```

The last thing to do is division. (I emphasize that this is still quotient-and-remainder division in the ring $\mathbb{F}_p[x]$.) And it's just what you'd think. Remember that you don't do carries, but also remember that there's no concept of "less than" for coefficients. E.g. the integer 231 doesn't go into the integer 114 because 231 is bigger than 114. But here, no problem. To get each quotient digit, divide the leading coefficient of your running remainder by the leading coefficient of the divisor (which is necessarily non-zero since you're not dividing by zero), and we saw in chapter 2 that you can always do that in $\mathbb{F}_p$. But do stop when the *degree* of the running remainder is less than the degree of the divisor. For example:

```
         33 = quotient
    +--------
231 |  1044
    |- 143    1st quotient digit = 3 since 1/2 = 3 mod 5; 231 * 3 = 143
    +--------
    |   114
    |-  143   2nd quotient digit = 3 since 1/2 = 3 mod 5; 231 * 3 = 143
    +--------
    |    21 = remainder
```

so here the quotient is 33 and the remainder is 21. Using the full notation, this means $x^3 + 4x + 4$ is $2x + 1$ mod $2x^2 + 3x + 1$. (Note that 231 is not irreducible, and doesn't need to be — this example merely shows ring division for some arbitrary ring elements.)

Next we need to know how to find irreducibles $r(x)$, and then we'll have all the machinery we need to take remainders when constructing $\mathbb{F}_p[x]/\langle r(x)\rangle$.

## 3.4 Simple determination of irreducibles in $\mathbb{F}_p[x]$

In appendix D on page 89 I give some techniques for irreducibility testing. Here I'll just give some simple methods.

If a polynomial has degree 3 or less and factors into smaller-degree terms, then at least one of the factors must be linear. We know that linear factors correspond to roots: If $a$ is a root of $f(x)$, then $f(x)$ has $x - a$ as a factor and vice versa. And it's easy to check for roots in $\mathbb{F}_p$, since $\mathbb{F}_p$ is finite: Just try all the elements.

This gives one method for irreducibility testing for $n \le 3$: Evaluate $r(0)$, $r(1)$, ..., $r(p-1)$ and if those are all non-zero, $r(x)$ is irreducible. E.g. with $p = 2$ and $r(x) = x^3 + x + 1$, $r(0) = 1$ and $r(1) = 1$ so $x^3 + x + 1$ is irreducible.

Even for polynomials of degree higher than 3, the root test can be used to quickly determine that a polynomial is *not* irreducible. When $p = 2$, this is particularly easy: $r(0)$ is just the constant term, so if the constant term is 0, $r(x)$ is reducible (except for $r(x) = x$). Also, $r(1)$ is just the sum of coefficients mod 2, so if there is an even number of non-zero coefficients,

$r(x)$ is reducible (except for $r(x) = x + 1$). E.g. with $p = 2$, $x^{37} + x^{28} + x^9 + x^7 + x$ is reducible, as is $x^{37} + x^{28} + x^9 + 1$.

The second simple method for irreducibility testing (after you've already checked for linear factors via the root test) is trial division. Since there are finitely many polynomials in $\mathbb{F}_p[x]$ with degree less than the degree of $r(x)$, just try them all. To be a little more sophisticated, avoid the trial factors you know are reducible. (An analogy with the integers: You don't have to test 6 as a factor of some number $n$, if you've already tried, or will try, 2 and 3.) Also, try factors with degree only up to $n/2$, since if there were factors of higher degree, you'd have already found the corresponding factor of lower degree. (Another analogy with the integers: when factoring an integer $n$ by trial division, try only up to $\sqrt{n}$.)

Let's use the above to enumerate all irreducible polynomials of degree up to 4 with $p = 2$.

For $n = 1$, we have $x$ and $x + 1$ (10 and 11 using the compact notation), which are already linear and therefore irreducible.

For $n = 2$, we have (using the compact notation) 100, 101, 110 and 111. Two of these have a zero constant term and so have 0 as a root; 101 has an even number of 1's and so has 1 as a root. That leaves 111 ($x^2 + x + 1$) which is quadratic and rootless, therefore irreducible.

For $n = 3$, there are the eight possibilities

$$1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111$$

All those ending in 0 have 0 as a root; 1001, 1010, 1100 and 1111 have 1 as a root. That leaves 1011 and 1101 ($x^3 + x + 1$ and $x^3 + x^2 + 1$) which are irreducible since quadratic and rootless.

For $n = 4$, there are the sixteen possibilities

$$\begin{gathered}10000, 10001, 10010, 10011, 10100, 10101, 10110, 10111,\\ 11000, 11001, 11010, 11011, 11100, 11101, 11110, 11111\end{gathered}$$

The root test pares the list down to

$$10011, 10101, 11001, 11111$$

Since the root test excludes all linear factors, only the quadratic factors remain and we may restrict ourselves to the irreducible quadratics. There is only one, namely, 111. Synthetic division as above shows that $10101 = 111^2$; the other three have non-zero remainder and therefore must be irreducible.

To summarize:

Monic irreducibles with $p = 2$, $n = 1$ through 4

| $n$ | Compact notation | Full notation |
|---|---|---|
| 1 | 10 | $x$ |
|  | 11 | $x+1$ |
| 2 | 111 | $x^2+x+1$ |
| 3 | 1011 | $x^3+x+1$ |
|  | 1101 | $x^3+x^2+1$ |
| 4 | 10011 | $x^4+x+1$ |
|  | 11001 | $x^4+x^3+1$ |
|  | 11111 | $x^4+x^3+x^2+x+1$ |

These will be used frequently in the examples to follow.

A third and time-honored method for irreducibility testing is simply to consult a table. See Lidl and Niederreiter for lots of tables, or my very abbreviated version in appendix H.1 on page 97.

## 3.5 Elements of $\mathbb{F}_{p^n}$

So now we know how to find monic irreducibles $r(x)$ for any $p$ and $n$. (The theory guarantees us we will find at least one.) Also, we know how to do ring division by $r(x)$, to produce remainders. Note that if $r(x)$ has degree $n$, then the remainders all have degree less than $n$. E.g. with $p=2$ and $n=2$, selecting $r(x)=x^2+x+1$, possible remainders are

$$0, 1, x, x+1$$

so there are four equivalence classes (i.e. all polynomials whose remainder mod $r(x)$ is 0, 1, $x$ or $x+1$).

More generally, there are $p$ possibilities for each coefficient, and $n$ of them: from the $x^0$ term up to and including the $x^{n-1}$ term. Thus there are $p^n$ elements in this set which jives with what the theory told us above, from vector-space considerations.

For the integers mod $p$, I said we often sloppily move between equivalence classes mod $p$, and canonical representatives. E.g. $7 \equiv 2 \bmod 5$ (correct), $\overline{7}=\overline{2}$ (correct), $7=2$ (sloppy). And I did that fully aware of the sloppiness — since it is tolerable, and since the alternative is an undesirable forest of overlines.

However, with this second mod — polynomials mod $r(x)$ — I will not be sloppy. Rather, I'll use a more precise notation from field theory. Let $x+\langle r(x)\rangle$ be the set of all multiples of $r(x)$, plus $x$. Let $u=x+\langle r(x)\rangle$. (If you like, $u=\overline{x}$ using the overline notation from section 2.1 on page 10.) These are all the polynomials in $\mathbb{F}_p[x]$ which are in the same equivalence class as $x$. Then I write

$$\mathbb{F}_p[x]/\langle r(x)\rangle \cong \mathbb{F}_p(u)$$

i.e. we *adjoin* to the field $\mathbb{F}_p$ a root $u$ of $r(x)$.

My first reaction to this a few months ago was, "Where did this $u$ come from? What was wrong with $x$?" But in fact, $x$ is an element of $\mathbb{F}_p[x]$, the polynomial ring, and $u$ is an element of the finite field $\mathbb{F}_{p^n} = \mathbb{F}_p[x]/\langle r(x)\rangle$ (which we haven't yet seen is really a field). This apparently subtle distinction turns out to be important later on, so that we don't go mad when we talk about splitting fields. And really, $u$ does behave much like $x$ does, so it's not a completely new thing: if $r(x)$ is, say, $x^4 + x + 1$ with $p = 2$, then $u^4 + u + 1$ is zero.

I will occasionally write such elements either using the full notation such as

$$0, 1, u, u + 1$$

or in the compact notation such as

$$00, 01, 10, 11$$

(Notice that, without context, one can't distinguish compactly written ring elements (in $x$) from field elements (in $u$). So, in what follows I will be careful to always make it clear what I'm referring to.)

## 3.6 Addition, subtraction and multiplication in $\mathbb{F}_{p^n}$

Addition and subtraction of elements of $\mathbb{F}_{p^n}$ is easy: Just do it componentwise. Addition and subtraction are just the same as in $\mathbb{F}_p[x]$ so I won't even write new examples.

For multiplication, there are two methods. The first is what I call the **ring-and-reduce** method; the second is **reduce-en-route**. (You can call them RAR and RER, and pretend you're a dog — or a car that won't start. Your acquaintances will think you're daft, if they didn't already.)

For the former (RAR), just combine lifting, ring multiplication and reduction mod $r(x)$, which were described in section 3.3 on page 18. (E.g. to compute $3 \cdot 4$ in $\mathbb{F}_p$, we lifted 3 and 4 from $\mathbb{F}_p$ to $\mathbb{Z}$, took 3 times 4 is 12 in the ring $\mathbb{Z}$, then divided by 5 to get the remainder 2 back in $\mathbb{F}_p$.) For example, with $p = 2$ and $r(x) = x^4 + x + 1$, to compute $(u^3 + 1) \cdot (u^2 + u + 1)$, first lift from $\mathbb{F}_{p^n}$ to $\mathbb{F}_p[x]$ to obtain $(x^3 + 1) \cdot (x^2 + x + 1)$, then multiply in $\mathbb{F}_p[x]$:

```
  1001
* 0111
------
  1001
```

```
 1001
1001
------
111111
```

which gets you the unreduced product. Then divide by $r(x)$ and take the remainder, discarding the quotient:

```
           11
      +--------
10011 | 111111
      | 10011
      +--------
      |  11001
      |  10011
      +--------
      |   1010
```

So, with $p = 2$ and $r(x) = x^4 + x + 1$,

$$1001 \cdot 0111 = 1010$$

or with full notation,

$$(u^3 + 1) \cdot (u^2 + u + 1) = u^3 + u$$

The second multiplication method (RER) is essentially the same, but is sometimes handier. Proceeding by example with $p = 2$ and $r(x) = x^4 + x + 1$,

$$u^4 + u + 1 = 0$$

from which

$$u^4 = -u - 1$$

With $p = 2$, minuses are pluses so

$$u^4 = u + 1$$

but please be careful for $p \neq 2$. In the RAR method, since input degrees are $n - 1$ or less, the degree of the unreduced product can be as big as $2n - 2$. In the RER method, whenever you see anything of degree $n$, reduce it at first appearance by replacing, for example, $u^4$ with $u + 1$.

Why RER? Two reasons. First, in a computer program, the RAR method takes twice the storage space for the unreduced product; the RER method does not. But more importantly for pencil-and-paper arithmetic, while the RAR method is probably easier for multiplication in general, the RER method is easier when you're multiplying by $u$, as we'll see in the next section.

## 3.7 Primitivity in $\mathbb{F}_{p^n}$

We saw above that the multiplicative group $\mathbb{F}_p^\times$ is cyclic, and so there are always primitive elements mod $p$. Your first guess for a generator might be 2, and often 2 is a generator, but not always. For $p = 7$, we saw that 2 doesn't generate $\mathbb{F}_7^\times$, but 3 does.

Likewise, it can be shown that the multiplicative group $\mathbb{F}_{p^n}^\times$ is cyclic, and so there are always primitive elements mod $r(x)$. Your first guess might be $u$, which sometimes is primitive but not always[4]. If $u$ is primitive, for brevity I say that $u$ is a **generating root**. Also, $r(x)$ in $\mathbb{F}_p[x]$ is said to be a **primitive polynomial**[5] if $u$ is a primitive element mod $r(x)$.

(Note: This is a somewhat non-standard use of the terms *primitive* and *generator* in the context of field theory, as described in more detail in section 5.7.)

We saw above that there exists a count function for the number of monic irreducibles given $p$ and $n$. That function always takes positive values, so you'll always find a monic irreducible. Likewise there exists a count function for the number of primitive monic irreducibles given $p$ and $n$ (see section 5.8). This function also always takes positive values, so you'll always find a primitive monic irreducible. In particular, when there is only one monic irreducible ($p = 2$, $n = 2$), then that monic irreducible is necessarily primitive.

## 3.8 Powers of $u$ and LFSRs

Taking powers of $u$ is where the RER method for multiplication shines. You can do the following:

- Start with $u$.
- In the compact notation, shift all coefficients left by one position. This amounts to multiplying by $u$.
- If there is a non-zero coefficient in the $u^n$ position, use $r(u)$ to get rid of it.

For example, with $p = 2$, $n = 4$, $r(x) = x^4 + x + 1$, which is 10011, $u^4 + u + 1 = 0$ so I may solve for $u^4$ to get $u^4 = -u - 1$, which is $u + 1$ since $p = 2$. So whenever I see 10000, I can subtract it out and add in 0011.

[4]Whether or not $u$ is a primitive element, it has some order in the multiplicative group. Lidl and Niederreiter [10] (as well as Berlekamp [3]) call this order the **period** of $r(x)$. In their tables, they call the period $e$.

[5]For polynomials with coefficients merely in a ring rather than a field, there is a completely different definition of primitive polynomial — namely, a polynomial the GCD of whose coefficients is a unit. However, it's clear from the context which kind of primitivity we're talking about.

| $k$ | $u^k$ | |
|---|---|---|
| 1 | | 0010 |
| 2 | | 0100 |
| 3 | | 1000 |
| 4 | 10000 | $=$ 0011 |
| 5 | | 0110 |
| 6 | | 1100 |
| 7 | 11000 | $=$ 1011 |
| 8 | 10110 | $=$ 0101 |
| 9 | | 1010 |
| 10 | 10100 | 0111 |
| 11 | | 1110 |
| 12 | 11100 | 1111 |
| 13 | 11110 | 1101 |
| 14 | 11010 | 1001 |
| 15 | 10010 | 0001 |

When $p = 2$, regardless of $n$, addition is the XOR operation (exclusive or). This shift-left-and-fold-in business can be implemented efficiently in an electronic circuit in which case it is called, aptly enough, a **linear feedback shift register**, or LFSR[6]. These circuits are important in coding theory and cryptography. With initial data set to 1, clearly the number of iterations before the sequence repeats will be maximal when $u$ is a primitive element, i.e. when $r(x)$ is primitive, since a left shift is equivalent to multiplying by $u$. Hence the saying that "primitive polynomials give rise to maximum-period linear feedback shift registers". Schneier [13] has a lot more to say about LFSRs. I also discuss LFSRs in part III of this paper. I discuss primitivity testing in greater detail in appendix E on page 92.

## 3.9 First reciprocation method in $\mathbb{F}_{p^n}$

We now know how to add, subtract and multiply in $\mathbb{F}_{p^n}$, and as with $\mathbb{Z}/m\mathbb{Z}$, all this works regardless of whether $r(x)$ is irreducible, i.e. whether or not $\mathbb{F}_p[x]/\langle r(x)\rangle$ is a field. But how do we divide? Given $a/b$, it suffices to know how to reciprocate $b$, since then we can use multiplication (which we already know how to do) to multiply $a$ and $1/b$.

As with $\mathbb{F}_p$, there are four methods for reciprocation. The first is simply a search, and for small $p^n$ this is quite fine. What is $1/u \bmod x^4 + x + 1$? You can try 0000, 0001, 0010, etc. until you happen upon 1001.

[6]Specifically, what I show here is a *reflected Galois-configuration LFSR*, which is distinct from Fibonacci-configuration LFSRs which you may encounter in the literature. Nonetheless, characteristic polynomials are the same.

## 3.10 Second reciprocation method in $\mathbb{F}_{p^n}$

You might wonder if the search method will always work, i.e. if $\mathbb{F}_{p^n}$ is really a field. The proof of the existence of reciprocals mod $r(x)$ is a constructive proof, and so gives our second method for reciprocation.

As with integers, given $f(u)$ in $\mathbb{F}_{p^n}$, lift to $f(x)$ in $\mathbb{F}_p[x]$. If $f(x)$ is 0 (or is a multiple of $r(x)$, which means $f(x)$ is 0 mod $r(x)$), then we expect no reciprocal. But if $f(x)$ is not a multiple of $r(x)$, then it is relatively prime to $r(x)$ since $r(x)$ is irreducible. This means its GCD with $r(x)$ is 1. Using the extended GCD, we can always write the output of the GCD algorithm as a linear combination of the two inputs, i.e.

$$1 = f(x)s(x) + r(x)t(x)$$

for some $s(x)$ and $t(x)$ in $\mathbb{F}_p[x]$. But $rt$ is a multiple of $r$ and so is zero mod $r$, so

$$1 \equiv f(x)s(x) \mod r(x)$$

which means the equivalence class of $s(x)$ is the reciprocal of the equivalence class of $f(x)$. After doing the mod operation to get back to $\mathbb{F}_{p^n}$, $s(u) = 1/f(u)$. (Actually, the GCD is a scalar, not necessarily 1, so go ahead and compute the GCD, then divide $s(x)$ by it.)

This is great for proving existence of reciprocals, but just as in the $\mathbb{F}_p$ case, the details of pencil-and-papering the extended GCD algorithm are tedious. (See appendix A on page 84 for more information on the extended-GCD method.) There are two better ways, as we will see.

## 3.11 Third reciprocation method in $\mathbb{F}_{p^n}$

Third, we can use Lagrange's theorem, just as in $\mathbb{F}_p$, and recognize that the multiplicative group $\mathbb{F}_{p^n}^{\times}$ has order $p^n - 1$. Therefore for all $a$ in $\mathbb{F}_{p^n}^{\times}$,

$$a^{p^n-1} = 1$$

from which

$$a^{p^n-2} = a^{-1}$$

This is the $p^n - 2$ rule for reciprocation. E.g. in $\mathbb{F}_{2^4}$, we just raise $a$ to the 14th power. As above, the repeated-squaring method for exponentiation (section 2.6, page 14) can dramatically reduce the number of multiplications needed.

Note also that we can make this true for 0 as well, i.e. for all of $\mathbb{F}_{p^n}$, by writing

$$x^{p^n} - x = 0$$

so we have a polynomial of which all elements of $\mathbb{F}_{p^n}$ are roots.

## 3.12 Fourth reciprocation method in $\mathbb{F}_{p^n}$; logarithms

Fourth, we can use the fact that $\mathbb{F}_{p^n}^{\times}$ is a cyclic group, then precompute a primitive element. (If we already know that $r(x)$ is a primitive polynomial, then we know $u$ will be a primitive element.) Then, we can use logarithms. If $g$ is a primitive element mod $r(x)$, then for all $a$ in $\mathbb{F}_{p^n}$,

$$a = g^k$$

for some $0 \leq k < p^n - 1$. Then

$$a^{-1} = g^{-k} = g^{p^n-1-k}$$

For example, with $p = 2$, $n = 4$ and $r(x) = x^4+x+1$ as above, and using $g = u$, take powers of $u$ using the RER method and sort by field element to obtain the log table:

| $\log_g(a) = k$ | $a = g^k$ |
|---:|---|
| 15 | 0001 |
| 1 | 0010 |
| 4 | 0011 |
| 2 | 0100 |
| 8 | 0101 |
| 5 | 0110 |
| 10 | 0111 |
| 3 | 1000 |
| 14 | 1001 |
| 9 | 1010 |
| 7 | 1011 |
| 6 | 1100 |
| 13 | 1101 |
| 11 | 1110 |
| 12 | 1111 |

and sort by log value to obtain the antilog table:

| $\log_g(a) = k$ | $a = g^k$ |
|---:|---|
| 1 | 0010 |
| 2 | 0100 |
| 3 | 1000 |
| 4 | 0011 |
| 5 | 0110 |
| 6 | 1100 |
| 7 | 1011 |
| 8 | 0101 |
| 9 | 1010 |
| 10 | 0111 |
| 11 | 1110 |
| 12 | 1111 |
| 13 | 1101 |
| 14 | 1001 |
| 15 | 0001 |

It's easy to reciprocate just by looking at the tables. E.g. $0011 = 0010^4$, so $1/0011 = 0010^{15-4} = 0010^{11} = 1110$.

You can multiply using the log and antilog tables by the familiar rule $\log_g(ab) = \log_g(a) + \log_g(b)$, where you must remember to do the addition mod $p^n - 1$:

$$ab = g^{\log_g(ab)} = g^{\log_g(a)+\log_g(b)}$$

Likewise you can do division in a single step (i.e. instead of finding the reciprocal, then multiplying by the reciprocal) using the familiar $\log_g(a/b) = \log_g(a) - \log_g(b)$:

$$a/b = g^{\log_g(a/b)} = g^{\log_g(a)-\log_g(b)}$$

As well, you can use logs to exponentiate (which will be useful below):

$$a^p = g^{\log_g(a^p)} = g^{p\log_g(a)}$$

See also the tables in section H.3 on page 99, or the much more extensive list in Lidl and Niederreiter [10].

In the pre-electronic era, such tables were the bee's knees. Nowadays, log tables might not seem as useful as they once were. But in fact, they can be used inside a computer program to implement lightning-fast finite-field arithmetic — at least, for fields that are small enough to tabulate in main memory. For example, the reference implementation of the AES cryptosystem [1] multiplies in $\mathbb{F}_{2^8}$ using log and antilog tables, with a modest memory requirement of half a kilobyte.

See also appendix B on page 86 for an efficient way to build large log tables (e.g. in a software implementation).

# Part II

# Galois theory

# Chapter 4

# The universal polynomial $x^{p^n} - x$

Now we know how to add, subtract, multiply and divide in $\mathbb{F}_{p^n}$, and we have the polynomial $x^{p^n} - x = 0$ of which all elements of $\mathbb{F}_{p^n}$ are roots. It's time now to look at the **universal polynomial** $x^{p^n} - x$.

The universal polynomial is really the key to the castle. From here we'll get all sorts of information about splitting fields, subfields, Galois structure, etc.

## 4.1 Reducibility of the universal polynomial over $\mathbb{F}_p$

The first theorem of interest is that $x^{p^n} - x$ is reducible in $\mathbb{F}_p[x]$, and factors[1] into all the monic irreducibles of degree $d$ for all $d$ dividing $n$.

For example, in section 3.4 on page 20 we listed out all the monic irreducibles for $p = 2$, and $n$ up to 4. What are the divisors $d$ of 4? Of course, 1, 2 and 4. It is easy to see (you can check by multiplying the factors back up again) that $x^{16} - x$ has the following six factors in $\mathbb{F}_2[x]$:

|  |  |  |  |
|---|---|---|---|
| $d = 1$: | $x,$ | $x + 1$ | |
| $d = 2$: | $x^2 + x + 1$ | | |
| $d = 4$: | $x^4 + x + 1,$ | $x^4 + x^3 + 1,$ | $x^4 + x^3 + x^2 + x + 1$ |

[1]Above we discussed how to find irreducibles. Assuming we know how to factor (in fact there are efficient factorization algorithms for $\mathbb{F}_p[x]$, e.g. the Berlekamp algorithm [10, Ch. 4.1] and appendix F), we know we can find monic irreducibles merely by factoring.

## 4.2 $\mathbb{F}_{p^n}$ as a splitting field

Now, this factorization of $x^{p^n} - x$ is in $\mathbb{F}_p[x]$ so it as yet has nothing to do with $\mathbb{F}_{p^n}$. The next key theorem is that $x^{p^n} - x$ splits[2] into linear factors in $\mathbb{F}_{p^n}[x]$, that is, $\mathbb{F}_{p^n}$ is the **splitting field** of $x^{p^n} - x$. (In fact, in a theoretical approach to finite fields, in constrast to the computational approach taken in this paper, one often starts with the universal polynomial and defines $\mathbb{F}_{p^n}$ as its splitting field.) Specifically, $x^{p^n} - x$ has no multiple roots, and the $p^n$ distinct roots are precisely the $p^n$ distinct elements of $\mathbb{F}_{p^n}$.

What does it really mean for $\mathbb{F}_{p^n}$ to be a splitting field of a polynomial with coefficients in $\mathbb{F}_p$, i.e. for $\mathbb{F}_{p^n}$ to be an extension field of $\mathbb{F}_p$? Here is precisely where it matters that I kept $x$ and $u$ distinct in section 3.5 on page 22. Also the compact notation comes in handy.

Here is an analogy with the reals: The polynomial $x^2 + 1$ is irreducible over $\mathbb{R}$, yet it splits into linear terms over $\mathbb{C}$, namely, $x^2 + 1 = (x + i)(x - i)$. Also recall that sometimes when first proving that $\mathbb{C}$ is a field, we write complex numbers $a + bi$ as ordered pairs $(a, b)$: $a$ is the base part and $b$ is the extension part. Likewise, numbers such as 0001, 0011, 1011 etc. are elements of the extension field $\mathbb{F}_{p^n}$, where the last digit is the base part (in $\mathbb{F}_p$) and the others are the extension parts.

So $x^2 + 1$ has coefficients strictly in $\mathbb{R}$, but it takes $\mathbb{C}$ to get it factored down to linear terms. Going the other way, notice that $(x + i)(x - i)$ has complex numbers in it, but when we do the multiplication back to $x^2 + 1$, the imaginary parts magically cancel out leaving only purely real coefficents.

Likewise for polynomials in $\mathbb{F}_p[x]$. It can be proved that any irreducible polynomial $f(x)$ of degree $n$ with coefficients in $\mathbb{F}_p$ splits into linear terms as long as the coefficients are extended to be in $\mathbb{F}_{p^n}$. For example, $x^2 + x + 1$ was above shown irreducible over $\mathbb{F}_2$. Yet it's easy to verify by FOILing that $(x+10)(x+11)$, using compact notation, or $(x+u)(x+u+1)$ using the full notation, is a factorization in $\mathbb{F}_{2^2}$. Recall that anything plus itself is zero in $\mathbb{F}_2$ and that $u^2 = u + 1$ in $\mathbb{F}_{2^2}$, using $x^2 + x + 1$ as the monic irreducible:

$$\begin{aligned}(x+u)(x+u+1) &= x^2 + ux + x + ux + u^2 + u \\ &= x^2 + x + u^2 + u \\ &= x^2 + x + u + 1 + u \\ &= x^2 + x + 1\end{aligned}$$

[2]More generally, a monic irreducible $f(x)$ of degree $n$ in $\mathbb{F}_p[x]$ factors into $d$ terms each of degree $n/d$ in $\mathbb{F}_{p^d}[x]$ for all $d|n$. For $d = n$, we have $n$ terms of degree 1, i.e. a splitting into linear factors.

## 4.3 The root chart

The factorization of $x^{p^n} - x$ into irreducibles over $\mathbb{F}_p$ is unique, but the details of the factorization into linear terms over $\mathbb{F}_{p^n}$ clearly will depend on our choice of $r(x)$ defining $\mathbb{F}_{p^n}$ arithmetic. (One of the field theorems is that any two splitting fields of the same polynomial are isomorphic, which together with the universal polynomial can be used to show that any two finite fields of order $p^n$ are isomorphic. Yet the arithmetic isn't bodily identical.)

For $p = 2$, $n = 4$, we saw in section 3.4 on page 20 that there are three possibilities for $r(x)$. Let

$$\begin{aligned} r_1(x) &= x^4 + x + 1 = 10011 \\ r_2(x) &= x^4 + x^3 + 1 = 11001 \\ r_3(x) &= x^4 + x^3 + x^2 + x + 1 = 11111 \end{aligned}$$

be the three monic irreducibles for $\mathbb{F}_{2^4}$.

I'll spare the details of the factorization algorithm (which in any case are better left to a computer). The factorization of $x^{16} - x$ into linears over $\mathbb{F}_{2^4}$, using $r_1$, $r_2$ and $r_3$ to define the finite-field arithmetic, is as follows. Since I'm factoring into linear terms over a splitting field, all the factors are of the form $(x - a)$ for some $a$. To save space, I'll just write the roots (the $a$'s).

Note: The compact notation in the first column (elements of the polynomial ring $\mathbb{F}_2[x]$) omits the variable $x$; the compact notation in subsequent columns (elements of the field $\mathbb{F}_{2^4}$) omits the variable $u$.

Root chart for $x^{16} - x$ over $\mathbb{F}_{2^4}$

| Irr. factors over $\mathbb{F}_2$ | Roots mod $r_1 = 10011$ | | | | Roots mod $r_2 = 11001$ | | | | Roots mod $r_3 = 11111$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00010 | 0000 | | | | 0000 | | | | 0000 | | | |
| 00011 | 0001 | | | | 0001 | | | | 0001 | | | |
| 00111 | 0110 | 0111 | | | 1011 | 1010 | | | 1101 | 1100 | | |
| $r_1$: 10011 | 0010 | 0100 | 0011 | 0101 | 0111 | 1100 | 0110 | 1101 | 0111 | 1010 | 0110 | 1011 |
| $r_1$: 11001 | 1011 | 1001 | 1101 | 1110 | 0010 | 0100 | 1001 | 1110 | 0011 | 0101 | 1110 | 1001 |
| $r_1$: 11111 | 1000 | 1100 | 1111 | 1010 | 1000 | 1111 | 0011 | 0101 | 1111 | 1000 | 0010 | 0100 |

Also, in the first column where I write the monic irreducibles for degrees $d$ dividing 4, I've marked the $r_i$'s since these are the same three polynomials that define $\mathbb{F}_{2^4}$ the arithmetic. That is, we are factoring $r_i$ mod $r_j$. (The smaller-degree factors of $x^{p^n} - x$ have to do with subfields, as we shall see below.)

Note: This chart may be obtained in a top-down fashion by factoring the universal polynomial $x^{p^n} - x$ over $\mathbb{F}_{p^n}$, or it may be obtained in a bottom-up fashion (with far less effort). Both methods are described in appendix G on page 96.

The root chart is chock-full of information, as the next several sections will explore.

# Chapter 5

# Basic Galois theory for $\mathbb{F}_{p^n}$

*Eadem mutata resurgo (Though changed, I arise the same)* — Jakob Bernoulli

One of this paper's main purposes is to provide examples to assist in the study of Galois theory. Now, since (as we shall see) the Galois group of a finite field is always cyclic, and since all extensions are Galois[1], one does not obtain the full panoply of behavior exhibited by algebraic number fields. However, finite fields are nice in part because we can write down all the elements of the field (try that with an algebraic number field!). Also, once you are comfortable with computing in finite fields you'll be in a good position to compute in algebraic number fields — a lot of the same quotient-and-remainder business over a polynomial ring comes into play.

## 5.1 Subfields

For each $\mathbb{F}_{p^n}$, there is a unique subfield $\mathbb{F}_{p^d}$ for all $d \,|\, n$. Why? First, for the dimension $p^d$: suppose you have a subfield of $\mathbb{F}_{p^n}$. By the vector-space considerations as discussed in section 3.2, the subfield is an extension field of its prime subfield $\mathbb{F}_p$ and so its size is necessarily a $p$-power, $p^d$, for some $1 \le d \le n$. So, we can call it an $\mathbb{F}_{p^d}$. For the same reason, since $\mathbb{F}_{p^n}$ is an extension field of $\mathbb{F}_{p^d}$, $p^n$ is a power of $p^d$, from which $d|n$.

(As well, if $\mathbb{F}_{p^d}$ is a subfield of $\mathbb{F}_{p^n}$, the multiplicative group of the former must be a subgroup of the multiplicative group of the latter, so $p^d - 1$ must divide $p^n - 1$. It's easy to see, using integer division, that $d|n$ if and only if $p^d - 1 \,|\, p^n - 1$.)

[1]The Galois group is cyclic, hence abelian, so all subgroups of the Galois group are normal, so all the subfields are Galois over $\mathbb{F}_p$.

So, *if* there is a subfield of $\mathbb{F}_{p^n}$, then it is an $\mathbb{F}_{p^d}$ for $d|n$. As above, the prime subfield $\mathbb{F}_p$ always exists. The fact that a unique $\mathbb{F}_{p^d}$ does in fact exist as a subfield of $\mathbb{F}_{p^n}$ for all the remaining $d|n$ is proved using Galois theory[2].

For example, for $\mathbb{F}_{2^6}$ we expect subfields $\mathbb{F}_2$, $\mathbb{F}_{2^2}$ and $\mathbb{F}_{2^3}$, along with $\mathbb{F}_{2^6}$ itself. These subfields are easy to construct. Recall that any field consists of 0 along with the field's multiplicative group, and that the multiplicative group of any finite field is cyclic. Any subfield of a finite field must contain 0, and the multiplicative group of the subfield must be a subgroup of the multiplicative group of the larger field. The former will have $p^d - 1$ elements; the latter will have $p^n - 1$ elements, and as above, $p^d - 1 \,|\, p^n - 1$ when $d|n$.

Suppose we have a cyclic group of order $ab$ with generator $g$. Then there is a subgroup of order $b$ generated by $g^a$. E.g. $(\mathbb{Z}_6, \oplus)$ has order 6 and generator 1; elements are 0, 1, 2, 3, 4 and 5. There is an order-3 subgroup generated by 2, with elements 0, 2 and 4.

Back to finite fields, by the same token, if $\mathbb{F}_{p^n}^\times$ has generator $g$, then $\mathbb{F}_{p^d}^\times$ has generator $g^{(p^n-1)/(p^d-1)}$. For example, let $g$ be a generator of $\mathbb{F}_{2^6}^\times$. To construct $\mathbb{F}_{2^3}$ as a subfield of $\mathbb{F}_{2^6}$, let $g$ be a generator of $\mathbb{F}_{2^6}^\times$. Then $\mathbb{F}_{2^3}$ is the 0 element along with the 7 distinct powers of $g^{63/7} = g^9$. (For now, I'm saying that this $\mathbb{F}_{2^3}$ is the specified subset of $\mathbb{F}_{2^6}$, but I'm not yet saying which elements of this $\mathbb{F}_{2^3}$ map isomorphically to which elements of a standard $\mathbb{F}_{2^3}$, where by "standard" I mean a field defined by a monic irreducible cubic in $\mathbb{F}_2[x]$. More on that below.)

In the root chart, we can see subfields. Look at the elements 0 and 1, along with the roots of $x^2 + x + 1$. Mod $x^4 + x + 1$, these four elements are 0, 1, $u^2 + u$ and $u^2 + u + 1$. Using addition, subtraction, multiplication and division you can easily see that these four elements satisfy all the field axioms, and so must be a field.

With reference to the antilogarithm table in section 3.12 on page 28, note that $u^2 + u$, $u^2 + u + 1$ and 1 have logarithms 5, 10 and 15, respectively with respect to the generator $u$, i.e. the elements of the copy of $\mathbb{F}_{2^2}$ inside $\mathbb{F}_{2^4}$ are 0, $u^5$, $u^{10}$ and $u^{15} = 1$. I refer to this as the **subfield/log criterion**: elements of a subfield may be determined by their logarithms. Specifically, an element of $\mathbb{F}_{p^n}$ is also an element of an $\mathbb{F}_{p^d}$, where $d|n$, if and only if it is zero or its logarithm is a multiple of $(p^n - 1)/(p^d - 1)$.

On the other hand, earlier in this paper we saw one (in fact, the only) monic irreducible for $p = 2$, $n = 2$: namely, $x^2 + x + 1$. (The root chart bears this out since there is only one quadratic factor for $x^{16} - x$.) This leads to an $\mathbb{F}_{2^2}$ with elements 0, 1, $v$, $v + 1$ where I use $v$ instead of $u$ to emphasize that these are different equivalence classes. Since these two copies of $\mathbb{F}_{2^2}$ are fields and have the same order, they must be isomorphic. Below we'll construct an explicit isomorphism between the two. More importantly, *this* is what people mean when

[2]As will be seen below, $\mathrm{Gal}(\mathbb{F}_{p^n}/\mathbb{F}_p) = \langle \rho \rangle$ with $|\rho| = n$, where $\rho$ is the $p$-power map. From group theory, a cyclic group of order $n$ has a unique subgroup of order $d$ for all $d|n$. By the Galois correspondence, this means there is a unique $\mathbb{F}_{p^d}$ for all $d|n$.

they say things like "$\mathbb{F}_{2^2} \subset \mathbb{F}_{2^4}$": They do *not* mean that the $x^2+x+1$ field is a subfield of the $x^4+x+1$ subfield with $v$ mapping to $u$ (it clearly is not). Rather they mean that the $\mathbb{F}_{2^2}$ field of $x^2+x+1$ may be **embedded** (i.e. mapped with a 1-1 homomorphism) into the $x^4+x+1$ field $\mathbb{F}_{2^4}$.

## 5.2 Frobenius automorphisms

It is easy to show that the map which raises elements of $\mathbb{F}_{p^n}$ to the $p$th power is an automorphism on $\mathbb{F}_{p^n}$, called the **Frobenius automorphism**. For the additive homomorphism property, use the **freshman's dream** for fields of characteristic $p$: $(a+b)^p = (a^p + b^p)$. (In the binomial expansion, $\binom{p}{i}$ is 0 mod $p$ except for $i = 0, p$ in which case it is 1.) For the multiplicative homomorphism property, $(ab)^p = a^p b^p$. For injectivity, $a^p = b^p \implies a^p - b^p = 0 = (a-b)^p \implies a = b$ since fields have no zero divisors. For surjectivity, the preimage of any $a$ under the Frobenius map is $a^{p^{n-1}}$.

Recall from section 2.5 on page 13 that we have $a^p = a$ for all elements of $\mathbb{F}_p$. That is, the automorphism $\rho$ which takes $p$th powers is the identity on the prime subfield $\mathbb{F}_p$ of $\mathbb{F}_{p^n}$.

## 5.3 Galois group and Frobenius orbits

Look at the action of $\rho$ on an element $a$ of $\mathbb{F}_{p^n}$: $\rho(a) = a^p$; $\rho^2(a) = \rho(\rho(a)) = (a^p)^p = a^{p^2}$; etc. up to $\rho^{n-1}(a) = a^{p^{n-1}}$. But then $\rho^n(a) = a^{p^n} = a$ so $\rho^n = \rho^0 = \iota$, the identity map. This is how we see[3] that the Galois group is in fact cyclic of order $n$, generated by the $p$-power map $\rho$. That is:

$$\mathrm{Gal}(\mathbb{F}_{p^n}/\mathbb{F}_p) = \langle \rho \rangle;\ |\rho| = n$$

Given $a$ in $\mathbb{F}_{p^n}$, the $\rho^i(a)$'s are the **orbit** of $a$ under the **action** of the automorphism group. There is an obvious equivalence relation on the field for any elements which are $p^i$ powers of one another. These equivalence classes partition the field, as is clear from the root chart.

Also we saw that $a^{p^n} = a$ for all elements of $\mathbb{F}_{p^n}$, so as well $a^{p^d} = a$ for all elements of $\mathbb{F}_{p^d}$. This means that when we look at the Galois correspondence below, when we view $\mathbb{F}_{p^d}$ as a subfield of $\mathbb{F}_{p^n}$ we will have $\rho^d$ being an identity on the base field $\mathbb{F}_{p^d}$ and we will expect

$$\mathrm{Gal}(\mathbb{F}_{p^n}/\mathbb{F}_{p^d}) = \langle \rho^d \rangle;\ |\rho^d| = n/d$$

[3]That the $\rho^i$'s account for *all* automorphisms on $\mathbb{F}_{p^n}$ is proved using Galois theory: the $n$ distinct powers of $\rho$ are enough to fill the Galois group since $[\mathbb{F}_{p^n} : \mathbb{F}_p] = n$.

which will have group order $n/d$. E.g. the Galois group for $\mathbb{F}_{2^6}$ over $\mathbb{F}_{2^2}$ will consist of the three automorphisms $a \mapsto a$, $a \mapsto a^4$ and $a \mapsto a^{16}$.

## 5.4 Characteristic and minimal polynomials

The **characteristic polynomial** for an element $a$ of a field $F$ with respect to a base field $K$ is defined to be

$$\prod_{\sigma \in \mathrm{Gal}(F/K)} x - \sigma(a)$$

which is clearly monic but not necessarily irreducible. For $\mathbb{F}_{p^n}$ over $\mathbb{F}_p$, this becomes

$$\prod_{i=0}^{n-1} x - a^{p^i}$$

For our example $\mathbb{F}_{2^4}$ over $\mathbb{F}_2$, this is

$$(x-a)(x-a^2)(x-a^4)(x-a^8)$$

The **minimal polynomial** of $a$ is the unique monic irreducible polynomial having $a$ as a root. It is shown in field theory [4, Ch. 14.2] that the characteristic polynomial is a power of the minimal polynomial, and in particular

$$C_a(x) = M_a(x)^{n/d}$$

where $d$ is the degree of the smallest extension field over the base containing $a$, which is the same as the degree of $a$'s minimal polynomial. Since $n/d$ is an integer, $d$ always divides $n$. That is, the minimal polynomial is like the characteristic polynomial, but we only include *distinct* $\sigma(a)$'s.

In the root chart, this behavior is very clear:

- For the elements 0, 1 of $\mathbb{F}_{2^4}$ which are also elements of the base field $\mathbb{F}_2$, $a^2 = a$ so the product

  $$(x-a)(x-a^2)(x-a^4)(x-a^8)$$

  reduces to

  $$(x-a)^4$$

  where the minimal polynomial is

  $$(x-a)$$

  which is of degree one (the 4 applications of the Frobenius map wrap around four times), corresponding to the fact that 0 and 1 are elements of $\mathbb{F}_2$.

- For the elements 110, 111 of $\mathbb{F}_{2^4}$ which are also elements of the intermediate field $\mathbb{F}_{2^2}$, $a^4 = a$ so the product
$$(x-a)(x-a^2)(x-a^4)(x-a^8)$$
reduces to
$$(x-a)^2(x-a^2)^2$$
where the minimal polynomial is
$$(x-a)(x-a^2)$$
which is of degree two (the 4 applications of the Frobenius map wrap around twice), corresponding to the fact that 110 and 111 are elements of $\mathbb{F}_{2^2}$ but not elements of $\mathbb{F}_2$.

- For the remaining elements of $\mathbb{F}_{2^4}$, the $a^{p^i}$'s are all distinct so the product
$$(x-a)(x-a^2)(x-a^4)(x-a^8)$$
does not reduce and the minimal polynomial is the same as the characteristic polynomial, which is of degree four (the 4 applications of the Frobenius map do not wrap around), corresponding to the fact that these elements of $\mathbb{F}_{2^4}$ are not elements of any smaller field.

## 5.5 Conjugates

Two elements of the field are said to be **conjugates** if they are roots of the same minimal polynomial, and clearly this also makes an equivalence relation on the elements of the field. From the root chart we see a clear example that conjugacy classes are the same as Frobenius orbits. For example, the polynomial $r_1 = x^4 + x + 1$, using $r_1$ arithmetic, has roots $u$, $u^2$, $u^4 = u + 1$, $u^8 = u^2 + 1$.

## 5.6 Orders and logs

In section 4.3 I showed the root chart for $\mathbb{F}_{2^4}$. Here is the same chart, but instead of field elements I show their logarithms. The polynomials $r_1(x) = x^4+x+1$ and $r_2(x) = x^4+x^3+1$ are primitive, so $g_1 = g_2 = u = 10$ can be used as a generator; $r_3(x) = x^4 + x^3 + x^2 + x + 1$ is imprimitive and I've selected $g_3 = u + 1 = 11$ as a generator.

Logarithmic root chart for $x^{16} - x$ over $\mathbb{F}_{2^4}$

| Irr. factors | Roots mod | | | | Roots mod | | | | Roots mod | | | | Order |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| over $\mathbb{F}_2$ | $r_1 = 10011$ | | | | $r_2 = 11001$ | | | | $r_3 = 11111$ | | | | |
| | $g_1 = 0010$ | | | | $g_2 = 0010$ | | | | $g_3 = 0011$ | | | | |
| 00010 | N/A | | | | N/A | | | | N/A | | | | N/A |
| 00011 | 0 | | | | 0 | | | | 0 | | | 1 | |
| 00111 | 5 | 10 | | | 5 | 10 | | | 5 | 10 | | 3 | |
| $r_1$: 10011 | 1 | 2 | 4 | 8 | 7 | 14 | 13 | 11 | 7 | 14 | 13 | 11 | 15 |
| $r_2$: 11001 | 7 | 14 | 13 | 11 | 1 | 2 | 4 | 8 | 1 | 2 | 4 | 8 | 15 |
| $r_3$: 11111 | 3 | 6 | 12 | 9 | 3 | 6 | 12 | 9 | 3 | 6 | 12 | 9 | 5 |

Here the subfield/log criterion is clear: 1, which is in $\mathbb{F}_2$, is $g_i^{15}$; the other two non-zero elements of $\mathbb{F}_{2^2}$ are $g_i^5$ and $g_i^{10}$. All conjugates have a log which is some power of two times the logs of the other conjugates.

Also, as Lidl and Niederreiter [10] point out, roots of $r_1$ and $r_2$ (which are primitive) have full order, i.e. 15, regardless of which $r_j$ is used for the splitting arithmetic; roots of $r_3$ (which is imprimitive) have smaller order regardless of $r_j$.

## 5.7 Two definitions of "generator"

In this paper, when I refer to a generator of $\mathbb{F}_{p^n}$, I refer to an element that "generates" the multiplicative group $\mathbb{F}_{p^n}^{\times}$, i.e. an element which can produce all the other non-zero elements only using multiplication. As well, I call such an element "primitive".

I should come clean and admit that this terminology is non-standard (although multiple authors use the term "primitive polynomial" in the sense I do). In field theory, it is common to define the field "generated" by an element $\alpha$ to be the smallest field containing $\alpha$, and to say that $\alpha$ is a "primitive" element of a field $F$ if it generates $F$ in this sense. This simply means that $\alpha$ is in $F$, but not in any proper subfield.

That is, when one talks of a field generated by $\alpha$, one refers to all the elements which can be obtained from $\alpha$ using multiplication *and* addition. Clearly, all you need to generate an $\mathbb{F}_{p^n}$ is any element with a degree-$n$ minimal polynomial. Then, multiplication will give you $n$ distinct powers $1, \alpha, \alpha^2, \ldots, \alpha^{n-1}$, and then addition will give you all the remaining $p^n$ elements.

In the root chart above, it is clear that there are 12 elements which are field generators in this sense: roots of the three quartics 10011, 11001 and 11111. However, only 8 of those 12 generate the multiplicative group using only multiplication.

My choice of using the terms "primitive" and "generator" in the group-theoretic sense rather than in the field-theoretic sense is because logarithms are key throughout this paper, and in order to define logarithms one needs a generator of the multiplicative group.

## 5.8 The irreducible-count and primitive-count formulae

Above I noted that there are formulae for the number of monic irreducibles of a given degree in $\mathbb{F}_p[x]$, and for the number of those which are also primitive. The root chart makes the derivations of those formulae obvious.

For the irreducible-count formula [4, Ch. 14.3]: When we factor $x^{p^n} - x$ in $\mathbb{F}_p[x]$, the result is all the monic irreducibles in $\mathbb{F}_p[x]$ of degree $d$ for all $d$ dividing $n$. Equating degrees, we have that

$$p^n = \sum_{d|n} dI(d)$$

where $I(d)$ is the number of monic irreducibles of degree $d$. E.g. in the root chart for $\mathbb{F}_{2^4}$, $I(1) = 2$ (2 factors of egree 1), $I(2) = 1$ (1 factor of degree 2) and $I(4) = 3$ (3 factors of degree 4), and $2 \cdot 1 + 1 \cdot 2 + 3 \cdot 4 = 16$. Take as given the Möbius inversion formula from elementary number theory, and recall that $\mu(m)$, for integer $m$, is 0 if $m$ is not squarefree, $+1$ if $m$ has an even number of distinct prime factors, or $-1$ if $m$ has an odd number of distinct prime factors. Note that we have a function of $n$ (namely, $p^n$) expressed in terms of a sum over divisors of $d$ of $n$ (namely, $dI(d)$) so we may invert $d$ and $n$ to write

$$nI(n) = \sum_{d|n} \mu(d)p^{n/d}$$

from which

$$I(n) = \frac{1}{n}\sum_{d|n} \mu(d)p^{n/d}$$

This function always has positive value, which then guarantees the existence of at least one monic irreducible for any prime $p$ and positive integer $n$.

For the primitive-count formula [11]: The multiplicative group of $\mathbb{F}_{p^n}$ is cyclic with order $p^n - 1$. From elementary number theory we know that this cyclic group is generated by elements whose logs are relatively prime to $p^n - 1$. The number of such logs has a name: the Euler *totient function*, or *phi function*. Thus there are $\phi(p^n - 1)$ generators of the multiplicative group of $\mathbb{F}_{p^n}$. Since they generate all of $\mathbb{F}_{p^n}^{\times}$, they can't be elements of a proper subfield so their minimal polynomials all have degree $n$. (In the logarithmic root chart for $\mathbb{F}_{2^4}$, the polynomials 10011 and 11001 are primitive since their roots have full order 15; 11111 is imprimitive since its roots only have order 5.) Also, if one of the roots of a polynomial is primitive, all of them are[4]. So the $\phi(p^n - 1)$ generators are accounted for by their $\phi(p^n - 1)/n$ minimal polynomials, each of which is primitive. Like the primitive-count function, this irreducible-count function always takes a positive value, guaranteeing the existence of at least one primitive monic irreducible for any prime $p$ and positive integer $n$.

## 5.9 The Galois correspondence

We saw that the Galois group of $\mathbb{F}_{p^n}$ is cyclic of order $n$, generated by the Frobenius automorphism $\rho$. Also we just took a look at subfields. It is now natural to ask more about the Galois correspondence for finite fields.

From the above discussions about subfields and Galois groups we have the following:

$$\begin{tikzcd}
\mathbb{F}_{2^4} = \mathbb{F}_2(\gamma),\ \gamma^4 + \gamma + 1 = 0 = \mathbb{F}_2(\alpha)(\beta) \arrow[d, dash] & \langle \rho^4 = \iota \rangle \arrow[d, dash] \\
\mathbb{F}_{2^2} = \mathbb{F}_2(\alpha),\ \alpha^2 + \alpha + 1 = 0 \arrow[d, dash] & \langle \rho^2 \rangle \arrow[d, dash] \\
\mathbb{F}_2 & \langle \rho \rangle
\end{tikzcd}$$

As mentioned above:

- The $p$-power map, $\rho$, is the identity on $\mathbb{F}_2$. The four maps $\rho$, $\rho^2$, $\rho^4$ and $\rho^{16} = \iota$ are distinct automorphisms on $\mathbb{F}_{2^4}$, forming the Galois group of $\mathbb{F}_{2^4}$ over $\mathbb{F}_2$.
- The $p$-power map, $\rho$, is the identity on $\mathbb{F}_2$. The two maps $\rho$ and $\rho^2 = \iota$ are distinct automorphisms on $\mathbb{F}_{2^2}$, forming the Galois group of $\mathbb{F}_{2^2}$ over $\mathbb{F}_2$.
- The $p^2$-power map, $\rho^2$, is the identity on $\mathbb{F}_{2^2}$. The two maps $\rho^2$ and $\rho^4 = \iota$ are distinct automorphisms on $\mathbb{F}_{2^2}$, forming the Galois group of $\mathbb{F}_{2^4}$ over $\mathbb{F}_{2^2}$.

[4]Given $a \in \mathbb{F}_{p^n}$, an arbitrary conjugate is $a^{p^i}$ for some integer $i$. If $a^k = 1$ then $(a^{p^i})^k = a^{kp^i} = (a^k)^{p^i} = 1$. If $a^k \neq 1$ then likewise $(a^{p^i})^k = (a^k)^{p^i}$, and since raising to the $p^i$ power is an automorphism on $\mathbb{F}_{p^n}$, a non-1 element is not sent to 1. Therefore conjugates have the same multiplicative order.

## 5.10 Field isomorphisms

Above we saw that in an automorphism, i.e. an isomorphism from a finite field to itself, roots of a given minimal polynomial map to other roots of the same minimal polynomial. Also, that mapping was $\alpha \mapsto \alpha^{p^i}$ for some $0 \leq i < n$.

Also, above several times I mentioned that finite fields of a given order are isomorphic. But, what is the nature of that isomorphism? It is one thing to prove the existence of something, but it is another thing to produce that item. It can be shown that these isomorphisms can be explicitly constructed as follows:

- A field homomorphism must be a group homomorphism on the additive group. In particular, 0 maps to 0.
- A field homomorphism must be a group homomorphism on the multiplicative group as well, and the multiplicative group is cyclic. Recall that for a homomorphism $\phi$ between cyclic groups, it suffices to specify the image of a generator $g$. By the homomomorphism property, for any $a = g^k$ in the first field, $\phi(a) = \phi(g^k) = \phi(g)^k$ in the second field. (Not all the group homomorphisms will be field homomorphisms, of course — they all preserve multiplication, but they won't necessarily all preserve addition as well.)
- Since the multiplicative groups of both fields are cyclic, there is at least one primitive element in the first field. Call that primitive element $g_1$. Also it is clear that primitive elements will have a minimal polynomial of degree $n$, not less, i.e. the minimal polynomial for a primitive element is the same as its characteristic polynomial. This is because a primitive element of the multiplicative group cannot be an element of a multiplicative subgroup, so such an element can't be belong to a subfield either.
- Find the minimal polynomial of the chosen generator in the first field.
- Map $g_1$ to one of the $n$ roots of that same minimal polynomial in the second field. It can be shown that these roots are all primitive elements in the second field. Call your choice $g_2$.
- Zero maps to zero. All remaining elements $a_1$ in the first field map to $a_2$ in the second field by
$$a_2 = g_2^{log_{g_1}(a_1)}$$

For example, in the root chart, we can construct an isomorphism between the $x^4 + x + 1$ field and the $x^4 + x^3 + 1$ field. Consult the log tables in appendix H.3 on page 99 to find a generator for the $x^4 + x + 1$ field, namely, 0010. Its minimal polynomial is in the left-hand column, namely, $x^4 + x + 1$. In the $x^4 + x^3 + 1$ field, $x^4 + x + 1$ has roots 0110, 0111, 1100 and 1101 so there are four possible isomorphisms. I'll arbitrarily choose the first. Remember that 0 maps to 0, and consult the log tables in appendix H.3 on page 99 to write down the following:

| | $x^4+x+1$: | $x^4+x^3+1$: |
|---:|---:|---:|
| $k$ | $g^k$ | $\phi(g)^k$ |
| 1 | 0010 | 0110 |
| 2 | 0100 | 1101 |
| 3 | 1000 | 0101 |
| 4 | 0011 | 0111 |
| 5 | 0110 | 1011 |
| 6 | 1100 | 1000 |
| 7 | 1011 | 0010 |
| 8 | 0101 | 1100 |
| 9 | 1010 | 0011 |
| 10 | 0111 | 1010 |
| 11 | 1110 | 1110 |
| 12 | 1111 | 1111 |
| 13 | 1101 | 1001 |
| 14 | 1001 | 0100 |
| 15 | 0001 | 0001 |
| N/A | 0000 | 0000 |

## 5.11 Embeddings and unembeddings

The above isomorphism works as well for embedding a smaller field in a larger one. Here, we want to see not only that an $\mathbb{F}_{p^d}$ is a subset of an $\mathbb{F}_{p^n}$, where $d|n$, but we also want to see which element of the former maps to which element of the latter, in a way that preserves addition and multiplication. Likewise, given an $\mathbb{F}_{p^d}$ embedded inside an $\mathbb{F}_{p^n}$, it would be nice to map those elements back to a standard $\mathbb{F}_{p^d}$, i.e. an $\mathbb{F}_{p^d}$ defined by a degree-$d$ monic irreducible in $\mathbb{F}_p[x]$.

The technique is as follows.

- Find the minimal polynomial of an arbitrarily chosen generator $g_s$ in the smaller field, the standard $\mathbb{F}_{p^d}$. This will be a degree-$d$ monic irreducible in $\mathbb{F}_p[x]$.
- Factor the minimal polynomial in the larger field. (To do this, either construct or consult a root chart, e.g. appendix G, or use a factorization algorithm, e.g. appendix F.) Select one of the roots; call your choice $g_l$. Note that this is not a generator of all of $\mathbb{F}_{p^n}$, but rather only a generator of the subfield $\mathbb{F}_{p^d}$ inside $\mathbb{F}_{p^n}$.

- To embed the smaller $\mathbb{F}_{p^d}$ into the larger $\mathbb{F}_{p^n}$, zero maps to zero and all remaining elements $a_s$ in the smaller field map to $a_l$ in the second field by

$$a_l = g_l^{log_{g_s}(a_s)}$$

- To unembed the copy of $\mathbb{F}_{p^d}$ contained in the larger field $\mathbb{F}_{p^n}$, to a standard $\mathbb{F}_{p^d}$, zero maps to zero and the remaining elements $a_l$ in the larger field map to $a_s$ in the second field by

$$a_s = g_s^{log_{g_l}(a_l)}$$

  Note that this log will only be defined for those element of $\mathbb{F}_{p^n}$ which are actually elements of $\mathbb{F}_{p^d}$: this is another instance of the subfield/log criterion.

Let's explicitly construct an isomorphism between $\mathbb{F}_{2^2}$ as defined by $x^2 + x + 1$ and $\mathbb{F}_{2^2}$ as a subfield of $\mathbb{F}_{2^4}$ with $x^4 + x + 1$. We know that $g_s = u$ is a generator of the former, with minimal polynomial $x^2 + x + 1$. From the root chart, we know that $x^2 + x + 1$ has roots $v^2 + v$ and $v^2 + v + 1$ in $\mathbb{F}_{2^4}$. Select $g_l = v^2 + v$. Then:

|  | $x^2 + x + 1$: | $x^4 + x + 1$ subfield: |
|---|---|---|
| $k$ | $g_s^k$ | $g_l^k$ |
| N/A | 00 | 0000 |
| 1 | 10 | 0110 |
| 2 | 11 | 0111 |
| 3 | 01 | 0001 |

## 5.12 Composite fields

Using this technique it is easy to compute composites of finite fields. Given two fields of degree $n$ and $m$, let $c$ be the least common multiple of $n$ and $m$. Use the above formula to map elements of $\mathbb{F}_{p^n}$ into $\mathbb{F}_{p^c}$, and elements of $\mathbb{F}_{p^m}$ into $\mathbb{F}_{p^c}$. Any irreducible degree-$c$ polynomial in $\mathbb{F}_p[x]$ may be used to define the $\mathbb{F}_{p^c}$ arithmetic.

## 5.13 Algebraic closure of $\mathbb{F}_p$

Above I used the example of $x^2 + 1$ in $\mathbb{R}[x]$. In this case, $\mathbb{C}$ is the splitting field for $x^2 + 1$, that is, $\mathbb{C}$ splits this *particular* polynomial. But moreover, $\mathbb{C}$ is the *algebraic closure* of $\mathbb{R}[x]$, i.e. $\mathbb{C}$ splits *all* polynomials in $\mathbb{R}[x]$. Furthermore, any polynomial in $\mathbb{C}[x]$ splits in $\mathbb{C}[x]$.

For finite fields, the situation is a little different: $\mathbb{F}_{p^n}$ is the splitting field for any particular irreducible degree-$n$ polynomial in $\mathbb{F}_p[x]$ (in fact, for *all* degree-$n$ irreducibles), but $\mathbb{F}_{p^n}$ fails to split higher-degree polynomials with coefficients in $\mathbb{F}_p$. Furthermore, $\mathbb{F}_{p^n}$ does not split all degree-$n$ polynomials with coefficients in the extension field $\mathbb{F}_{p^n}$. It can be shown, in fact [4, Ch. 14.3], that no finite field is algebraically closed, and that the algebraic closure of $\mathbb{F}_p$, written $\overline{\mathbb{F}_p}$, is the infinite field which is the union of $\mathbb{F}_{p^n}$ for all integer $n$.

What does it mean to take such a union? This is the kind of thing we did in section 5.11, where we embedded a smaller field in a larger one. Computation in $\overline{\mathbb{F}_p}$ requires liberal amounts of these kinds of embeddings.

# Chapter 6

# Multiple extensions

Up to now in this document we've looked at single extensions, i.e. an $\mathbb{F}_{p^n} = \mathbb{F}_p(\alpha)$ is an extension field of $\mathbb{F}_p$ when $\alpha$ is a root of some degree-$n$ monic irreducible polynomial in $\mathbb{F}_p[x]$. Here I want to consider multiple extensions: for example, let $\alpha$ again be a root of some degree-$n$ monic irreducible polynomial in $\mathbb{F}_p[x]$, and then let $\beta$ be a root of some degree-$m$ monic irreducible polynomial in $\mathbb{F}_p(\alpha)[y]$. The result is an $\mathbb{F}_{p^{nm}}$ which may be written as $\mathbb{F}_p(\alpha)(\beta)$.

The terms *single extension* and *multiple extension* are somewhat non-standard. In field theory, it is shown that for so-called *perfect fields*, which includes all the finite fields, any multiple extension $F(\alpha)(\beta)$ is *simple* if it can be written as $F(\gamma)$ for some $\gamma$. For finite fields, this is always the case — we *can* write all extensions thus. However, in this section I want to consider the case when we *choose* not to do so. Why? Principally, to test our understanding of Galois theory. In particular, I want to show how to compute in a multiple extension, and to show how elements of an $\mathbb{F}_p(\gamma)$ line up with their counterparts in an $\mathbb{F}_p(\alpha)(\beta)$.

## 6.1 Minimal polynomials over intermediate fields

Of course, we can think of $\mathbb{F}_{2^2}$ as an extension of $\mathbb{F}_2$, all by itself. This $\mathbb{F}_{2^2}$ is $\mathbb{F}_2$ adjoin $\alpha$ where $\alpha$ is a root of $x^2 + x = 1$. (Up to now I've been using $u$ for the extension variable, but now that I need more than one I'll switch over to $\alpha$, $\beta$ and $\gamma$ for this section and the next.) This is a quadratic extension whose elements are linear combinations of 1 and $\alpha$ with coefficients in the base field $\mathbb{F}_2$, i.e. elements are $0, 1, \alpha, \alpha + 1$. As above, the Galois group of this extension is $\{\iota, \rho\}$ and $a^2 = a$ for all $a$ in the base field $\mathbb{F}_2$. The minimal polynomial of $\alpha$ is

$$(x - \alpha)(x - \alpha^2) = (x - \alpha)(x - \alpha - 1) = x^2 + x + 1$$

using $\alpha^2 + \alpha + 1$ arithmetic.

Also we can think of $\mathbb{F}_{2^4}$ as an extension of $\mathbb{F}_2$: this $\mathbb{F}_{2^4}$ is $\mathbb{F}_2$ adjoin $\gamma$ where $\gamma$ is a root of $x^4+x+1$. This is a quartic extension whose elements are linear combinations of 1, $\gamma$, $\gamma^2$ and $\gamma^3$ with coefficients in the base field $\mathbb{F}_2$. The Galois group of this extension is $\{\iota, \rho, \rho^2, \rho^3\}$, and $a^2 = a$ for all elements $a$ of the base field. The minimal polynomial of $\gamma$ is

$$(x-\gamma)(x-\gamma^2)(x-\gamma^4)(x-\gamma^8) = (x-\gamma)(x-\gamma^2)(x-\gamma-1)(x-\gamma^2-1) = x^4+x+1$$

using $\gamma^4+\gamma+1$ arithmetic.

Third, we've seen how $\mathbb{F}_{2^2}$ sits as a subfield of $\mathbb{F}_{2^4}$: this $\mathbb{F}_{2^2}$ is $\{0, 1, \gamma^2+\gamma, \gamma^2+\gamma+1\}$ as above (when $\gamma$ was called $u$).

Now, how do we see $\mathbb{F}_{2^4}$ as an extension field not of $\mathbb{F}_2$ but rather of $\mathbb{F}_{2^2}$? If $\mathbb{F}_{2^2} = \mathbb{F}_2(\alpha)$, then we should be able to write $\mathbb{F}_{2^4}$ as $\mathbb{F}_2(\alpha)(\beta)$ for some $\beta$. This will be a quadratic extension of $\mathbb{F}_2(\alpha)$ so elements of this $\mathbb{F}_{2^4}$ will be linear combinations of 1 and $\beta$ with coefficients in $\mathbb{F}_2(\alpha)$.

We lack the minimal polynomial of $\beta$. We need this to do arithmetic in $\mathbb{F}_2(\alpha)(\beta)$: otherwise we can't simplify powers of $\beta$, in the way that $\alpha^2+\alpha+1=0$ allows us to simplify powers of $\alpha$. That is, we need $M_\beta(y)$ in order to write

$$\mathbb{F}_{2^4} = \mathbb{F}_2(\alpha)[y]/\langle M_\beta(y)\rangle$$

The Galois group of the extension $\mathbb{F}_{2^4}/\mathbb{F}_{2^2}$ is $\langle\rho^2\rangle = \{\iota, \rho^2\}$, since $y^4 = y$ on the base field $\mathbb{F}_{2^2}$.

We need to select a generator $\beta$ of $\mathbb{F}_{2^4}$. Since $x^4+x+1$ is a primitive polynomial, $\gamma$ is a generator of $\mathbb{F}_{2^4}$ and we can use $\gamma$'s polynomial and say that $\beta^4+\beta+1=0$. Then, consulting the log table in appendix H.3 for multiplication, we have

$$\begin{aligned}
M_\beta(y) &= \prod_{\sigma\in \mathrm{Gal}(\mathbb{F}_{2^4}/\mathbb{F}_{2^2})} (y-\sigma(\beta)) \\
&= (y-\beta)(y-\beta^4) \\
&= (y+\beta)(y+\beta^4) \text{ since } p=2 \\
&= (y+0010)(y+0011) \\
&= y^2+y+0110 \\
&= y^2+y+\beta^2+\beta
\end{aligned}$$

To get this as a polynomial with coefficients in $\mathbb{F}_{2^2}$, i.e. with coefficients in $\alpha$ rather than in $\beta$, use the unembedding technique above and observe (perhaps from the root chart) that $\beta^2+\beta$ is an element of the $\mathbb{F}_{2^2}$ inside $\mathbb{F}_{2^4}$. It is the 1st power of $\beta^2+\beta$, whose minimal polynomial is $y^2+y+1$. Over the standard $\mathbb{F}_{2^2}$, this factors as $(y-\alpha)(y-\alpha-1)$. Arbitrarily select the first root, $\alpha$, and take its 1st power to write

$$M_\beta(y) = y^2 + y + \alpha$$

Was it just good fortune that the coefficient $\beta^2 + \beta$ was an element of $\mathbb{F}_{2^2}$, allowing us to replace it with an $\alpha$? No; in fact, this is precisely what the Galois theory guarantees.

## 6.2 Double-compact notation

Now that we have this minimal polynomial for $\beta$, just as we used $\alpha^2 + \alpha + 1 = 0$ to simplify powers of $\alpha$ by $\alpha^2 = \alpha + 1$, here also we can simplify powers of $\beta$ by $\beta^2 = \beta + \alpha$. The resulting log table, after some pencil-and-paper work which I've omitted, is as follows. First I must introduce the **double compact notation**, of the form 11:10. The adjacent digits are assumed to be in $\alpha$, e.g. $11 = \alpha + 1$; the colon-delimited extended digits are in $\beta$, e.g. $1 : 1 = \beta + 1$. So, 11:10 would be $(\alpha + 1)\beta + \alpha = \alpha\beta + \beta + \alpha$.

| $k$ | $\gamma^k$ (compact) | $\gamma^k$ (full) | $\beta^k$ (compact) | $\beta^k$ (full) |
|---|---|---|---|---|
| 1 | 0010 | $\gamma$ | 01:00 | $\beta$ |
| 2 | 0100 | $\gamma^2$ | 01:10 | $\beta + \alpha$ |
| 3 | 1000 | $\gamma^3$ | 11:10 | $(\alpha + 1)\beta + \alpha$ |
| 4 | 0011 | $\gamma + 1$ | 01:01 | $\beta + 1$ |
| 5 | 0110 | $\gamma^2 + \gamma$ | 00:10 | $\alpha$ |
| 6 | 1100 | $\gamma^3 + \gamma^2$ | 10:00 | $\alpha\beta$ |
| 7 | 1011 | $\gamma^3 + \gamma + 1$ | 10:11 | $\alpha\beta + \alpha + 1$ |
| 8 | 0101 | $\gamma^2 + 1$ | 01:11 | $\beta + \alpha + 1$ |
| 9 | 1010 | $\gamma^3 + \gamma$ | 10:10 | $\alpha\beta + \alpha$ |
| 10 | 0111 | $\gamma^2 + \gamma + 1$ | 00:11 | $\alpha + 1$ |
| 11 | 1110 | $\gamma^3 + \gamma^2 + \gamma$ | 11:00 | $(\alpha + 1)\beta$ |
| 12 | 1111 | $\gamma^3 + \gamma^2 + \gamma + 1$ | 11:01 | $(\alpha + 1)\beta + 1$ |
| 13 | 1101 | $\gamma^3 + \gamma^2 + 1$ | 10:01 | $\alpha\beta + 1$ |
| 14 | 1001 | $\gamma^3 + 1$ | 11:11 | $(\alpha + 1)\beta + \alpha + 1$ |
| 15 | 0001 | $1$ | 00:01 | $1$ |

## 6.3 Second example for intermediate fields

Throughout much of this paper I've used finite fields with $p = 2$, $n = 4$: these are small enough that it's easy to write down all the field elements, yet large enough that there is some interesting behavior, namely, more than one monic irreducible (which doesn't happen e.g. with $p = 2$, $n = 2$), and an intermediate field (which doesn't happen e.g. with any $p$ for $n = 3$).

However, the $p = 2$, $n = 4$ case doesn't give quite as interesting a subfield diagram as it could. The next smallest more interesting case is $p = 2$, $n = 6$. Here I won't write down all the irreducibles: there are 9 of them, and I'll choose $x^6 + x + 1$ which happens to be primitive[1].

$$\begin{tikzcd}
 & \mathbb{F}_{2^6} = \mathbb{F}_2(\alpha)(\delta) = \mathbb{F}_2(\beta)(\varepsilon) = \mathbb{F}_2(\gamma),\ \gamma^6 + \gamma + 1 = 0 \arrow[dl, dash] \arrow[dr, dash] & \\
\mathbb{F}_{2^2} = \mathbb{F}_2(\alpha),\ \alpha^2 + \alpha + 1 = 0 \arrow[dr, dash] & & \mathbb{F}_{2^3} = \mathbb{F}_2(\beta),\ \beta^3 + \beta + 1 = 0 \arrow[dl, dash] \\
 & \mathbb{F}_2 &
\end{tikzcd}$$

$$\begin{tikzcd}
 & \langle \rho^6 = \iota \rangle \arrow[dl, dash] \arrow[dr, dash] & \\
\langle \rho^2 \rangle \arrow[dr, dash] & & \langle \rho^3 \rangle \arrow[dl, dash] \\
 & \langle \rho \rangle &
\end{tikzcd}$$

Here is a root chart for $\mathbb{F}_{2^6}$, but displaying, for brevity, only roots mod the single monic irreducible $x^6 + x + 1$ rather than mod all the sextics. (For $\mathbb{F}_{2^4}$, I tabulated roots mod all the quartics.) Logarithms with respect to generator $u = 000010$ are displayed in parentheses to the right of the field elements. This makes the root chart a log table as well (although entries aren't in a convenient order) so we can consult this for multiplication. Also I include root charts for the standard subfields $\mathbb{F}_{2^3}$ and $\mathbb{F}_{2^2}$.

```
Root chart for 1000011, g = 000010
Linears
  0000010: 000000( -)
  0000011: 000001( 0)
Quadratic
  0000111: 111011(21) 111010(42)
Cubics
  0001011: 001110(27) 010111(54) 011001(45)
  0001101: 011000( 9) 001111(18) 010110(36)
Sextics
  1000011: 000010( 1) 000100( 2) 010000( 4) 001100( 8) 010011(16) 001001(32)
  1001001: 000110( 7) 010100(14) 011100(28) 011111(56) 011010(49) 001011(35)
  1010111: 001000( 3) 000011( 6) 000101(12) 010001(24) 001101(48) 010010(33)
  1011011: 001010(13) 000111(26) 010101(52) 011101(41) 011110(19) 011011(38)
```

[1]You might suspect a pattern here: $x^2 + x + 1$, $x^3 + x + 1$, $x^4 + x + 1$ and $x^6 + x + 1$ are all irreducible and primitive in $\mathbb{F}_2[x]$. But don't be fooled: it is not the case that $x^n + x + 1$ is irreducible for all $n$, let alone primitive.

```
  1100001: 100101(31) 100001(62) 110001(61) 111101(59) 101110(55) 100111(47)
  1100111: 100000( 5) 110000(10) 111100(20) 101111(40) 100110(17) 100100(34)
  1101101: 100011(11) 110101(22) 101101(44) 100010(25) 110100(50) 101100(37)
  1110011: 101001(23) 110010(46) 111000(29) 111111(58) 101010(53) 110111(43)
  1110101: 101000(15) 110011(30) 111001(60) 111110(57) 101011(51) 110110(39)
```

Observe again the subfield/log criterion: the roots of the linears and the quadratic have logs divisible by $21 = (2^6-1)/(2^2-1)$; the roots of the linears and the cubics have logs divisible by $9 = (2^6-1)/(2^3-1)$.

```
Root chart for 1011, g = 010
Linears
  0010: 000(-)
  0011: 001(0)
Cubics
  1011: 010(1) 100(2) 110(4)
  1101: 011(3) 101(6) 111(5)
```

```
Root chart for 111, g = 10
Linears
  010: 00(-)
  011: 01(0)
Quadratic
  111: 10(1) 11(2)
```

$\mathbb{F}_{2^2}$ is an extension of $\mathbb{F}_2$. Likewise, $\mathbb{F}_{2^3}$ is an extension of $\mathbb{F}_2$. Also we can think of $\mathbb{F}_{2^6}$ as an extension of $\mathbb{F}_2$: this $\mathbb{F}_{2^6}$ is $\mathbb{F}_2$ adjoin $\gamma$ where $\gamma$ is a root of $x^6+x+1$. (In the root chart, $\gamma$ appears as 000010.) $\mathbb{F}_{2^6}$ is a sextic extension of $\mathbb{F}_2$ whose elements are linear combinations of 1, $\gamma$, $\gamma^2$, $\gamma^3$, $\gamma^4$ and $\gamma^5$ with coefficients in the base field $\mathbb{F}_2$. The Galois group of this extension is $\{\iota, \rho, \rho^2, \rho^3, \rho^4, \rho^5\}$ and $a^2 = a$ for all $a$ in the base field $\mathbb{F}_2$. The minimal polynomial of $\gamma$ is

$$\begin{aligned} & (x-\gamma)(x-\gamma^2)(x-\gamma^4)(x-\gamma^8)(x-\gamma^{16})(x-\gamma^{32}) \\ = \; & (x+000010)(x+000100)(x+010000)(x+001100)(x+010011)(x+001001) \\ = \; & x^6+x+1 \end{aligned}$$

using $\gamma^6+\gamma+1$ arithmetic.

Now, how do we see $\mathbb{F}_{2^6}$ as an extension field not of $\mathbb{F}_2$ but rather of $\mathbb{F}_{2^2}$ and $\mathbb{F}_{2^3}$? We should be able to write $\mathbb{F}_{2^6}$ as $\mathbb{F}_2(\alpha)(\delta)$ for some $\delta$. This will be a cubic extension of $\mathbb{F}_2(\alpha)$ so elements of this $\mathbb{F}_{2^6}$ will be linear combinations of 1, $\delta$ and $\delta^2$ with coefficients in $\mathbb{F}_2(\alpha)$. Likewise, we should be able to write $\mathbb{F}_{2^6}$ as $\mathbb{F}_2(\beta)(\varepsilon)$ for some $\varepsilon$. This will be a quadratic extension of $\mathbb{F}_2(\beta)$ so elements of this $\mathbb{F}_{2^6}$ will be linear combinations of 1 and $\varepsilon$ with coefficients in $\mathbb{F}_2(\beta)$.

We lack the minimal polynomials of $\delta$ and $\varepsilon$. We need them to simplify powers of $\delta$ and $\varepsilon$ so we can do arithmetic in

$$\mathbb{F}_{2^6} = \mathbb{F}_2(\alpha)[y]/\langle M_\delta(y)\rangle$$

and

$$\mathbb{F}_{2^6} = \mathbb{F}_2(\beta)[y]/\langle M_\varepsilon(y)\rangle$$

The Galois group of $\mathbb{F}_2(\alpha)(\delta)$ over $\mathbb{F}_2(\alpha)$ is $\langle\rho^2\rangle = \{\iota, \rho^2, \rho^4\}$, since $a^4 = a$ fixes the base field $\mathbb{F}_{2^2}$. Likewise, the Galois group of $\mathbb{F}_2(\beta)(\varepsilon)$ over $\mathbb{F}_2(\beta)$ is $\langle\rho^3\rangle = \{\iota, \rho^3\}$, since $a^8 = a$ fixes the base field $\mathbb{F}_{2^3}$.

To find the minimal polynomials of $\delta$ and $\varepsilon$, obtain a generator of the $\mathbb{F}_{2^6}$ defined by $\gamma$. Since $x^6 + x = 1$ is a primitive polynomial, $\gamma$ generates the multiplicative group. Again we appropriate $\gamma$'s polynomial for $\delta$ and $\varepsilon$, so $\delta^6 + \delta + 1 = 0$ and $\varepsilon^6 + \varepsilon + 1 = 0$. Since the Galois groups of $\mathbb{F}_{2^6}/\mathbb{F}_{2^2}$ and $\mathbb{F}_{2^6}/\mathbb{F}_{2^3}$ are $\langle\rho^2\rangle$ and $\langle\rho^3\rangle$, respectively, by definition the minimal polynomials are

$$\begin{aligned} M_\delta(y) &= (y-\delta)(y-\delta^4)(y-\delta^{16}) \\ M_\varepsilon(y) &= (y-\varepsilon)(y-\varepsilon^8) \end{aligned}$$

Using the logarithms included in the $\mathbb{F}_{2^6}$ root chart for multiplication, the former simplifies to

$$\begin{aligned} M_\delta(y) &= (y-\delta)(y-\delta^4)(y-\delta^{16}) \\ &= (y+\delta)(y+\delta^4)(y+\delta^{16}) \\ &= (y+000010)(y+010000)(y+010011) \\ &= (y+000010)(y^2+000011y+111100) \\ &= y^3+y^2+111010y+111011 \end{aligned}$$

With reference to the root chart for $\mathbb{F}_{2^6}$, 111010 is $\delta^{42}$ and 111011 is $\delta^{21} = (\delta^{21})^2$. These are roots of $x^2+x+1$ in $\mathbb{F}_{2^6}$, which from the root chart for $\mathbb{F}_{2^2}$ has corresponding roots $\alpha$ and $\alpha+1$. Thus, using the unembedding $\delta^{42} \mapsto \alpha$,

$$M_\delta(y) = y^3 + y^2 + \alpha y + \alpha + 1$$

Likewise,

$$\begin{aligned} M_\varepsilon(y) &= (y-\varepsilon)(y-\varepsilon^8) \\ &= (y+\varepsilon)(y+\varepsilon^8) \\ &= (y+000010)(y+001100) \\ &= y^2+001110y+011100 \end{aligned}$$

With reference to the root chart for $\mathbb{F}_{2^6}$, 001110 is $\varepsilon^{27}$ and 011000 is $\varepsilon^9$. The former, $\varepsilon^{27}$, is a root of $x^3+x+1$ in $\mathbb{F}_{2^6}$, which from the root chart for $\mathbb{F}_{2^3}$ has a corresponding root $\beta$. The other coefficient, $\varepsilon^9$, is $(\varepsilon^{27})^5$ which maps to $\beta^5 = \beta^2+\beta+1$. Thus, using the unembedding $\varepsilon^{27} \mapsto \beta$,

$$M_\varepsilon(y) = y^2 + \beta y + \beta^2 + \beta + 1$$

As a sanity check, one can verify by factorization (e.g. appendix F) that these two minimal polynomials are irreducible over their respective base fields and have $\delta$ and $\varepsilon$, respectively, as roots.

(Also note that one can obtain the minimal polynomials $M_\delta$ and $M_\varepsilon$ by factoring $y^6+y+1$ over $\mathbb{F}_{2^2}$ and $\mathbb{F}_{2^3}$, respectively, and selecting in each case the factors of which $\delta$ and $\varepsilon$, respectively, are roots. I've not pursued that method in this paper since it is more time-consuming on paper.)

Now that the minimal polynomials have been obtained, we can write a log table as above. Since there are 64 field elements, I'll show only a few rows.

| $k$ | $\delta^k$ (compact) | $\varepsilon^k$ (compact) | $\gamma^k$ (compact) |
|---|---|---|---|
| 1 | 01:00 | 001:000 | 000010 |
| 2 | 01:00:00 | 010:111 | 000100 |
| 3 | 01:10:11 | 011:101 | 001000 |
| 4 | 11:01:11 | 011:010 | 010000 |
| 5 | 10:10:10 | 100:010 | 100000 |
| 6 | 01:01 | 001:001 | 000011 |
| 7 | 01:01:00 | 011:111 | 000110 |
| 8 | 10:11 | 001:010 | 001100 |
| 9 | 10:11:00 | 111 | 011000 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| 21 | 11 | 101:100 | 111011 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| 27 | 11:11 | 010 | 001110 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| 42 | 10 | 101:101 | 111010 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| 61 | 01:11:00 | 111:001 | 110001 |
| 62 | 10:10:11 | 100:011 | 100001 |
| 63 | 01 | 001 | 000001 |

# Part III

# Linear feedback shift registers

# Chapter 7

# LFSRs and linear transformations

Linear feedback shift registers are perhaps the single most important application of finite fields. In this chapter we'll see what they are, look at several different configurations, and discover that they are all special cases of a general situation.

The standard reference for LFSRs is Golomb [5]. Also see Lidl and Niederreiter [10], and in Schneier [13]. At the same time, this chapter serves to clarify some concepts as presented in Schneier [13].

## 7.1 Homogeneous linear recurrence relations

Anyone who has taken discrete mathematics is familiar with *recurrence relations*. Often, the first example presented is the *Fibonacci sequence*, given by

$$\begin{aligned} s_0 &= 0 \\ s_1 &= 1 \\ s_{k+2} &= s_{k+1} + s_k, \ k = 0, 1, 2, \dots \end{aligned}$$

By applying these rules we get the sequence 0, 1, 1, 2, 3, 5, 8, etc.

The idea is to define a sequence (of integers, reals, finite-field elements, etc.) in terms of previous elements. Terminology:

- If the value of an element is a function of the $n$ values before it ($n = 2$ for the Fibonacci sequence), then we say we have an **nth-order recurrence relation**.

- This function may be called the **recurrence rule**, since it's the rule which tells us how to produce the next element of the sequence.
- For an $n$th-order recurrence relation, to avoid indexing into negative $k$ values where the sequence is undefined, we require $n$ **initial conditions**, $s_0$ through $s_{n-1}$ — in this example, $s_0 = 0$ and $s_1 = 1$. It's important to note that the initial conditions are part of what defines the sequence — if you use the Fibonacci sequence's recurrence rule but use different initial conditions, that's not the Fibonnaci sequence.
- If the recurrence rule is a linear function (as is the case for the Fibonacci sequence; $s_k = s_{k-1}^2$ would be a non-linear 1st-order recurrence relation on the integers[1]), then we say we have a **linear recurrence relation**.
- If there is a constant term in the recurrence rule, e.g. $s_k = s_{k-2} + s_{k-1} + 3$, then we say we have an **inhomogeneous linear recurrence relation**; otherwise we say we have a **homogeneous linear recurrence relation**.

In this paper I'll confine myself to homogeneous linear recurrence relations. In general, we have

$$\begin{aligned} s_{k+n} &= c_{n-1}s_{k+n-1} + c_{n-2}s_{k+n-2} + \ldots + c_1 s_{k+1} + c_0 s_k \\ &= \sum_{i=0}^{n-1} c_i s_{k+i} \end{aligned}$$

## 7.2 Matrices for homogeneous linear recurrence relations

We can define recurrence relations over any ring, group, etc. — any set of elements with at least one mathematical operation. Here we'll look specifically at recurrence relations over the field $\mathbb{F}_q = \mathbb{F}_{p^n}$.

Here is an example, a 4th-order homogeneous linear recurrence relation, taken over $\mathbb{F}_2$:

$$s_{k+4} = s_{k-3} + s_{k-4}$$

Now, thinking ahead to where I'm going, I can write a **state vector**

$$\mathbf{s}_k = (s_{k+3}, s_{k+2}, s_{k+1}, s_k)^t$$

[1]Although it would be linear over $\mathbb{F}_2$, since $a^2 = a$ for all $a$ in $\mathbb{F}_2$.

with the obvious generalization to $n$th-order recurrence relations. The $k$th state vector, $\mathbf{s}_k$, consists simply of the $n$ values needed to produce the next element of the sequence. Given that, I can write the following matrix equation to get the next state vector:

$$\begin{pmatrix} s_{k+4} \\ s_{k+3} \\ s_{k+2} \\ s_{k+1} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} s_{k+3} \\ s_{k+2} \\ s_{k+1} \\ s_k \end{pmatrix}$$

If you write out the matrix multiplication, all this says is that

$$s_{k+4} = s_{k+1} + s_k$$

which is true, along with

$$s_{k+3} = s_{k+3}, \; s_{k+2} = s_{k+2}, \; s_{k+1} = s_{k+1}$$

which are undeniable. Writing the above matrix as $T$, I have

$$\mathbf{s}_{k+1} = T\mathbf{s}_k$$

As you can easily see, given an $n$th-order linear recurrence relation

$$s_{k+n} = c_{n-1}s_{k+n-1} + c_{n-2}s_{k+n-2} + \ldots + c_1 s_{k+1} + c_0 s_k$$

the $T$ matrix will be

$$\begin{pmatrix} c_{n-1} & c_{n-2} & \cdots & c_1 & c_0 \\ 1 & 0 & & 0 & 0 \\ 0 & 1 & & 0 & 0 \\ \vdots & & \ddots & & \\ 0 & 0 & & 1 & 0 \end{pmatrix}$$

Why would I write such a matrix equation? Because we now have a **linear transformation** from the **n-dimensional vector space** $(\mathbb{F}_q)^n$ to itself. And quite a bit is known about linear transformations.

## 7.3 Registers

When you're standing on the ground watching a train go by, of course the ground is motionless while the train whizzes past you. But if you're on the train, it seems to stay still while the ground rushes past in the other direction. Looking at a recurrence relation is like watching a train: we stand on the integers while the recurrence relation chugs out one sequence element after another. Looking at the state vectors is like being on the train: we keep the $n$ previous values next to us, and they burble and change while the subscripts rush along underneath.

In electronic circuits, which is where this all started (see also [2]), the cars in the train are called a **register**. Let's take another look at that matrix multiplication, which transforms a state vector at one time tick into the state vector at the next time tick. Let := stand for **assignment**. Letting the train stand still, I'll let $v_0$ through $v_{n-1}$ stand for the values in the register at a given time step.

We have

$$\begin{aligned} v_3 &:= v_1 + v_0 \\ v_2 &:= v_3 \\ v_1 &:= v_2 \\ v_0 &:= v_1 \end{aligned}$$

or (please pardon the corny ASCII graphics):

```
   o-------------o-(+)--o
   |             |      |
   |             ^      ^
   v             |      |
o------o------o--|---o--|---o
|  v3 --> v2 --> v1 --> v0  | ---> output
o------o------o-------o------o
```

The term *linear feedback shift register* is very clear here: bits in the register are shifted one position to the right at each time step; the vacancy at the left is filled by values fed back into the register, using a linear function.

The entire contents of the register (the state vector) is the **internal state**, namely, all the $n$ history values needed for the recurrence relation, but only the shifted-out value is the **output**. Mathematically, the state is in $(\mathbb{F}_q)^n$ but the output is in $\mathbb{F}_q$. In particular, for this example where $q = 2$, we get only one output bit per time tick. We'll discuss below what to do about this.

## 7.4 Periodicity of recurrence relations over $\mathbb{F}_q$

Suppose you have a function $f$ from a finite set $S$ to itself, then **iterate** that function. E.g. given $s$ in $S$, compute $f(s)$, then $f(f(s))$, etc. Since there are only finitely many elements in the set, you can't keep getting different iterates[2], so some $j$th iterate will be equal to some $i$th iterate. At that point, the function will necessarily start to repeat.

Now, the first few values don't ever have to appear again. For example, square an odd integer mod 10. Say you start with 3; its square mod 10 is 9, whose square is 1, and all subsequent squares are 1. I.e. you get the sequence $3, 9, 1, 1, 1, \ldots$. The 3 and the 9 don't ever reappear, but the sequence is **ultimately periodic** with period 1. By analogy with the shape of the Greek letter $\rho$, we sometimes call the start-up phase (e.g. $3, 9$) the **tail** and the periodic phase (e.g. $1, 1, 1, \ldots$) the **loop** or the **cycle**.

Now, our linear transformations (the non-linear ones too, for that matter) are functions from the finite set $(\mathbb{F}_q)^n$ to itself, so *LFSRs always repeat.* In engineering practice, one typically wants:

- The longest possible period, or **maximal period**, for a given amount of history (i.e. $n$).
- The maximal number of states reachable from any other, i.e. we should be able to run through as many of the possible states as we can;
- The minimal number of **lock-up** states, i.e. state vectors that map to themselves.

We'll see below how easy these goals are to achieve, using some of the finite-field techniques from earlier in this paper. (To avoid being coy, I'll say that the zero vector will be a lock-up state, but we'll always, for any $q$ and $n$, be able to construct an LFSR of period $q^n - 1$ which cycles through all of the $q^n - 1$ non-zero states, regardless of initial state.) But first, I want to look at a slightly different LFSR.

## 7.5 Powers of $u$ revisited

Let's take another look at the RER method for multiplying by $u$, as described in section 3.8. Suppose for the sake of example that we're working in $\mathbb{F}_{2^4}$, with an arbitrary finite-field element which is

$$a_3u^3 + a_2u^2 + a_1u + a_0$$

and suppose the polynomial $r(x)$ which we use to define our finite-field arithmetic is

$$x^4 + x + 1$$

When we multiply by $u$, we get

$$a_3u^4 + a_2u^3 + a_1u^2 + a_0u$$

[2]Computer scientists call this the *pigeonhole principle.*

But we want to reduce mod $r$. Since $u^4 = -u - 1$, we get

$$a_2u^3 + a_1u^2 + (a_0 - a_3)u - a_3$$

Graphically,

| | | | | | |
|---|---|---|---|---|---|
| | $a_3$ | $a_2$ | $a_1$ | $a_0$ | Input |
| $a_3$ | $a_2$ | $a_1$ | $a_0$ | 0 | Multiplied by $u$ |
| | $a_2$ | $a_1$ | $a_0 - a_3$ | $-a_3$ | Reduced mod $r$, using RER |

That is, letting $b$'s represent the coefficients of $u \cdot a$,

$$\begin{aligned} b_3 &= a_2 \\ b_2 &= a_1 \\ b_1 &= a_0 - a_3 \\ b_0 &= -a_3 \end{aligned}$$

Now, this can be represented by a matrix (as you might expect, since multiplication by $u$ is certainly a linear transformation on $\mathbb{F}_{p^n}$). Recalling that plus is the same as minus when $p = 2$, we have

$$\begin{pmatrix} b_3 \\ b_2 \\ b_1 \\ b_0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix}$$

In the general case, it's easy to see that if

$$r(x) = x^n + r_{n-1}x^{n-1} + r_{n-2}x^{n-2} + \ldots + r_1x + r_0$$

then the multiply-by-$u$ matrix looks like

$$\begin{pmatrix} -r_{n-1} & 1 & 0 & & 0 & 0 \\ -r_{n-2} & 0 & 1 & & 0 & 0 \\ \vdots & & & \ddots & & \\ -r_1 & 0 & 0 & & 0 & 1 \\ -r_0 & 0 & 0 & & 0 & 0 \end{pmatrix}$$

We could implement an LFSR as follows: Again let $v$'s stand for register values, and let

$$\begin{aligned} v_3 &:= v_2 \\ v_2 &:= v_1 \\ v_1 &:= v_0 - v_3 \\ v_0 &:= -v_3 \end{aligned}$$

It would look like this:

```
   o-------------o------o
   |             |      |
   ^             v      v
   |             |      |
o--|---o------o-(+)--o-(+)--o
|  v3 <-- v2 <-- v1 <-- v0  |
o------o------o------o------o
```

That is, it looks just like the previous LFSR, *except with the arrows reversed.*

## 7.6 LFSR configurations

Now we have two ways to form LFSRs. In the first case, we started with a recurrence relation over $\mathbb{F}_q$, then created vectors over $(\mathbb{F}_q)^n$. In the second case, we looked at arithmetic in $\mathbb{F}_{q^n}$, which we can view as a vector space $(\mathbb{F}_q)^n$ over $\mathbb{F}_q$.

The first LFSR we looked at arose naturally from recurrence relations such as the Fibonacci sequence. This LFSR is called a **Fibonacci configuration**. The second LFSR we looked at arose naturally from multiplication in a finite, or Galois, field. This LFSR is called a **Galois configuration**. (To be quite picky, Galois-configuration LFSRs are typically presented as a left-to-right mirror image of what I've shown here, so I should say I'm using a *reflected* Galois configuration.)

xxx different functions, different internal state – but same *output*?!?

## 7.7 Companion matrices

xxx note that LFSRs can be defined over Fq, where (practically) q is a power of 2. mention TSRs just for fun ... ref to Tsaban and Vishne "very readable paper".

xxx no crappy line breaks in ASCII art!

# Part IV

# Software implementation

# Chapter 8

# Computing with finite fields in GP

If you haven't guessed by now, even though all the methods presented in this paper can be done by hand, it quickly gets tedious even with the assistance of tables. It can become more entertaining, however, if you let a machine do the repetitive work (but only after you also know how to do it yourself, with the machine turned off!). The computational algebraic number theory package GP [6] can be used to compute in finite fields. Also, of course, you can use Mathematica, Maple, GAP and probably several other packages. Here I'll talk about GP. In chapter 9 I'll discuss implementation of $p = 2$ in software, since that is an important special case.

Going the other way, if you're looking at an implementation of a finite-field algorithm written by someone else — whether in C or assembler (software), Verilog or VHDL (hardware), etc. — the implemenation may look not at all familiar. Chapter 9 will explain some of those snippets of C code.

## 8.1 $\mathbb{F}_p$ in GP

As described in 3.5 on page 22, we use the equivalence $\mathbb{F}_p[x]/\langle r(x)\rangle \cong \mathbb{F}_p(u)$. Throughout this paper I've used the latter; GP uses the former. I'll show how to convert notations using GP.

Modular arithmetic (modulo integers or polynomials) is implemented in GP using `Mod`. The first argument is a ring element; the second is a modulus. The result is the canonical representative of the equivalence class of the first argument mod the second. Use `lift` to lift back to the containing ring.

```
? a=Mod(2,11);
? a
%1 = Mod(2, 11)

? lift(a)
%2 = 2

? lift(a^4)
%3 = 5

? lift(a^5)
%4 = 10

? lift(a^10)
%5 = 1

? lift(1/a)
%6 = 6

? lift(a^-6)
%7 = 5
```

## 8.2 $\mathbb{F}_{p^n}$ in GP

Specify the polynomial $r(x)$ in $\mathbb{F}_p[x]$ which will be used to define the finite-field arithmetic. Multiply by `Mod(1,p)` to apply the ring homomorphism from $\mathbb{Z}[x]$ to $\mathbb{F}_p[x]$, or use `Mod` on each coefficient to write the polynomial directly in $\mathbb{F}_p[x]$. Then use `lift` to remove clutter for printing.

```
? rpoly = (x^6 + x + 1) * Mod(1,2);
\\ Could also write: rpoly = Mod(1,2) * x^6 + Mod(1,2) * x + Mod(1,2)

? polisirreducible(rpoly)
%1 = 1

? rpoly
%2 = Mod(1, 2)*x^6 + Mod(1, 2)*x + Mod(1, 2)

? lift(rpoly)
%3 = x^6 + x + 1
```

Next, create an element of the finite field as follows: use `Mod` again to signify reduction mod $r$. Use `lift` twice for printing: once to get past the polynomial mod, and again to lift the coefficients. Then use `subst` to convert to $u$ notation. (This is not only due to personal preference: it also makes the output much easier to read.)

```
? a = Mod(Mod(1,2)*x^3+x, rpoly)
%4 = Mod(Mod(1, 2)*x^3 + Mod(1, 2)*x, Mod(1, 2)*x^6 + Mod(1, 2)*x + Mod(1, 2))

? uprint(f) = subst(lift(lift(f)), x, u);

? uprint(a)
%5 = u^3 + u

? uprint(a^2)
%6 = u^2 + u + 1

? uprint(a^3)
%7 = u^5 + u^4 + u^2 + u

? uprint(1/a)
%8 = u^5 + u^4 + u^2

? uprint(a^63)
%9 = 1
```

## 8.3 Multiple extensions in GP

Note: The presentation in this section uses the notation of section 6.3 on page 49.

To review, we have the following sequence of rings and fields when we construct a multiple extension:

- The integer ring $\mathbb{Z}$
- The field $\mathbb{F}_p = \mathbb{Z}/\langle p \rangle$
- The polynomial ring $\mathbb{F}_p[x]$
- The field $\mathbb{F}_p[x]/\langle r_1(x) \rangle = \mathbb{F}_p(\alpha)$ where $\alpha$ is a root of $r_1(x)$
- The polynomial ring $\mathbb{F}_p(\alpha)[y]$

- The field $\mathbb{F}_p(\alpha)[y]/\langle r_2(y)\rangle = \mathbb{F}_p(\alpha)(\beta)$ where $\beta$ is a root of $r_2(y)$

A mod operation is used each time we pass from a ring to a field, where the modulus is irreducible (e.g. $p$, $r_1(x)$, $r_2(y)$). Thus, for double extensions there will be three mods.

Also note that GP (as of this writing, namely, version 2.2.6) does not implement multivariate polynomials per se, of the form $F[x,y]$. Rather, it implements them in the form $F[x][y]$: polynomials with coefficients in $y$, whose coefficients are polynomials in $x$. To tell GP which variable comes first, type e.g. `x;y;.` or `y;x;.` (However, GP always defines `x` as a variable at startup.)

```
d;a; \\ delta and alpha
e;b; \\ epsilon and beta
c;   \\ gamma

p  = 2;

\\ This is the F_{2^6} defined by gamma, with a generator.
rc = Mod(1, p) * (c^6 + c + 1);
gc = Mod(c, rc);

\\ This is the F_{2^6} defined by alpha and delta, with a generator.
ra = Mod(1, p) * (a^2 + a + 1);
rb = Mod(a, ra) * (d^3 + d^2 + a*d + a+1);
gd = Mod(d, rb);

\\ This is the F_{2^6} defined by beta and epsilon, with a generator.
rb = Mod(1, p) * (b^3 + b + 1);
re = Mod(b, rb) * (e^2 + b*e + b^2+b+1);
ge = Mod(e, re);

for (i=0,63,print(i, " ", lift(lift(lift(gc^i)))))
print("");
for (i=0,63,print(i, " ", lift(lift(lift(gd^i)))))
print("");
for (i=0,63,print(i, " ", lift(lift(lift(ge^i)))))
```

Partial output:

```
0 1
1 c
```

```
2 c^2
3 c^3
...
61 c^5 + c^4 + 1
62 c^5 + 1
63 1

0 1
1 d
2 d^2
3 d^2 + a*d + (a + 1)
...
61 d^2 + (a + 1)*d
62 a*d^2 + a*d + (a + 1)
63 1

0 1
1 e
2 b*e + (b^2 + b + 1)
3 (b + 1)*e + (b^2 + 1)
...
61 (b^2 + b + 1)*e + 1
62 b^2*e + (b + 1)
63 1
```

# Chapter 9

# $p = 2$ in C

A main purpose of this paper is to connect mathematical theory of finite fields with engineering practice. Sometimes we use software to automate tasks which we find repetitive. Computer algebra systems, such as GP, do well at this for general $\mathbb{F}_{p^n}$. However, we can do further optimizations in the $p = 2$ case, since operations may be done in binary and may take advantage of some hardware parallelism (as we'll see below). For this reason, the $p = 2$ case is the most important in applications (e.g. cryptography and error correction).

Finite-field algorithms can implemented in software (say in C or assembler) or in electronic circuitry (e.g. in Verilog, VHDL or Handel-C). It can be disconcerting to see these low-level bit operations and not have any idea of how it relates to the mathematical theory. This section is a little different from the GP section: when I discussed GP, I talked about what *you* can do to implement finite-field arithmetic. For this section, I want to talk not only about how you can implement finite-field arithmetic, but also about how *other people* implement it. In particular, I want explain some commonly encountered paradigms, in order to shed some light on some of those mysterious circuit diagrams and snippets of C code you see in the literature or on the job.

Of course, syntaxes and applications vary between programming languages, but regardless of the implementation, you will see the same kinds of XOR and shift operations. I'll choose to give my examples in C: C is a common implementation language, it is (mostly) processor independent, and (unlike hardware design languages) it is easy to experiment with, without needing special hardware or tools. I'll assume you are able to compile and run a hello-world program in C, that you can use the basic I/O facilities of the `stdio` library, that you can write and call a function, etc. However, I won't assume you're a C guru, and in particular I will not assume knowledge of machine arithmetic.

## 9.1 Hexadecimal notation

First of all, we need to get familiar with **hexadecimal notation**. The Verilog and VHDL languages let you type your values in binary, if you wish. C does not, so you need to know hex to be able to *write* C code. Also, hexadecimal is the notation of choice (even outside of C), so you need to know hex to be able to *read* the C code as well.

Hexadecimal notation is simply base-16 arithmetic. The first 10 digits are 0-9; the remaining 6 (corresponding to 10-15) are a-f[1]. We use the 0x prefix[2] to indicate hexadecimal values, e.g. 0x13 in hex is 19 in decimal.

The nice thing about base 16 is that every hex digit (or *nybble*) corresponds to 4 bits[3]. Thus it is easy to translate back and forth between binary and hex. Furthermore, this enables a compact notation which makes numbers easier to remember[4]. For example, is it easier to remember this:

100000100110000010001110110110111

or this:

0x104c11db7?

Personally, I'd choose the latter.

Here is a table. If you're going to be implementing bit-manipulation algorithms, it would be best to know it by heart.

[1]Or, A-F depending on personal preference. C compilers accept both.

[2]This is the convention in C. In some assemblers, and in much literature from the Intel corporation, you'll see e.g. 23h in place of 0x23. Some other assemblers use a dollar sign, e.g. $23.

[3]*Octal*, or base 8, was common up through the 1970s or so. Each octal digit, 0-7, represents 3 bits. A word size of 36 bits was common on older IBM mainframes, and 3 goes evenly into 36. Today, processor word sizes are always a power of two, e.g. 8, 16, 32 or 64 bits, and bytes are always 8 bits, so base 16 is more natural.

[4]Also enabling engineers to invent creative test patterns for software and hardware designs, e.g. 0xdeadbeef, 0xbadd1dea, 0xdeaddadd8badf00d, etc. etc.

| Hex | Decimal | Binary |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| a | 10 | 1010 |
| b | 11 | 1011 |
| c | 12 | 1100 |
| d | 13 | 1101 |
| e | 14 | 1110 |
| f | 15 | 1111 |

When you see multi-digit hexadecimal numbers, simply translate them one nybble at a time. E.g. 0x14e becomes 0001 0100 1110. Likewise, convert from binary to hex four bits at a time, but be sure to group into foursomes *starting from the right.* E.g. 0101101 is 010 1101 which is 0x2d.

(To convert back and forth between binary and decimal, or between hex and decimal, is not as straightforward. Use a pencil and paper, use a scientific calculator, or do it in your head. For this document, we won't need to convert between hex and decimal notation so it's a non-issue.)

## 9.2 I/O

When you type in a *literal* in your C code, you must prefix it with 0x so the compiler can distinguish it from decimal. For example:

```
unsigned x = 0x13;
```

When you read in a string or a file, you can omit the 0x prefix if your code is expecting hexadecimal input: the `scanf` (scan standard input), `sscanf` (scan a string) and `fscanf` (scan from a file stream) functions use `%x` to specify hex input. For example:

```
char * string = "13";
unsigned x;
```

```
    ...
    if (sscanf(string, "%x", &x) != 1)
            printf("Couldn't scan input \"%s\" as hex.\n", string);
```

When you print a hex value, again use `%x`. Additionally, use a preceding digit to indicate field width (e.g. if you are printing columnar data which you want to align nicely), and optionally precede that with a zero to indicate zero fill. For example:

```
    unsigned a = 0x2, r = 0x13;
    printf("<<%x>> <<%x>>\n", a, r);       Results in <<2>> <<13>>
    printf("<<%4x>> <<%4x>>\n", a, r);     Results in <<   2>> <<  13>>
    printf("<<%04x>> <<%04x>>\n", a, r);   Results in <<0002>> <<0013>>
```

## 9.3 Word sizes

Back in the day when C was invented, it was considered important that the size of C's integral types not need to be the same everywhere, but rather match a given processor's word size. Decades later, that opinion is not always held. Nonetheless, while the ANSI C specification says that a `char` shall be one byte (always), for the rest it only requires that `sizeof(short)` $\leq$ `sizeof(int)` $\leq$ `sizeof(long)`.

Having said that, I can also say that the following is true:

- For every machine I've worked on in the last decade, a `short` is 2 bytes and an `int` is 4.
- For Windows 3.1 and earlier, an `int` was 2 bytes. But Windows 3.1 is fast becoming history. For Windows 9x and above, an `int` is 4 bytes.
- For any 32-bit Unix or Linux system, an `int` is 4 bytes.
- A `long` is usually 4 bytes, with the exception of some 64-bit systems where it may be 8 bytes.
- The `long long` type is not standard, but is very common and is typically implemented by the GNU C compiler (GCC). When present, it is always (in my experience) 8 bytes.
- The `sizeof` operator will always tell you the sizes of the various data types for your system.

What this means is that in most of the previous decade, and in the current decade, on everyday PC and workstation hardware (which nowadays typically have 32-bit processors), an `int` is 4 bytes. I tell you this not to encourage sloppiness, but rather to clue you to some of the assumptions you will see other people make.

If you are writing code on a system for which you know the data sizes used by your compiler, you may wish to simply use `unsigned` for your polynomials. (This is shorthand for `unsigned int`; signed vs. unsigned will be explained below). If you wish to write portable code (e.g. something your peers can safely use on their Alphas or Sparc 64's, something which you can safely re-use on your next project 5 years from now, something which won't break when you port it from your 32-bit PC test environment to the 8-bit embedded processor on your production system, etc.), your best bet is to make a header file something like the following:

```
// This file is named mytypes.h
#ifndef MYTYPES_H
#define MYTYPES_H // Protect against multiple inclusions
typedef unsigned char      int8u;
typedef          char      int8s;
typedef unsigned short     int16u;
typedef          short     int16s;
typedef unsigned int       int32u;
typedef          int       int32s;
typedef unsigned long long int64u;
typedef          long long int64s;
#endif // MYTYPES_H
```

Then, (a) include this header file in all your sources, using `#include "mytypes.h"`; (b) always use `int32u` etc. instead of plain C types; (c) when you port your code to some other environment, you'll only need to change that one header file.

In this document, I'll always use `unsigned`.

A final note about the longer and shorter data sizes: compilers *do* take care of them for you. A common misconception among hardware folks is that processors can only load and store in multiples of their word size, where the word size is defined to be the number of bits in CPU registers. This is simply not true. A 32-bit general-purpose processor has 8-bit, 16-bit and 32-bit load and store operations. A 16-bit processor has 8-bit and 16-bit load and store operations. A 64-bit processor has 8-, 16-, 32- and 64-bit load and store operations. For data types larger than the processor's word size, the compiler will issue as many load/store operations as necessary. E.g. if you have a 32-bit processor, you can use 64-bit `long long` values and rest assured that the compiler will generate a pair of 32-bit load operations to read one of these variables from memory, and a pair of 32-bit store operations to write them to memory. Likewise, the compiler will take care of the carry-ins and carry-outs across machine words needed to add, subtract, multiply and divide data types longer than the CPU word size (e.g. 32-bit `long` on a 16-bit processor, or 64-bit `long long` on a 32-bit processor).

## 9.4 Signed vs. unsigned

Unlike Java and Fortran, the C language provides signed and unsigned versions of integer types. Addition and subtraction produce the same results, bit for bit, regardless of signedness. The key differences in arithmetic are (1) integer division and (2) right-shift operations. The details of both, and twos' complement arithmetic in general, are simple enough but are beyond the scope of this paper (see e.g. the excellent treatment in chapter 4 of Hennessy and Patterson [8]). For the current discussion, in which we will do mainly bitwise operations and left shifts, the difference is not too important. Nonetheless, I'll use unsigned arithmetic.

## 9.5 Representation of integers

As mentioned above, the C language provides several integral types. For the sake of discussion, I'll assume 32-bit words and use type `unsigned`. The zero integer is represented with all bits zero, i.e. 0x00000000. The value 1 is 0x00000001, where the right-hand side[5] is the **least significant bit**. The left-hand side is the **most significant bit**. We count in the obvious way in base two.

Notice that 32-bit unsigned integers thus have a range from 0 to $2^{32} - 1 = 4,294,967,295$; 16-bit unsigned integers range from 0 to $2^{16} - 1 = 65,535$; 8-bit unsigned integers range from 0 to $2^8 - 1 = 255$.

## 9.6 Integer arithmetic

There's a lot to say about computer integer arithmetic, most of it not relevant to us (again, see chapter 4 of Hennessey and Patterson [8] for complete information). However, I do want to point out, for use below, what addition looks like. Suppose you want to add 13 and 14. In binary, these are 1101 and 1110. The addition looks like:

```
  1101
+ 1110
------
 11011
```

which is 27 in decimal and checks out. Note in particular how you do this: just as in elementary school (but now with base 2, not base 10), add columns from right to left,

[5]We write it this way regardless of the CPU's byte endianness, i.e. byte order within larger integral types. Nothing being discussed in this paper is endianness-dependent.

carrying out and carrying in if necessary. Note in particular that a given bit in the output can depend on *all* the input bits above and to the right.

I won't discuss machine arithmetic for subtraction, multiplication and division, although you can probably guess that they can be implemented using elementary-school methods. However, I will point out the following properties of integer arithmetic in C, some of which often come as a surprise to engineers, and at least one of which is distasteful to mathematicians:

- All arithmetic is performed modulo $2^n$ where $n$ is the number of bits of the data type being operated on[6], i.e. 8, 16, 32 or 64.

  For addition, this may not be too much of a surprise. However, when you multiply two $n$-bit integers, the full product can be as much as $2n$ bits. Moreover, some CPU architectures (e.g. MIPS) provide an instruction which takes two 32-bit operands and produces a 64-bit product. This is accessible from MIPS assembler, but not directly from C. Hardware people are often surprised by this discarding of data.

  Most importantly, C provides absolutely no automatic exception handling for overflow cases (although CPUs generally do raise an exception at the hardware level when a divide by zero is attempted). If you are concerned about integer overflow, you must check for it[7].

- Division of two integers produces another integer, not a rational or a float. Using Euclid's algorithm, given $a$ and $b$ we write $a = qb + r$. The C statement `q = a / b` produces precisely that quotient; the C statement `r = a % b` produces the remainder. (C does not provide rational arithmetic, although you can easily implement your own rational data structure and corresponding functions. And if you want a floating-point quotient, unlike in Perl, you must first cast to `float` or `double`.)

  For example, in C, 7/2 is 3.

- Using Euclid's division algorithm, we generally take $0 \leq r < b$. However, C's mod operator does not do this. If $a < 0$, then $r < 0$. This is mathematically repugnant, but someone felt it important enough to design it into the language. For example, we think of 8 mod 5 as being 3, just as $-12$ mod 5 is 3. But in C, the former is 3 and the latter is -2.

  To produce a mathematically correct mod, use C's mod, then add $b$ if the result is negative. For example:

  ```
  static __inline__ int posmod(int a, int b)
  {
  ```

[6]I won't discuss mixing data types, e.g. what happens when you multiply a `short` times a `long` and storing the result in an `int`.

[7]Note that IEEE floating-point arithmetic is completely different in this regard: `Inf` is used for overflow situations, `NaN` for 0/0 and other oddities.

```
        int r = a % b;
        if (r < 0)
                r += b;
        return r;
}
```

## 9.7 Bit numbering in C

In C, the least significant bit is numbered 0; the most significant bit is numbered $n-1$ for $n$-bit integers[8]. The following idioms are often seen:

```
unsigned x = 1 << 5;
```

This results in 0x20, which is $2^5$. In general, `1<<n` is $2^n$ although please remember that we are working with fixed word sizes[9], so don't expect `1<<397` to be $2^{397}$.

Here is a partial table:

| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| `1<<n` | 0x01 | 0x02 | 0x04 | 0x08 | 0x10 | 0x20 | 0x40 | 0x80 |

## 9.8 Representation of polynomials

Remember that we are working with $p = 2$, so all polynomial coefficients are 0 or 1. A polynomial $\sum_{i=0}^{n} a_i x^i$ is represented by setting the $i$th bit to $a_i$. For example,

$$x^4 + x + 1 = 10011 = 0x13$$

Finite-field elements are represented no differently. As with the compact notation introduced in section 3.3 (which was really me writing in binary), the $x$ or the $u$ disappears and is inferred by context. So $u^2 + u$ would be 0110, or 0x06.

[8]This has absolutely nothing to with hardware bit ordering at the schematic level (PowerPC calls the MSB the 0 bit), which is invisible to software.

[9]GP, along with most computer algebra systems, implements arbitrary precision arithmetic. In this section, though, I'm focusing on low-level implementations such as are encountered in the engineering world, where fixed word sizes are the norm.

## 9.9 Addition using XOR

Addition in $\mathbb{F}_2$ is simply addition mod 2. Here is an addition table:

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

The logic people (see also [7]) have another name for this function from $\mathbb{F}_2 \times \mathbb{F}_2$ to $\mathbb{F}_2$: the **exclusive OR**. If we put on our logic hats and think of 0 as corresponding to **false** and 1 as corresponding to **true**, then the result (the sum) of two inputs is true if one *or* the other is true, but not both (hence the *exclusive* part).

What about addition of polynomials? Just as before, we add componentwise, with no carries. For example, suppose we want to compute the sum of $x^3 + x^2 + 1$ and $x^3 + x^2 + x$. In binary, these are 1101 and 1110. The addition looks like:

```
  1101
+ 1110
------
  0011
```

which is $x + 1$ and checks out. Note that this is like integer addition, but *without the carries*. In particular, a given bit in the output depends *only* on the input bits above it. Thus, XORing an integer is a **bitwise operation**: each bit in the output depends only on the corresponding bits in the input.

All processors implement XOR instructions. This is a single instruction, which completes in a single clock cycle or so[10], and is very fast, as compared to, say, integer multiplication or division. At the hardware level, it requires very simple circuitry — again, in contrast to integer multiplication or division. In particular, note that for an $n$-bit processor we get $n$ bit additions all at the same time. This is why finite-field arithmetic with $p = 2$ is so efficient. In a hardware design, one is not limited to 32-bit or 64-bit words, etc.: One can declare very wide bit arrays and have all those XORs going on in parallel.

In C, this is spelled very simply. The XOR operator is a single caret. For example, let's compute the sum of $x^3 + x^2 + 1$ and $x^3 + x^2 + x$. In hex, these are 0xd and 0xe. The addition looks like:

```
        unsigned a = 0xd;
        unsigned b = 0xe;
        unsigned c = a ^ b;
```

[10]Non-pipelined systems notwithstanding.

## 9.10 Scalar multiplication using AND

Multiplication in $\mathbb{F}_2$ is simply multiplication mod 2. Here is a multiplication table:

| $\cdot$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

The logic people have another name for this function from $\mathbb{F}_2 \times \mathbb{F}_2$ to $\mathbb{F}_2$: the **AND**. Again thinking of 0 as false and 1 as true, then the result (the product) of two inputs is true if one is true *and* the other is true.

## 9.11 Bit-manipulation paradigms

C has several bitwise operators: AND, OR, XOR, NOT, LEFT-SHIFT and RIGHT-SHIFT. They all have standard uses which you will encounter.

- XOR: We've seen this already for polynomial addition. Also, it can be used to *toggle* one or more bits. For example:

  ```
  int i;
  unsigned x = 0xee;
  for (i = 0; i < 10; i++)
          x ^= 0x02;
  ```

  This will repeatedly toggle bit 1 (remember bit numbers start at 0 on the right) of the given value, resulting in the sequence 0xee, 0xec, 0xee, 0xec, . . . .

- OR: This **binary operator** (i.e. it takes two inputs) produces output 1 if either one input is 1, *or* the other input is 1. Unlike XOR for + and AND for $\cdot$, it doesn't have as clear a mathematical meaning. However it pops up a lot in the **bit set** idiom:

  ```
  unsigned x = 0xa0;
  x |= 1;
  ```

  This **sets** bits 0, whether it was set or not. If bit 0 is already known to be a 0, then this is equivalent to adding 1.

- NOT: This **unary operator** (i.e. it takes one input) **negates** its input, i.e. turns 1 bits to 0 bits and vice versa. It is spelt with a tilde, as follows:

```
unsigned a =  0xa0;
unsigned b = ~a;
```

This is often used in conjunction with the AND operator, as we will see.

- AND: We've seen this already for scalar multiplication. It crops up much more often, though, in two idioms:

  The **bit test** operation sees if certain bits are set. For example:

  ```
  if (x & 0x10)
          x ^= 0x13;
  ```

  Here, if `x` has bit 4 set, then bits 4, 1 and 0 are toggled.

  The **bit clear** idiom uses AND and NOT. For example:

  ```
  x &= ~0x20;
  ```

  This means, clear bit 5.

- LEFT-SHIFT and RIGHT-SHIFT: For left shift, each bit is moved one position to the left. The old most significant bit is shifted off; a zero bit moves in to fill the empty spot in the least significant bit. For right shift, the least significant bit is shifted off the right; if the input is unsigned, a zero bit moves in to fill the vacancy on the left[11].

  For example:

  ```
  int i;
  unsigned x = 1;

  for (i = 0; i < 4; i++) {
          printf(" %02x", x);
          x <<= 1;
  }
  for (i = 0; i < 4; i++) {
          printf(" %02x", x);
          x >>= 1;
  }
  printf("\n");
  ```

  This prints out 0x01, 0x02, 0x04, 0x08, 0x10, 0x08, 0x04, 0x02.

[11] For signed integers, *sign extension* is used on right shift, i.e. the new MSB takes the value of the old MSB. This is perfectly appropriate from an integer-arithmetic point of view, since a right shift corresponds to integer division by 2 and we want, say, $-12/2$ to be $-6$. However, for our purposes here, doing polynomial rather than integer arithmetic, we always want unsigned arithmetic. In fact this is the only point at which signed vs. unsigned matters for us.

## 9.12 Field multiplication using shifts and XORs

Field multiplication is as described in section 3.6. We just need to remember that we are getting a no-brained piece of silicon to do our work for us, so we need to spell out some things that you and I would take for granted. For example, what's the degree of a polynomial? You know just at a glance. For a particular application, of course your polynomial degree is probably known ahead of time. But for general-purpose use, the following will do[12]:

```
unsigned polydeg(unsigned r)
{
        int d = 0;
        if (r == 0) {
                return 0;
                // Or, treat this as an error condition, depending on
                // how you want to handle the degree of the zero polynomial.
                // For my purposes, it suffices to assign it degree zero, just
                // like all the other constant polynomials.
        }
        while (r >>= 1)
                d++;
        return d;
}
```

Given that, the following will suffice to perform multiplications. Again, for a given implementation you'll probably know your degree so you won't have to search for it, and the loops could be unrolled etc.

```
unsigned ffmul(unsigned a, unsigned b, unsigned r)
{
        int degb = polydeg(b);
        int degr = polydeg(r);
        unsigned bit_at_deg = 1 << degr;
        unsigned prod = 0;
        unsigned temp;
        int i, power;

        for (i = 0; i <= degb; i++) {
                // Test if this bit position is set.
                if (!(b & (1 << i)))
```

[12]There are several obvious ways to optimize this as well, but the current implementation is preferred for clarity.

```
                        continue;

                // Now multiply a by the power of x on the current term
                // of b, reducing mod r en route.
                temp = a;
                for (power = 1; power <= i; power++) {
                        temp <<= 1;
                        if (temp & bit_at_deg)  // If x^n appears,
                                temp ^= r;      // then subtract r(x).
                }
                // Add in this partial product.
                prod ^= temp;
        }

        return prod;
}
```

Note the use of several bit-manipulation paradigms.

## 9.13 Repeated squaring

Repeated squaring is easy to implement in software, since it matches up with binary arithmetic. For example:

```
unsigned ffpower(
        unsigned  a,
        int power, // This handles positive powers only.
        unsigned  r)
{
        unsigned a2 = a;
        unsigned out = 1;

        while (power != 0) {
                if (power & 1) // Test current bit
                        out = ffmul(out, a2, r);
                power = (unsigned)power >> 1; // Prepare to test next bit
                a2 = ffmul(a2, a2, r);
        }
        return out;
}
```

As described in section 2.6, we form $a$, $a^2$, etc. (in the `a2` variable), but include in our result (`out`) only the correct ones.

## 9.14 Reciprocation and division

Given repeated squaring, it's easy to recriprocate. Using Lagrange's theorem as described in section 3.11, we can reciprocate by raising to the $p^n - 2$ power:

```
unsigned ffrecip(unsigned b, unsigned r)
{
        int n = polydeg(r);
        int pn2 = (1 << n) - 2; // Incorrect if n == 0.
        if (b == 0) {
                ... Handle the divide-by-zero error as you wish.
        }
        return ffpower(b, pn2, r);
}
```

Then, $a/b$ is $a \cdot (b^{-1})$:

```
unsigned ffdiv(unsigned a, unsigned b, unsigned r)
{
        unsigned binv;
        binv = ffrecip(b, r);
        return ffmul(a, binv, r);
}
```

## 9.15 Arithmetic with implicit leading one

xxx add is the same

xxx hard-code polydeg calls

xxx re-write mul

## 9.16 Polynomial basis vs. log domain

## 9.17 Table lookups

xxx lexical ordering

xxx dissect the schneier examples, and attribute them.

xxx more about why Gal cfg is easier in C

# Part V

# Appendices

# Appendix A

# Blankinship's algorithm for extended GCD

Since $\mathbb{Z}$ and $\mathbb{F}_p[x]$ are both Euclidean domains, we have a Euclidean GCD algorithm for both. This algorithm is usually (in my experience) presented as follows:

- Input $a$, $b$; output $g$.
- If $b = 0$, $g := a$; stop. (This avoids a divide-by-zero exception below.)
- Let $c := a$, $d := b$.
- Top of loop:
  - Use integer or polynomial division on $c$ and $d$ to obtain quotient $q$ and remainder $r$ such that $q = cd + r$ and $r < d$ (for integers) or $\deg(r) < \deg(d)$ (for polynomials).
  - If $r = 0$, $g = d$; stop.
  - $c := d$, $d := r$.
  - Go to top of loop.

To then obtain $m$, $n$ such that $ma + nb = g$, one then works the algorithm in reverse. Not only is this tedious, but also it requires one to remember (on paper) or allocate space for (in software) all the intermediate $c$, $d$ values.

A superior method is due to Blankinship [9, Ch. 1.2.1]. The derivation is given in Knuth; here I will describe the algorithm. The key is that $m$ and $n$ are computed along with the GCD $g$, with no need for backtracking. The algorithm is:

- Input $a$, $b$; output $g$.

- If $b = 0$, $g := a$; stop. (This avoids a divide-by-zero exception below.)
- Let $c := a$, $d := b$.
- Let $m' := 1$, $m := 0$, $n' := 0$, $n := 1$.
- Top of loop:
  - Use integer or polynomial division on $c$ and $d$ to obtain quotient $q$ and remainder $r$ such that $q = cd+r$ and $r < d$ (for integers) or $\deg(r) < \deg(d)$ (for polynomials).
  - If $r = 0$, $g := d$; stop.
  - $c := d$, $d := r$.
  - Let $t := m'$, $m' := m$, $m = t - qm$.
  - Let $t := n'$, $n' := n$, $n = t - qn$.
  - Go to top of loop.

# Appendix B

# Shanks algorithm for discrete logs in a cyclic group

Above we saw how to compute logs using a linear search. That is, given a cyclic group $G$ of order $N$, a generator $g$ and an element $a$, simply test all the $N$ powers $g^0$, $g^1$, $g^2$, $\ldots$, $g^{N-1}$ to find which one is equal to $a$. (If none of them equals $a$, then $G$ is not a group, its order isn't really $N$, $g$ isn't really a generator, or $a$ isn't really a member of the group.)

This is fine for small groups. The time requirement is clearly $O(N)$, with storage space $O(1)$. However, for large groups there is a method due to Shanks [12] with much better execution time, at the expense of more storage space.

Let $G$, $N$, $g$ and $a$ be as above, and let $k$ be the logarithm of $a$ with respect to $g$, i.e. $a = g^k$, with $0 \leq k < N$. The trick is to view $k$ as a two-digit, base-$m$ number $i + jm$ where $m = \lfloor\sqrt{N}\rfloor$, and $0 \leq i, j < m$. Then

$$\begin{aligned} a &= g^{i+jm} \\ &= g^i g^{jm} \\ ag^{-i} &= g^{jm} \end{aligned}$$

The algorithm is as follows:

- Make two arrays of ordered pairs. The first array consists of the $m$ pairs $(i, ag^{-i})$; the second consists of the $m$ pairs $(j, g^{jm})$. (When you generate these arrays, probably you'll loop over the $i$'s and $j$'s, so the arrays will already be sorted by $i$'s and $j$'s.)
- Choose some ordering of group elements and sort both arrays by the second elements of the ordered pairs, i.e. the $ag^{-i}$'s and the $g^{jm}$'s.
- Search the arrays for a pair of pairs such that $ag^{-i} = g^{jm}$.

It can be shown (and is easy to see) that you can search a pair of sorted length-$m$ arrays in $O(m)$ comparisons. Clearly this is $O(\sqrt{N})$ in space, more expensive than the linear search. But the time savings from $O(N)$ to $O(\sqrt{N})$ is dramatic.

# Appendix C

# How to find primitive elements of $\mathbb{F}_p$

In section 2.7 on page 14 I discussed one way to find a primitive element of $\mathbb{F}_p$: search for it. That is, given a non-zero element $g$ of $\mathbb{F}_p$, write down all the $p-2$ powers $g$, $g^2$, $\ldots$, $g^{p-1}$ and check if they are all distinct. While this suffices to show that an element is a generator, it is more work than necessary.

From group theory, recall that if $g$ is a generator of the multiplicative group $\mathbb{F}_p^\times$ which has order $p-1$, then $g^{p-1} = 1$ and no smaller positive power sends $g$ to 1. Also recall that if $g$ were to have smaller order $k$, i.e. $g^k = 1$ for $k < p-1$, that order would need to divide the order of the group, i.e. $k \,|\, p-1$. Thus we can factor $p-1$, and raise $g$ only to powers dividing $p-1$. If any proper divisor sends $g$ to 1, $g$ is not a generator. As an error check on one's computations, it is nice to also verify that $g^{p-1}$ *is* 1.

For example, let's see if 2 generates $\mathbb{F}_{23}$. Since proper divisors of $p-1 = 22$ are 1, 2 and 11, we only need to check 2, $2^2$ and $2^{11}$. Clearly, neither 2 nor 4 is 1 mod 23; $2^{11} = 2048$, which mod 23 is 1 since $2047 = 23 \cdot 89$. Therefore 2 is not a generator. Likewise 3 and 4 fail, but 5 succeeds.

# Appendix D

# How to find irreducible polynomials in $\mathbb{F}_p[x]$

In section 3.4 on page 20 I discussed a few techniques for finding irreducible polynomials in $\mathbb{F}_p[x]$:

- For degree three or less, use the root test since if there is a non-trivial factorization, at least one factor must be linear.
- For degree four above, first use the root test to check for linear factors, then use trial division by factors of degree 2 through $n/2$.

Here are a few more options.

## D.1 Universal-polynomial algorithm

There is an irreducibility algorithm, much more efficient than trial division, which I found in a snippet of C code on thee web a while back. I would attribute it to the proper person, but I have since lost the URL and haven't found it in any other references. Nonetheless, I've discovered that the concept is very simple.

In section 4 on page 31 I defined the universal polynomial $x^{p^n} - x$, and noted that this factors into the product of all monic irreducibles of degree $d$ for all $d$ dividing $n$. Suppose $r(x)$ is a given polynomial in $\mathbb{F}_p[x]$ and let $m$ be the degree of $r(x)$. Divide $r(x)$ by its leading coefficient to make it monic, if it is not already: thus any factors of $r$ will also be monic. If $r(x)$ has an irreducible factor of degree $i$ less than $m$, then that factor will divide

$x^{p^i} - x$. Contrapositively, if $r(x)$ has no common factors with $x^{p^i} - x$ for $i < m$, then $r(x)$ is irreducible.

(Example: Suppose $r(x)$ is of degree 8, with irreducible factors $r_1$ and $r_2$ of degrees 3 and 5, respectively. Then $r_1$ must be one of the factors of $x^{p^3} - x$, and $r_2$ must be one of the factors of $x^{p^5} - x$.)

Now, $x^{p^m} - x$ can be a very high-degree polynomial, which makes sense since it can have a lot of factors. Fortunately, we don't care about all of those factors, so we can avoid huge polynomials as follows: For each $i$ from 1 up to and including $m - 1$, find $x^{p^i} - x$ mod $r(x)$ and compute the GCD of $r$ with the remainder. If there is a common factor, it will be retained mod $r(x)$. Reducing mod $r(x)$ just keeps the degree reasonable. If the GCD is 1 for each power, $r$ is irreducible.

More specifically, here is the algorithm:

```
• Input $p$ a prime, $r(x)$ in $\mathbb{F}_p[x]$.
• Set $m := \deg\langle r(x)\rangle$
• Set $U(x) := x$
• For $i$ from 1 to $m - 1$:
    – Set $U(x) := (U(x))^p \mod r$. (Here, $U$ is $x^{p^i} \mod r$.)
    – Set $g(x) := \mathrm{GCD}(r(x), U(x) - x)$
    – If $g(x) \neq 1$ then return REDUCIBLE
• Return IRREDUCIBLE
```

## D.2 Derivative test for multiple factors

One may quickly test for repeated factors by computing the GCD of $r$ and its formal derivative. This is not an irreducibility test, but it can be a quick way to reveal some factors. Also it is a necessary preliminary for the next option.

## D.3 One pass of Berlekamp's algorithm

The above universal-polynomial algorithm is efficient, but for implementation in a computer program it turns out that it is even more efficient to run one pass of Berlekamp's factorization algorithm (which requires squarefree input): [10, Ch. 4.1] and appendix F.

## D.4 All-irreducibles algorithm

The discussion in section 4.3 on page 33 immediately suggests a technique to produce all the remaining irreducibles of a given degree, given a single one (found e.g. by one of the above methods).

- Given $r(x)$ of degree $n$ in $\mathbb{F}_p[x]$, write down all the $p^n$ elements of the finite field $\mathbb{F}_p[x]/\langle r(x)\rangle$.
- Perhaps after making up a log table to facilitate computation, for each element $\alpha$ take repeated $p$th powers until you get a repetition. This will give you Frobenius orbits of each element.
- For each Frobenius orbit, take all the distinct $\rho^i(\alpha)$'s, write down $\prod_i (x - \rho^i(\alpha))$ and multiply it out using arithmetic defined by $r(x)$. This will give you minimal polynomials for each element.
- You will obtain a result much like the root chart in section 4.3 on page 33. Discard all minimal polynomials of degree less than $n$. The remaining minimal polynomials are all the monic irreducibles of degree $n$.

Example:

- Given $x^3 + x + 1$ of degree 3 in $\mathbb{F}_2[x]$, we have the eight field elements 000, 001, 010, 011, 100, 101, 110, 111.
- $000^2 = 000$
- $001^2 = 001$
- $010^2 = 100$; $100^2 = 110$; $110^2 = 010$.
- $011^2 = 101$; $101^2 = 111$; $111^2 = 011$.
- $(x - 000) = x$
- $(x - 001) = x + 1$
- $(x - 010)(x - 100)(x - 110) = x^3 + x + 1$
- $(x - 011)(x - 101)(x - 111) = x^3 + x^2 + 1$
- Degree-3 polynomials in the above are $x^3 + x + 1$ and $x^3 + x^2 + 1$.

# Appendix E

# How to find primitive polynomials in $\mathbb{F}_p[x]$

In appendix C on page 88 I discussed how to find primitive elements of $\mathbb{F}_p$. The key is that $\mathbb{F}_p^\times$ is cyclic of order $p-1$; given $a$ in $\mathbb{F}_p$, it suffices to check that $a^d \neq 1$ for all proper divisors $d$ of $p-1$.

Here the same concept applies. Given $r(x)$, first test that $r(x)$ is irreducible as described in appendix D on page 89. Second, $r(x)$ will be primitive if $u$ is a primitive element mod $r$. Let $n = \deg(r)$ and work in the finite field $\mathbb{F}_p[x]/\langle r(x) \rangle$. Recall that this field has a multiplicative group of order $p^n - 1$. Find all proper divisors $d$ of $p^n - 1$. If $u^d \neq 1$ for all of them, then $u$ is a primitive element mod $r$, and $r$ is a primitive polynomial. Otherwise, $r$ is not primitive.

Note: The amount of work involved depends highly on the factorization of $p^n - 1$. Also note that for $p = 3$ and $n = 2$, or for $p = 2$ and various values of $n$, $p^n - 1$ can be a prime number. In such cases, *all* monic irreducibles of degree $n$ are primitive, by Lagrange's theorem. (In fact, by Lagrange's theorem all elements of the field other than 0 and 1 have order $p^n - 1$.)

# Appendix F

# Factoring in $\mathbb{F}_p[x]$ and $\mathbb{F}_{p^n}[x]$

In the main body of the paper I discussed factoring by trial division, which is possible since given a polynomial $f(x)$ in $\mathbb{F}_{p^n}[x]$ there are finitely many polynomials of degree less than the degree of $f(x)$. This is fine (and even preferable) for small $p$, $n$ and degree. However, for factoring larger polynomials in a software implementation, a much more efficient method is available.

The following algorithm (Berlekamp's along with necessary preprocessing) is proved correct in Lidl and Niederreiter [10]; here, I show only the algorithm itself. I note that, in contrast to the much harder problem of factoring polynomials over $\mathbb{Z}[x]$ or $\mathbb{Q}[x]$, this algorithm is easy to implement in software and also is very efficient. That is, factoring polynomials in $\mathbb{F}_p[x]$ and $\mathbb{F}_{p^n}[x]$ is "easy" whereas factoring in $\mathbb{Z}[x]$ or $\mathbb{Q}[x]$ (by current methods, at least) is "hard".

Input is $f(x) \in \mathbb{F}_{p^n}[x]$. For input in $\mathbb{F}_p[x]$, just use $n = 1$. Let $m = \deg(f(x))$.

## F.1 Recursion

The algorithm is recursive: as presented here, it will either determine that a polynomial is irreducible, or it will produce a pair of non-trivial factors. If one's goal is merely to determine whether $f(x)$ is irreducible, this pass is sufficient, but if one's goal is to produce a complete factorization of $f(x)$, one must repeat the algorithm for both factors.

## F.2 Squarefree preparation

Berlekamp's algorithm requires squarefree input. This may be obtained by first taking the GCD of the input polynomial and its derivative:

- Assume that the input $f(x)$ is non-zero and has degree greater than 1 (otherwise we don't need to be here).
- $g(x) = \mathrm{GCD}(f(x), f'(x))$
- If $g(x)$ has degree 0, $f(x)$ is squarefree; proceed to Berlekamp's algorithm.
- Else if $f'(x) = 0$, the input is a perfect $p$ power of the form

  $$a_k x^{kp} + a_{k-1} x^{(k-1)p} + \ldots + a_1 x^p + a_0$$

  for $k = m/p$, with $p$th root

  $$a_k x^k + a_{k-1} x^{k-1} + \ldots + a_1 x + a_0$$

  This root may or may not be squarefree; recursively apply this step to the $p$th root.
- Else $f(x)$ has proper factors $g(x)$ and $f(x)/g(x)$. Repeat this step for the former, and proceed to Berlekamp's algorithm for the latter.

## F.3 Berlekamp's algorithm

Input $f(x) \in \mathbb{F}_{p^n}[x]$ is assumed squarefree as described above, with $m = \deg(f(x))$. The key insight [10] is that if we can find an $h(x) \in \mathbb{F}_{p^n}[x]$ such that $h(x)^{p^n} \equiv h(x) \bmod f(x)$, then

$$f(x) = \prod_{c \in \mathbb{F}_{p^n}} \mathrm{GCD}(f(x), h(x) - c)$$

The algorithm uses linear algebra on the field $\mathbb{F}_{p^n}$ to find such an $h(x)$.

- Allocate an $m \times m$ matrix $B$, whose elements will be elements of $\mathbb{F}_{p^n}$.
- Populate the $B$ matrix with row $i$ equal to the coefficients of $x^{p^n i} \bmod f(x)$, for $0 \leq i < m$.
- Replace $B$ with $B - I$, i.e. subtract 1 on the main diagonal.
- Transpose the $B$ matrix, i.e. $B := B^t$.

- Using row reduction, put the matrix in row echelon form. (Note: One can use column reduction with an untransposed $B$, but I feel that, at least for myself, row reduction is more familiar and less error-prone.)
- The nullity of the matrix is the number of irreducible factors of $f(x)$. However, as presented here the algorithm will only consider the cases when the nullity is 1, or greater than 1.
- Compute a basis for the nullspace of the reduced matrix. If the nullspace has dimension 1 (i.e. if the rank of the reduced matrix is $m-1$), then $f(x)$ is irreducible in $\mathbb{F}_{p^n}[x]$; stop.
- For each row, let $h(x)$ have coefficients corresponding to row elements; for each $c \in \mathbb{F}_{p^n}$, compute $f_1(x) = \mathrm{GCD}(f(x), h(x) - c)$.
- When an $h(x)$ and a $c$ is found[1] such that $\deg(f_1(x))$ is greater than 0 and less than $\deg(f(x))$, $f_1(x)$ and $f_2(x) = f(x)/f_1(x)$ are proper factors. Recursively apply this algorithm to each of them until a factorization into irreducibles has been obtained.

[1]Such will always be found, but not all rows will suffice: e.g. one row will produce $h(x) = 1$, which does not lead to a non-trivial factorization.

# Appendix G

# How to construct a root chart

There are two ways to construct a root chart (example root charts appear in sections 4.3 and 6.3):

(i) Given the polynomial $x^{p^n} - x$ and a monic irreducible degree-$n$ polynomial $r(x)$ in $\mathbb{F}_p[x]$ (see appendix D for irreducibility testing) with which to define arithmetic for an $\mathbb{F}_{p^n}$, use Berlekamp's algorithm (appendix F) to factor $x^{p^n} - x$ in $\mathbb{F}_p[x]$. Then, for each resulting monic irreducible factor, use Berlekamp's algorithm in $\mathbb{F}_{p^n}[x]$, or use trial evaluations over all elements of $\mathbb{F}_{p^n}$, to find roots in $\mathbb{F}_{p^n}$ for each of the resulting factors.

(ii) Use Galois theory as described in section 5:

- Write down a logarithm table for $\mathbb{F}_{p^n}$.
- For increasing divisors $d$ of $n$, use the subfield/log criterion to find elements of $\mathbb{F}_{p^d}$, i.e. elements whose log is a multiple of $(p^n - 1)/(p^d - 1)$.
- For each resulting element $a$ of $\mathbb{F}_{p^d}$, write down the $d$ distinct Frobenius-orbit elements $a$, $a^p$, …, $a^{p^{d-1}}$. Cross them all off from the log table so they will be processed only once. Also compute the product

  $$\prod_{i=1}^{d} (x - a^{p^i})$$

  This is the minimal polynomial for all $d$ elements in the Frobenius orbit. Write down this minimal polynomial, followed by the $d$ roots.
- Once all elements in the log table have been crossed off, sort rows of the root chart lexically by the minimal polynomials in the left-hand column.

The latter method is far easier to implement with pencil and paper, and also runs faster in software.

# Appendix H

# Tables

## H.1 Tables of some irreducible polynomials in $\mathbb{F}_p[x]$

Primitive polynomials are marked with an asterisk. Please see Lidl and Niederreiter [10] for much more extensive tables.

| $\mathbb{F}_{2^2}$ | | $\mathbb{F}_{2^3}$ | | $\mathbb{F}_{2^4}$ | | $\mathbb{F}_{2^5}$ | | $\mathbb{F}_{2^6}$ | |
|---|---|---|---|---|---|---|---|---|---|
| 111 | * | 1011 | * | 10011 | * | 100101 | * | 1000011 | * |
| | | 1101 | * | 11001 | * | 101001 | * | 1001001 | |
| | | | | 11111 | | 101111 | * | 1010111 | |
| | | | | | | 110111 | * | 1011011 | * |
| | | | | | | 111011 | * | 1100001 | * |
| | | | | | | 111101 | * | 1100111 | * |
| | | | | | | | | 1101101 | * |
| | | | | | | | | 1110011 | * |
| | | | | | | | | 1110101 | |

| $\mathbb{F}_{3^2}$ | | $\mathbb{F}_{3^3}$ | |
|---|---|---|---|
| 101 | | 1021 | * |
| 112 | * | 1022 | |
| 122 | * | 1102 | |
| | | 1112 | |
| | | 1121 | * |
| | | 1201 | * |
| | | 1211 | * |
| | | 1222 | |

## H.2 Some primitive irreducible polynomials in $\mathbb{F}_{2^n}$

| $n$ | $r$ in hex | Tap bits | $n$ | $r$ in hex | Tap bits |
|---|---|---|---|---|---|
| 1 | 3 | 0 | 33 | 200000053 | 6 4 1 0 |
| 2 | 7 | 1 0 | 34 | 40000001b | 4 3 1 0 |
| 3 | b | 1 0 | 35 | 800000005 | 2 0 |
| 4 | 13 | 1 0 | 36 | 100000003f | 5 4 3 2 1 0 |
| 5 | 25 | 2 0 | 37 | 200000003f | 5 4 3 2 1 0 |
| 6 | 43 | 1 0 | 38 | 4000000063 | 6 5 1 0 |
| 7 | 83 | 1 0 | 39 | 8000000011 | 4 0 |
| 8 | 11d | 4 3 2 0 | 40 | 10000000039 | 5 4 3 0 |
| 9 | 211 | 4 0 | 41 | 20000000009 | 3 0 |
| 10 | 409 | 3 0 | 42 | 40000000027 | 5 2 1 0 |
| 11 | 805 | 2 0 | 43 | 80000000059 | 6 4 3 0 |
| 12 | 1053 | 6 4 1 0 | 44 | 100000000021 | 5 0 |
| 13 | 201b | 4 3 1 0 | 45 | 20000000001b | 4 3 1 0 |
| 14 | 402b | 5 3 1 0 | 46 | 400000000003 | 1 0 |
| 15 | 8003 | 1 0 | 47 | 800000000021 | 5 0 |
| 16 | 1002d | 5 3 2 0 | 48 | 100000000002d | 5 3 2 0 |
| 17 | 20009 | 3 0 | 49 | 2000000000071 | 6 5 4 0 |
| 18 | 40027 | 5 2 1 0 | 50 | 400000000001d | 4 3 2 0 |
| 19 | 80027 | 5 2 1 0 | 51 | 800000000004b | 6 3 1 0 |
| 20 | 100009 | 3 0 | 52 | 10000000000009 | 3 0 |
| 21 | 200005 | 2 0 | 53 | 20000000000047 | 6 2 1 0 |
| 22 | 400003 | 1 0 | 54 | 4000000000007d | 6 5 4 3 2 0 |
| 23 | 800021 | 5 0 | 55 | 80000000000047 | 6 2 1 0 |
| 24 | 100001b | 4 3 1 0 | 56 | 100000000000095 | 7 4 2 0 |
| 25 | 2000009 | 3 0 | 57 | 200000000000011 | 4 0 |
| 26 | 4000047 | 6 2 1 0 | 58 | 400000000000063 | 6 5 1 0 |
| 27 | 8000027 | 5 2 1 0 | 59 | 80000000000007b | 6 5 4 3 1 0 |
| 28 | 10000009 | 3 0 | 60 | 1000000000000003 | 1 0 |
| 29 | 20000005 | 2 0 | 61 | 2000000000000027 | 5 2 1 0 |
| 30 | 40000053 | 6 4 1 0 | 62 | 4000000000000069 | 6 5 3 0 |
| 31 | 80000009 | 3 0 | 63 | 8000000000000003 | 1 0 |
| 32 | 1000000af | 7 5 3 2 1 0 | 64 | 1000000000000001b | 4 3 1 0 |

## H.3 Some logarithm tables for $\mathbb{F}_{p^n}$

Log
$p = 2,\ r = 111,\ g = 10$

| $k$ | $g^k$ | order |
|---|---|---|
| 3 | 01 | 1 |
| 1 | 10 | 3 |
| 2 | 11 | 3 |

Antilog
$p = 2,\ r = 111,\ g = 10$

| $k$ | $g^k$ | order |
|---|---|---|
| 1 | 10 | 3 |
| 2 | 11 | 3 |
| 3 | 01 | 1 |

Log
$p = 2,\ r = 1011,\ g = 10$

| $k$ | $g^k$ | order |
|---|---|---|
| 7 | 001 | 1 |
| 1 | 010 | 7 |
| 3 | 011 | 7 |
| 2 | 100 | 7 |
| 6 | 101 | 7 |
| 4 | 110 | 7 |
| 5 | 111 | 7 |

Antilog
$p = 2,\ r = 1011,\ g = 10$

| $k$ | $g^k$ | order |
|---|---|---|
| 1 | 010 | 7 |
| 2 | 100 | 7 |
| 3 | 011 | 7 |
| 4 | 110 | 7 |
| 5 | 111 | 7 |
| 6 | 101 | 7 |
| 7 | 001 | 1 |

Log
$p = 2,\ r = 1101,\ g = 10$

| $k$ | $g^k$ | order |
|---|---|---|
| 7 | 001 | 1 |
| 1 | 010 | 7 |
| 5 | 011 | 7 |
| 2 | 100 | 7 |
| 3 | 101 | 7 |
| 6 | 110 | 7 |
| 4 | 111 | 7 |

Antilog
$p = 2,\ r = 1101,\ g = 10$

| $k$ | $g^k$ | order |
|---|---|---|
| 1 | 010 | 7 |
| 2 | 100 | 7 |
| 3 | 101 | 7 |
| 4 | 111 | 7 |
| 5 | 011 | 7 |
| 6 | 110 | 7 |
| 7 | 001 | 1 |

Log

$p = 2$, $r = 10011$, $g = 10$

| $k$ | $g^k$ | order |
|---|---|---|
| 15 | 0001 | 1 |
| 1 | 0010 | 15 |
| 4 | 0011 | 15 |
| 2 | 0100 | 15 |
| 8 | 0101 | 15 |
| 5 | 0110 | 3 |
| 10 | 0111 | 3 |
| 3 | 1000 | 5 |
| 14 | 1001 | 15 |
| 9 | 1010 | 5 |
| 7 | 1011 | 15 |
| 6 | 1100 | 5 |
| 13 | 1101 | 15 |
| 11 | 1110 | 15 |
| 12 | 1111 | 5 |

Antilog

$p = 2$, $r = 10011$, $g = 10$

| $k$ | $g^k$ | order |
|---|---|---|
| 1 | 0010 | 15 |
| 2 | 0100 | 15 |
| 3 | 1000 | 5 |
| 4 | 0011 | 15 |
| 5 | 0110 | 3 |
| 6 | 1100 | 5 |
| 7 | 1011 | 15 |
| 8 | 0101 | 15 |
| 9 | 1010 | 5 |
| 10 | 0111 | 3 |
| 11 | 1110 | 15 |
| 12 | 1111 | 5 |
| 13 | 1101 | 15 |
| 14 | 1001 | 15 |
| 15 | 0001 | 1 |

Log

$p = 2$, $r = 11001$, $g = 10$

| $k$ | $g^k$ | order |
|---|---|---|
| 15 | 0001 | 1 |
| 1 | 0010 | 15 |
| 12 | 0011 | 5 |
| 2 | 0100 | 15 |
| 9 | 0101 | 5 |
| 13 | 0110 | 15 |
| 7 | 0111 | 15 |
| 3 | 1000 | 5 |
| 4 | 1001 | 15 |
| 10 | 1010 | 3 |
| 5 | 1011 | 3 |
| 14 | 1100 | 15 |
| 11 | 1101 | 15 |
| 8 | 1110 | 15 |
| 6 | 1111 | 5 |

Antilog

$p = 2$, $r = 11001$, $g = 10$

| $k$ | $g^k$ | order |
|---|---|---|
| 1 | 0010 | 15 |
| 2 | 0100 | 15 |
| 3 | 1000 | 5 |
| 4 | 1001 | 15 |
| 5 | 1011 | 3 |
| 6 | 1111 | 5 |
| 7 | 0111 | 15 |
| 8 | 1110 | 15 |
| 9 | 0101 | 5 |
| 10 | 1010 | 3 |
| 11 | 1101 | 15 |
| 12 | 0011 | 5 |
| 13 | 0110 | 15 |
| 14 | 1100 | 15 |
| 15 | 0001 | 1 |

Log

$p = 2$, $r = 11111$, $g = 11$

| $k$ | $g^k$ | order |
|---|---|---|
| 15 | 0001 | 1 |
| 12 | 0010 | 5 |
| 1 | 0011 | 15 |
| 9 | 0100 | 5 |
| 2 | 0101 | 15 |
| 13 | 0110 | 15 |
| 7 | 0111 | 15 |
| 6 | 1000 | 5 |
| 8 | 1001 | 15 |
| 14 | 1010 | 15 |
| 11 | 1011 | 15 |
| 10 | 1100 | 3 |
| 5 | 1101 | 3 |
| 4 | 1110 | 15 |
| 3 | 1111 | 5 |

Antilog

$p = 2$, $r = 11111$, $g = 11$

| $k$ | $g^k$ | order |
|---|---|---|
| 1 | 0011 | 15 |
| 2 | 0101 | 15 |
| 3 | 1111 | 5 |
| 4 | 1110 | 15 |
| 5 | 1101 | 3 |
| 6 | 1000 | 5 |
| 7 | 0111 | 15 |
| 8 | 1001 | 15 |
| 9 | 0100 | 5 |
| 10 | 1100 | 3 |
| 11 | 1011 | 15 |
| 12 | 0010 | 5 |
| 13 | 0110 | 15 |
| 14 | 1010 | 15 |
| 15 | 0001 | 1 |

# Appendix I

# Storage representations

We've already seen a few different ways to represent elements of finite fields, using the canonical embodiment $\mathbb{F}_p[x]/\langle r(x)\rangle$:

- As equivalence classes of remainders mod $r$, e.g. $x^2 + x + \langle x^4 + x + 1\rangle$.
- As $\mathbb{F}_p[u] \cong \mathbb{F}_p[x]/\langle r(x)\rangle$, with $u$ a root of $r(x)$, e.g. $u^2 + u$.
- As an array or $n$-tuple of coefficients, e.g. $[0, 1, 1, 0]$ or $(0, 1, 1, 0)$ but usually written just 0110.

There are some other notations that are also encountered in the literature:

- As logarithms, as discussed above. This is sometimes referred to as the **log domain**. It leaves you with the question of how to write 0, which doesn't have a log.
- We can treat an $n$-tuple of coefficients as a base-$p$ integer, then convert bases from $p$ to 10. E.g. elements of $\mathbb{F}_{2^4}$ would then convert from 0000, 0001, 0010, . . . , 1111 to 0, 1, 2, . . . , 15. Since this is decimal, it looks confusingly like arithmetic mod $p^n$ but of course, unless $n = 1$, it's not (which is one reason I don't like this notation).
- For base $p = 2$, engineers often use base-16 or **hexadecimal** notation. For digits, use 0-9 for 0-9 and a-f (or A-F, depending on personal preference) for 11-15; prefix with 0x to distinguish from decimal. E.g. 1010 is 10 in decimal, which is 0xa in hex. Likewise, 10101110 is 0xae in hex. The three monic irreducibles in $\mathbb{F}_{2^4}$ are written 0x13, 0x19 and 0x1f; 0x104c11db7 is a degree-32 irreducible, primitive polynomial in $\mathbb{F}_2[x]$ (which incidentally is used for error detection in Ethernet networking); the AES polynomial [1] is 0x11b.

# Bibliography

[1] J. Daemen and V. Rijmen. *The Rijndael Block Cipher.* `csrc.nist.gov/CryptoToolkit/aes/rijndael`.

[2] P. Alfke. *Efficient shift registers, LFSR counters, and long pseudo-random sequence generators.* Xilinx Corporation, 1996. `http://direct.xilinx.com/bvdocs/appnotes/xapp052.pdf`

[3] E. Berlekamp. *Algebraic Coding Theory* (revised 1984 edition). Aegean Park Press, 1984.

[4] D.S. Dummit and R.M. Foote. *Abstract Algebra* (2nd ed.). John Wiley and Sons, 1999.

[5] S. Golomb. *Shift Register Sequences (revised edition).* Aegean Park Press, 1982.

[6] H. Cohen, K. Belebas et al. PARI/GP. `pari.math.u-bordeaux.fr`.

[7] P. Horowitz and W. Hill. *The Art of Electronics.* Cambridge University Press, 1989.

[8] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, 1990.

[9] D.E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (2nd ed.). Addison-Wesley, 1973.

[10] R. Lidl and H. Niederreiter. *Finite Fields.* Cambridge University Press, 1997.

[11] T. Murphy. *Course 373, Finite Fields.* `www.maths.tcd.ie/pub/Maths/Courseware/FiniteFields/FiniteFields.pdf`.

[12] A. Odlyzko. *Discrete logs: the past and the future.* `www.dtc.umn.edu/~odlyzko/doc/discrete.logs.future.pdf`.

[13] B. Schneier. *Applied Cryptography.* John Wiley and Sons, 1995.

[14] B. Tsaban and U. Vishne. *Efficient linear feedback shift registers with maximal period.* `front.math.ucdavis.edu/cs.CR/0304010`.

[15] R. Williams. *A painless guide to CRC error detection algorithms.* `ftp.rocksoft.com/papers/crc_v3.txt`.