Spring 5-1-2014

Polynomial Factoring Algorithms and their Computational Complexity

Nicholas Cavanna
University of Connecticut - Storrs, nicholas.j.cavanna@uconn.edu

Polynomial Factoring Algorithms and their Computational Complexity

Nicholas Cavanna

advisor: Dr. Jeremy Teitelbaum

May 23, 2014

Introduction

Finite fields, and the polynomial rings over them, have many neat algebraic properties and identities that are very convenient to work with. In this paper we will start by exploring said properties with the goal in mind of being able to use said properties to efficiently irreducibly factorize polynomials over these fields, an important action in the fields of discrete mathematics and computer science. Necessarily, we must also introduce the concept of an algorithm's speed as well as particularly speeds of basic modular and integral arithmetic operations. Outlining these concepts will have laid the groundwork for us to introduce the Berlekamp algorithm, as well as the Cantor-Zassenhaus algorithm: two different approaches to the problem of factoring polynomials over finite fields, both in the algebraic properties the utilize as well as how the algorithms actually operate. This will lead us to a much harder problem, factoring polynomials over the integers. We will explain and prove how an elegant solution to this problem is directly related to finding the shortest vector in a lattice which also happens to use factoring over finite fields as a starting point. To conclude, we will give an efficient algorithm for factoring polynomials over the integers that circumvents the shortest vector requirement via an approximate solution, the LLL lattice basis reduction algorithm.

Field and Rings

In this section, we will briefly clarify the algebraic structures whose properties that will be used in the Berlekamp and Cantor-Zassenhaus factoring algorithms. As stated earlier, these will be taking a polynomial over a finite field and factoring it into irreducible polynomials.

Proposition 1. *Let $\mathbb{F}$ be a field with finitely many elements (a "finite field"), then the following hold:*

1. $char(\mathbb{F}) > 0$

2. char$(\mathbb{F}) = p$ *for some prime* p.

3. $|\mathbb{F}| = p^k$ *for some* k

Proof. 1. Since there are only finitely many elements in $\mathbb{F}$, for some $a > b > 0$, $1 \cdot a = 1 \cdot b$, which implies $(a-b) \cdot 1 = 0$, so $\text{char}(\mathbb{F}) | (a-b)$ and thus it is non-zero.

2. Assume that $\text{char}(\mathbb{F}) = q \cdot r$ for some $q, r > 1$, then $qr \cdot 1 = 0$. However, all non-zero elements in a field are invertible which implies there are no zero-divisors, so $q = 0$ or $r = 0$, a contradiction, so it must have prime characteristic.

3. Given $\text{char}(\mathbb{F}) = p$ for some prime p, then $\mathbb{F}_p$ is a subfield of $\mathbb{F}$, so $\mathbb{F}$ is a finite k-dimensional vector space over $\mathbb{F}_p$ and thus $\mathbb{F}$ has p^k elements. □

If we are to consider a polynomial over a finite field $\mathbb{F}_q$, where $q = p^k$, then it is a function $f(x) = \sum_{i=0}^{n} c_i x^i$ where n is the degree and $c_i \in \mathbb{F}_q$. We use the notation $\mathbb{F}_q[X]$ to describe the ring of polynomials over a finite field. Later we show how to construct these fields.

Definition 1. *Euclidean domain*

A Euclidean domain R *is an integral domain such that there exists a function* $d(\cdot) : R/\{0\} \to \mathbb{N}$ *such that for all elements* a,$b \in R$, *and* $b \neq 0$, *there exists* $q, r \in R$ *where*

$$a = bq + r, \text{ where } r = 0 \text{ or } d(r) < d(b).$$

By this definition, we can see $\mathbb{F}_q[X]$ is a Euclidean domain as polynomial rings over fields admit (polynomial) division with unique remainder with respect to the degree of the polynomial.

Definition 2. *A principal ideal domain* R *is an integral domain in which every ideal is principal i.e. generated by one element.*

Proposition 2. *All Euclidean domains are principal ideal domains.*

Proof. Given an ideal I of a Euclidean domain R with Euclidean function d, if $I = (0)$ then we are done. Assume $I \neq (0)$. Since R is Euclidean, then consider the element $b \in I$ such that $d(b)$ is minimal with respect to the non-zero elements of I, where we can make such a choice as d is a function into the well-ordered natural numbers. Given another $a \in I$, $a = bq + r$, for some q,$r \in R$ such that $d(r) < d(b)$ or $d(r) = 0$. Rearranging, $a - bq = r \in I$ as a and $b \in I$, but as $d(b)$ was minimal for all elements of I, then $r = 0$, implying $a = bq$ and $a \in (b) = I$, so I is a principal ideal. □

Proposition 3. *All principal ideal domains are unique factorization domains.*

See [1, page 286] for a proof.

This is an important result, because as we now know that $\mathbb{F}_q[X]$ is a unique factorization domain, so we know that if we obtain a factorization of a polynomial over a finite field, then it is unique up to scaling by units.

Definition 3. *A monic polynomial $f(x) \in \mathbb{F}_q[X]$ of degree ≥ 1 is called irreducible if the only polynomials in $\mathbb{F}_q[X]$ that divide $f(x)$ are 1 and $f(x)$.*

Let us now introduce the Euclidean algorithm for calculating the greatest common divisor of two elements of $\mathbb{F}_q[X]$, where the "greatest" common divisor is with respect to the Euclidean function, the divisor of two polynomials with the highest degree.

Algorithm 1. *Euclidean algorithm*

Input: Two non-zero polynomials in $\mathbb{F}_q[X]$, $f(x)$ and $g(x)$, where $\deg(f) = n \geq \deg(g)$, Without loss of generality

Ouput: $\gcd(f(x), g(x))$

Step 1: Divide $f(x)$ by $g(x)$ to find $q(x), r(x)$ such that $f(x) = g(x)q(x) + r(x)$, and $deg(r(x)) < deg(g(x))$ or $r(x) = 0$

Step 2: If $r(x) = 0$, then $\gcd(f(x), g(x)) = g(x)$, else set $f(x) \leftarrow g(x)$ and $g(x) \leftarrow r(x)$ and go to step 1.

By working through the Euclidean algorithm backwards, one can arrive at the conclusion that given two polynomials $f(x)$ and $g(x)$ with $\gcd(f(x), g(x)) = d(x)$, there exists $u(x)$, $v(x) \in \mathbb{F}_q[X]$, such that $f(x)u(x) + g(x)v(x) = d(x)$. This is known as Bèzout's Identity.

Next we will introduce the definition of a square-free polynomial. This will prove to be an important part of factoring polynomials over finite fields.

Definition 4. *A square-free polynomial in $\mathbb{F}_q[X]$ is a monic polynomial of degree ≥ 1 that has no multiple irreducible polynomial factors i.e. $f(x) = \prod_i^n f_i(x)^{e_i}$, where $f_i(x)$ are monic non-trivial irreducible polynomials, is square-free iff $e_i = 1$ for all i.*

Proposition 4. *A monic polynomial $f(x) \in \mathbb{F}_q[X]$ is square-free iff*

$$(f(x), f'(x)) = 1$$

.

Proof. Given a square-free polynomial $f(x) \in \mathbb{F}_q[X]$ of degree n, in its splitting field

$$f(x) = \prod_{i=1}^{n}(x - \alpha_i).$$

Thus

$$f'(x) = \sum_{i=1}^{n}(\prod_{i \neq j}(x - \alpha_j)),$$

in the splitting field. Given any $x - \alpha_i$,

$$(x - \alpha_i) \not| \prod_{i \neq j}(x - \alpha_j),$$

and moreover each $x - \alpha_i$ divides all but one of the summation terms in $f'(x)$, which implies that $\gcd(f(x), f'(x)) = 1$ in the splitting field, so it holds in $\mathbb{F}_q[X]$.

If $f(x)$ is not square-free then $f(x) = g(x)^2 h(x)$ for some monic polynomials $g(x), h(x) \in \mathbb{F}_q[X]$ with degree ≥ 1, which implies

$$f'(x) = 2g(x)g'(x)h(x) + g(x)^2 h'(x),$$

and thus $g(x)|f(x)$ and $g(x)|f'(x)$, so $\gcd(f(x), f'(x)) \neq 1$ and by the contrapositive we are done. □

We can construct a finite field with p^d elements, $\mathbb{F}_q$, by considering a polynomial ring $\mathbb{F}_p[X]$ modulo an irreducible polynomial $f(x) \in \mathbb{F}_p[X]$ of degree d.

This is a ring with p^d elements, as if $f(x) = \sum_{i=0}^{d} c_i x^i$, for $c_i \in \mathbb{F}_p$, then by the uniqueness of the remainder via the division algorithm, there is a unique representative modulo $f(x)$ with degrees from 0 to $d - 1$, which implies there are p^d elements. Taking another polynomial $g(x) \in \mathbb{F}_p[X]$ such that $f(x) \not| g(x)$, we know that $(g(x), f(x)) = 1$, as $f(x)$ is irreducible and by Bezout's identity there exists $u(x), v(x) \in \mathbb{F}_p[X]$ such that

$$g(x)u(x) + f(x)v(x) = 1.$$

So now in $\mathbb{F}_p[X]/f(x)$, $\overline{g}(x)\overline{u}(x) = 1$, i.e. $\overline{g}(x)$ has an inverse, namely $\overline{u}(x)$. Since all elements of $\mathbb{F}_q$ are the reduction modulo $f(x)$ of a polynomial in $\mathbb{F}_p[X]$, then all elements of $\mathbb{F}_q$ are invertible and $\mathbb{F}_q = \mathbb{F}_p[X]/f(x)$ is a field.

Now we propose an equivalence that holds for all polynomial rings over finite fields:

Proposition 5. *In $\mathbb{F}_q[X]$ we have:*

$$x^q - x \equiv \prod_{s \in \mathbb{F}_q} (x - s)$$

Proof. The elements in $\mathbb{F}_q^\times$ form a multiplicative group of order $p^n - 1$, and thus for all elements $s \in \mathbb{F}_q$, $s^{q-1} = 1$, which implies that $s^q - s = 0$. We then know that for all $s \in \mathbb{F}_q^\times$, s is a root of the polynomial $x^q - x$ and clearly $s = 0$ is a root as well, and as there are at most q roots of the degree q polynomial, $x^q - x \equiv \prod_{s \in \mathbb{F}_q} (x - s)$ due to unique factorization of $\mathbb{F}_q[X]$.

□

Corollary 1. *Let $v(x) \in \mathbb{F}_q[X]$, then $v(x)^q - v(x) \equiv \prod_{s \in \mathbb{F}_q} (v(x) - s) \pmod{f(x)}$*

Another property of finite fields we will introduce is a linear mapping property. A linear map is a function $m : X \to Y$, where X and Y are vector spaces, such that for all scalars $a \in \mathbb{F}$ and $x, y \in X$, $m(ax) = am(x)$ and $m(x + y) = m(x) + m(y)$.

Lemma 1. *The function $\delta : \mathbb{F}_q[X] \to \mathbb{F}_q[X]$ defined by $\delta(f(x)) = f(x)^q$ is a linear map and is called the Frobenius Transformation*

Proof. For $f(x), g(x) \in \mathbb{F}_q[X]$,

$$\begin{aligned}(f(x) + g(x))^q =& f(x)^q + \binom{q}{1} f(x)^{q-1} g(x) + \ldots + \binom{q}{q-1} f(x) g(x)^{q-1} + g(x)^q \\ =& f(x)^q + g(x)^q,\end{aligned}$$

as $q | \binom{q}{k}$, for all k such that $1 < k < q$. Given $c \in \mathbb{F}_q$,

$$\delta(cf(x)) = (cf(x))^q = c^q f(x)^q = cf(x)^q,$$

due to Proposition 4, so as it satisfies both properties, additivity and scaling, $\delta(\cdot)$ is a linear map. □

Corollary 2. *If $\gcd(f(x), f'(x)) = 0$, for some non-constant $f(x) \in \mathbb{F}_q[X]$, then $f(x) = g(x)^p$ for some $g(x) \in \mathbb{F}_q[X]$*

Proof. If $\gcd(f(x), f'(x)) = 0$, we have that $f(x) = \sum_{i=0}^{n} a_i x^i$, for some $n \geq 1$, and $a_n \neq 0$, and $f'(x) = \sum_{i=1}^{n} i \cdot a_i x^{i-1}$, as $f(x)$ is non-constant, so $f'(x) = 0$. This implies that $ia_i = 0$ for all i, and as $a_i \neq 0$ for at least some i, then $i = 0$ for such i, which implies that the exponents of the terms of $f(x)$ are divisible by p, and thus $f(x) = g(x^p)$ for some $g(x)$, and by the frobenius mapping property, we have $f(x) = g(x^p) = g(x)^p$. □

Computational Complexity

In this paper, we will be discussing factoring algorithms for polynomials over finite fields. In order to thoroughly discuss the algorithms themselves, we first need to outline the concepts needed to compare and contrast algorithms.

Definition 5. *An algorithm is a set of rules applied to an input with an intended result*

When discussing an algorithm there are three fundamental characteristics that define it: what it does, how it does it, and how "fast" it does it. The speed of a physical implementation of an algorithm will vary greatly between processors and programming languages, but there exists an objective measurement of an algorithm's speed. At the theoretical level we can compare algorithms speed by viewing the required number of operations, e.g. addition, and other binary mathematical operations, and shifts/carrying of values, as a function of the input length. The computational time incorporates all steps needed to perform the algorithm, including the reading of the input.

From this definition of computational time, it is reasonable to discuss the limiting behavior of such a function of the input, as this gives us a perspective on how efficient the algorithm is regardless of our input. We will introduce the following notation to describe the asymptotic bound on an algorithm's computational time:

Definition 6. *Given input $n \in \mathbb{N}$ we say that if for some functions $f(n)$ and $g(n)$, there exists N such that for all $n > N$, $f(n) \leq c \cdot g(n)$ for some constant c, then $f(n) = O(g(n))$ or $f = O(g)$*

Example 1. *If $f(n) = 2n^3 + 3n$, then $f = O(n^3)$*

The notation in this definition is for an arbitrary function that takes integral input, but we will be more specific with our notation and for a function denoting an algorithm's running time we call it $T(n)$.

Many algorithms in mathematics utilize basic binary operations of the integers, so understanding the computational time needed to perform them is important to address. Suppose we wish to add two numbers of length $n \in \mathbb{N}$. Using the standard 'grade school method' of performing addition by adding column wise and carrying, there are n digits to read of each number, n additions being performed and at most n carries, as well as the $n+1$ digit output. All together there are at most $4n+1$ operations being performed, and as reading the inputs takes $2n$ operations, we can conclude that this is the most efficient algorithm for addition and that $T(n) = O(n)$. Likewise, an algorithm for subtraction is almost identically the same procedure, except carrying is replaced with borrowing.

Multiplying, as one may expect, is a slightly more complicated affair. Given two n digit numbers, written in binary, say $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$, the standard multiplication algorithm is based on the fact that

$$ab = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2^{i+j} a_i b_j$$

We will now introduce the algorithm.

Algorithm 2 (Standard multiplication algorithm)**.**

Input: Two n-digit binary numbers, $a = a_{n-1} \dots a_0$, $b = b_{n-1} \dots b_0$, *and* i *is the index of the digits of* a *and likewise* j *is the index of the digits of* b

Output: $a \cdot b$

Procedure:

Step 1: Set $j = 0$, *for* $i \in [0, n-1]$, *calculate* $2^{i+j} a_i b_j$, *and set*

$$(ab)_j = \sum_{i=0}^{n-1} 2^{i+j} a_i b_j$$

Step 2: If $j < n-1$, *go to step 1 and set* $j = j+1$. *If* $j = n-1$ *go to step 3.*

Step 3: Set $ab = \sum_{j=0}^{n-1} (ab)_j$ *and output.*

By iterating through all i for a fixed j and then iterating through i, we can see that there are n^2 single digit multiplications occurring and n^2 numbers being added. A single digit multiplication operation, if we have restricted our numbers

to binary as stated, involves only 0s or 1s, and the only operations involved are checking whether either of the digits are 0. The calculation time for a single digit binary multiplication is then $O(1)$, and thus we have $O(O(1)n^2) = O(n^2)$ operations resulting from the multiplication. Next, we are adding n^2 numbers which takes at most $O(n^2)$ operations, yielding an upper bound of $O(n^2)$ operations to multiply using this method.

A faster algorithm for multiplying is and is known as the Karatsuba algorithm. It utilizes a technique known as 'divide and conquer', in that it breaks up the problem of multiplying two numbers into multiplying several smaller numbers [2, page 295].

Algorithm 3 (Karatsuba algorithm)**.** *Given two $2n$-bit numbers $u = 2^n U_1 + U_0$ and $v = 2^n V_1 + V_0$, where $U_1 = (u_{2n-1}, \ldots, u_n)$ and $U_0 = (u_{n-1}, \ldots, u_0)$ and similarly for V_1 and V_0. Then we have*

$$\begin{aligned} uv &= 2^{2n} U_1 V_1 + 2^n (U_1 V_0 + U_0 V_1) + U_0 V_0 \\ &= (2^{2n} + 2^n) U_1 V_1 + 2^n (U_1 - U_0)(V_0 - V_1) + (2^n + 1) U_0 V_0. \end{aligned}$$

Note we have exchanged multiplying two 2n-bit numbers with three multiplications of two at most n-bit numbers , plus some digit shifts, additions and subtractions. all $O(n)$ operations.

Proposition 6. *The Karatsuba fast multiplication algorithm has computational time $T(n) = O(n^{1.585})$.*

Proof. We have $T(2n) \leq 3T(n) + cn$ for some c, as our $2n$-bit numbers can be multiplied in the same number of operations as three occurrences of two n-bit numbers being multiplied plus some O(n) operations due to shifts and addition/subtraction.

Now assume $T(2^k) \leq c(3^k - 2^k)$.

Given this assumption, we have:

$$\begin{aligned} T(2^{k+1}) \leq 3T(2^k) + 2^k c \leq 3c(3^k - 2^k) + c2^k &= c3^{k+1} - c2^{k+1} \\ &= c(3^{k+1} - 2^{k+1}). \end{aligned}$$

So our hypothesis is true by induction. Now we have for some c,

$$T(n) \leq T(2^{\lceil \log n \rceil}) \leq c(3^{\lceil \log n \rceil} - 2^{\lceil \log n \rceil}) < 3c3^{\log n} = 3cn^{log3}$$

Which implies $T(n) = O(n^{log3}) = O(n^{1.585})$ □

Also of note is that the Karatsuba algorithm's method of 'divide and conquer' can further be extended by considering an rn-bit number broken into r parts and so forth, and using similar manipulations to achieve a larger, set of smaller numbers being multiplied and added. Asymptotically, the computational time of such an modification of the algorithm would be faster, but since the number of shifts and additions would be greater as r increases, it is less feasible than the Karatsuba algorithm.

Since the algorithms we will be introducing are over finite fields, we are more concerned with algorithms for arithmetic on polynomials with coefficients in the finite field $\mathbb{F}_p$ with p elements, or its extension fields. Addition of equivalence classes modulo p can be performed by adding the integer representations and checking if the sum is larger than p, if so, then we subtract a copy of p so it is between 0 and p. If it is less than p, we are done. Each equivalence class representation is at most $\log(p)$ digits long, so this can be done in computational time $O(\log(p))$.

Multiplication can be done by multiplying the two equivalence classes' as if they were integers and then using the same procedure as above i.e. deleting a copy of p so the multiplication result is in $[0, p)$. As both equivalence classes can be input as a number between 0 and p then we know each number is $\log(p)$ digits large at least, then we have the computation time for modular multiplication as being $O(\log(p)^2) = O(\log(p))$ at most. In a field, division can be thought of as multiplication by an inverse element, and we can compute said inverse by Euclid's algorithm and thus it has the same asymptotic computational time as modular multiplication.

Polynomial multiplication over $\mathbb{F}_p$ can be done by viewing n degree polynomials as strings of n elements of $\mathbb{F}_p$. This means that we have, using the basic multiplication method, at most n^2 multiplications of at most $\log(p)$ digit coefficients with some additions and reductions modulo p, which are insignificant given large enough inputs, giving us a computation complexity of $O((\log(p)n^2)$. As there are finitely many operations that can be performed between two representatives modulo p, one can store these outputs in a table given small enough p so we can consider modular arithmetic a constant-time operation for small prime fields and polynomial multiplication over $\mathbb{F}_p$ can be done in $O(n^2)$ time.

As introduced previously, the Euclidean algorithm for calculating the greatest common divisor is an important one over unique factorization rings, and will be used extensively in our factoring algorithms.

Lemma 2. *The Euclidean algorithm between two polynomials $f(x)$, $g(x)$ over a field $\mathbb{F}_p$ has computational time $O(deg(f), deg(g))$, and thus with an upper bound of $O(n^2)$, where $n = \max\{deg(f), deg(g)\}$.*

Proof. Claimed in [2, page 446]. □

The computational complexity of the standard operations discussed are summarized below:

Operation	**Input**	**Output**	**Algorithm**	**Complexity**
Addition	two n-bit #'s	number	standard method	$O(n)$
Multiplication	two n-bit #'s	number	standard method	$O(n^2)$
Multiplcation	two $2n$-bit #'s	number	Karatsuba method	$O(n^{1.585})$
Modular Addition	two #'s modulo p	number modulo p	standard method	$O(\log(p))$
Modular Multiplication	two n-digit #'s modulo p	number modulo p	standard method	$O(\log(p)n^2)$
Gcd in $\mathbb{F}_p[X]$	two degree n polynomials	their gcd	Euclidean algorithm	$O(nm)$

Factoring Algorithms over Finite Fields

We will now introduce two factoring algorithm for polynomials over finite fields. We will focus on the case of $\mathbb{F}_p$, but the computational complexity and methods scale as one would expect to polynomials over $\mathbb{F}_q$.

Efficient algorithms that factor polynomials and integers are important in the fields of computational number theory, algebra, as well as for use in cryptography and data encryption. For example, in the Diffie-Hellman public key-exchange, data can be encoded in large numbers and polynomials over fields, with the "key" to cracking them being finding the irreducible factors of said number and/or polynomial. This is considered a "hard" problem, as in it is difficult to perform the algorithm in a realistic time-frame, particularly for large prime fields and/or high degree polynomials.

The Chinese Remainder Theorem will be utilized in both algorithms and thus is introduced below.

Theorem 1 (Chinese Remainder Theorem)**.** *The theorem states that given a principal ideal domain R, $R/\prod_i I_i \cong R/I_1 \times R/I_2 \times ... \times R/I_n$, where I_i are coprime ideals of R. In other words, there exists an isomorphism between the two structures which can be viewed as the projection of $R/\prod_i I_i$ into the direct product.*

For proof of said theorem,see [1, page 265]

This is a generalization of the well-known theorem in elementary number theory where for $m, n \in \mathbb{Z}$ such that $\gcd(m, n) = 1$,

$$\mathbb{Z}/(mn) \cong \mathbb{Z}/(m) \times \mathbb{Z}/(n).$$

Example 2. *If* $x \equiv 5 \pmod{15}$*, then* $x \equiv 2 \pmod{3}$ *and* $x \equiv 0 \pmod{5}$.

Example 3. $\mathbb{F}_2/(x^3+1) \cong \mathbb{F}_2/(x+1) \times \mathbb{F}_2/(x^2+x+1) \cong \mathbb{F}_2 \times \mathbb{F}_4$ *as* $x^3+1 \equiv (x+1)(x^2+x+1) \pmod{2}$

Berlekamp Algorithm

The Berlekamp algorithm was created by Elwyn Berlekamp in 1967 while he was working as a mathematical researcher for Bell Labs, and the algorithm remained the foremost finite field polynomial factorization algorithm until the Cantor-Zassenhaus algorithm was conceived in 1981. It is a deterministic algorithm, meaning given an input, it will always calculate the same output for that input.

The algorithm itself utilizes the Frobenius linear map mentioned in the previous section and the Chinese Remainder Theorem to factor a monic squarefree polynomial. Say we desire to factor a square-free polynomial $u(x)$ over $\mathbb{F}_p$. We know that given a squarefree polynomial $v(x)$ over a finite field $\mathbb{F}_p$, $v(x)^p = v(x)$ by the Frobenius linear map. The Chinese Remainder Theorem (CRT) says,

$$\mathbb{F}_p[X]/(f(x)) \cong \prod_{i=1}^{n} \mathbb{F}_p[X]/(f_i(x)),$$

where $\prod_{i=1}^{n} f_i(x) = f(x)$ and each $f_i(x)$ is an irreducible factor of $f(x)$. These are the key components that allow this algorithm to work.

First we will present an algorithm to ensure a polynomial is square-free.

Algorithm 4 (Square-free algorithm)**.**

Input: A monic non-zero polynomial in $\mathbb{F}_p[X]$*,* $f(x) = \prod_i f_i(x)^{e_i}$

Output: A square free polynomial in $\mathbb{F}_p[X]$*,* $f_0(x) = \prod_i f_i(x)$.

Step 1: Calculate $\gcd(f(x), f'(x))$.

Step 2: If $\gcd(f(x), f'(x)) = 0$*, then* $f(x) = g(x)^p = g(x^p)$*. Set* $f(x) = g(x)$ *and repeat step 1. If* $\gcd(f(x), f'(x)) = 1$*, set* $f_0(x) = f(x)$ *and the algorithm is done. If* $\gcd(f(x), f'(x)) \neq 1$*, declare* $\gcd(f(x), f'(x)) = d(x)$ *and set* $f_0(x) = \frac{f(x)}{d(x)}$ *and the algorithm is done.*

Once we have run this algorithm, we can run the Berlekamp algorithm on the output:

Algorithm 5 (Berlekamp Algorithm)**.**

Input: A square-free monic polynomial $f(x) \in \mathbb{F}_p[X]$

Output: Irreducibles $f_i(x)$ *with* $deg(f_i(x)) > 0$ *and* $f = \prod_i f_i(x)$

Procedure*:*

Step 1: Generate the Berlekamp matrix $\mathcal{Q}$ *as outlined below.*

Step 2: Use a null-space algorithm to generate a linearly independent basis for the kernel of $v(\mathcal{Q} - I)$ *consisting of* $v^{[1]}, v^{[2]}, \ldots, v^{[r]}$*, where* r *is the rank of the matrix. If* $r = 1$*, our polynomial is irreducible and we are done. Otherwise proceed to step 3.*

Step 3: Set $k = 2$ *and set* $i = 1$*.*

For each $s \in \mathbb{F}_p$*, set* $f_i(x) = gcd(f(x), v^{[k]}(x) - s)$ *if* $gcd(f(x), v^{[k]}(x) - s)$ *is not* 1 *or* $f(x)$ *and set i=i+1 and* $f(x) = \frac{f(x)}{f_i(x)}$ *and repeat step 3 for the next s. Else, repeat step 3 for the next s. If the step has been performed for all s, then go to step 4. If at any point* $f(x) = 1$*, then terminate.*

Step 4: Repeat step 3 replacing k *with* $k + 1$ *if* $k < r$*.*

The $\mathcal{Q}$ matrix mentioned in the algorithm is defined as follows:

Definition 7. *For a polynomial* $f(x) \in \mathbb{F}_p$*,the Berlekamp matrix Q is*

$$\mathcal{Q} = \begin{pmatrix} q_{0,0} & q_{0,1} & \cdots & q_{1,n} \\ q_{1,0} & a_{1,1} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ q_{n,1} & q_{n,2} & \cdots & q_{n-1,n-1} \end{pmatrix}$$

Where $x^{pk} \equiv q_{0,k} + q_{1,k}x + \ldots + q_{n-1,k}x^{n-1} \ mod\ f(x)$

Theorem 2. *The Berlekamp algorithm gives the factorization of a squarefree monic polynomial over* $\mathbb{F}_p$ *into irreducibles.*

Proof. Assume we have reduced our polynomial to a squarefree monic polynomial of degree n using our squarefree algorithm.

Given a polynomial $f(x) \in \mathbb{F}_p[X]$ such that $f(x) = \prod_{i=1}^{r} f_i(x)$ and an r-tuple $(s_1, ..., s_r)$, where $s_i \in \mathbb{F}_p$, there exists a unique $v(x)$ such that

$$v(x) \equiv s_i \pmod{f_i(x)}$$

by the Chinese Remainder Theorem. We also know that

$$v(x)^p \equiv s_i^p \equiv s_i \equiv v(x) \pmod{f_i(x)}$$

for all i since $s \in \mathbb{F}_p$, so such $v(x)$ satisfy

$$v(x)^p \equiv v(x) \bmod f(x).$$

Thus we can see that $f(x) | \prod_{s \in \mathbb{F}_p} (v(x) - s) = v(x)^p - v(x)$ so some irreducible factor of $f_i(x)$ divides a term of the form $v(x) - s$, and we know that the $\gcd(v(x) - s_i, v(x) - s_j) = 1$ for $i \neq j$, so each irreducible factor divides a *unique* factor of the form $v(x) - s$ if $i \neq j$. Thus if we can calculate the set

$$V = \{v(x) | v(x)^p \equiv v(x) \bmod f(x)\},$$

or furthermore a basis of V we can factor $u(x)$. Of note is the fact that the set V is a vector space as it closed under addition as shown by the Frobenius map and is closed under scaling.

We define the matrix $\mathcal{Q}$ by representing each pth multiple power of x as previously stated.

Consider

$$v(x) = v_{n-1}x^{n-1} + v_{n-2}x^{n-2} + \ldots + v_0$$

as a vector $v = (v_0, \ldots, v_{n-2}, v_{n-1})$, then $V = \{v : vQ = v\}$, because $v(x) \equiv v(x^p) \equiv v(x)^p \bmod f(x)$,

$$\implies \sum_i v_i x^i \equiv \sum_i v_i x^{pi} \bmod f(x)$$

$$\implies \sum_i v_i x^i \equiv \sum_i v_i (\sum_k q_{i,k} x^k) \bmod f(x)$$

$$\implies x^i \equiv \sum_k q_{i,k} x^k \bmod f(x) \implies v_i = \sum_k q_{i,k} v_k.$$

Which is equivalent to $v\mathcal{Q} = v$, so the solution set of this equality provides all such v we want.

We can now use a null space algorithm on $v(\mathcal{Q} - I) = 0$ as in step 3 to solve for linearly independent vectors over $F_p[X]$: $v^{[1]}, v^{[2]}, \ldots, v^{[r]}$, where r will be the number of irreducible factors of $u(x)$. Each of these can be represented as a polynomial like above. Thus, the set

$$v(x) = t_1 v^{[1]} + t_2 v^{[2]} + \ldots + t_n v^{[r]}$$

are all the solutions to $v(\mathcal{Q} - I) = 0$, where we take $v^{[1]} = (1, 0, \ldots, 0)$, the basis of all trivial $v(x) \in \mathbb{F}_p$. Since there are only p^r possible solutions, due to how we defined $v(x)$, these are all of them.

We now use the equality

$$f(x) = \prod_{s \in \mathbb{F}_p} \gcd(f(x), v(x) - s), \ \forall i \in \{1, \ldots, r\}, v(x) \in V \backslash \mathbb{F}_p$$

to find the irreducible factors of f.

This is an equality as

$$\forall s \in \mathbb{F}_p, \ \gcd(f(x), v(x) - s) | f(x) \implies \prod_{s \in \mathbb{F}_p} (\gcd(f(x), v(x) - s) | f(x)$$

and as shown earlier, for each $f_i(x)$ of $f(x)$, there exists a unique $v(x) - s_i$ such that $f_i(x) | (v(x) - s_i)$, and clearly $f_i(x) | f(x)$, so

$$f(x) | \prod_{s \in \mathbb{F}_p} \gcd(f(x), v(x) - s) \implies f(x) = \prod_{s \in \mathbb{F}_p} \gcd(f(x), v(x) - s)$$

Thus, knowing this, we can take each basis vector $v^{[i]}$, starting at $i = 2$ and calculate

$$f(x) = \prod_{s \in \mathbb{F}_p} \gcd(f(x), v(x)^{[i]} - s).$$

If an irreducible factor is found (other than 1 or $f(x)$), then we can divide $f(x)$ by it and continue the process with each of of the basis vectors $v^{[i]}$ for $2 \geq r$. Eventually, we will have factorized the whole polynomial, and we'll know when we have r non-trivial factors. $\square$

Theorem 3. *The computational complexity of the Berlekamp factoring algorithm is* $O(n^3 + rpn^2)$.

Proof. Note that as stated earlier, calculating the gcd of two polynomials is $O(n^2)$, so calculating whether the polynomial is squarefree and dividing it so takes $O(n^2)$ operations. Additionally, checking if $f(x) = g(x)^p$ takes linear time. In order to create the vectors of matrix, we have p divisions in the field, each at $O(n^2)$ in total taking $O(pn^2)$ operations. Calculating the basis of the $\mathcal{Q}$ matrix using Gaussian elimination costs $O(n^3)$ [2, page 446] operations. Lastly we have r basis vectors and p field elements to iterate through in step 4, which yields at most $O(rpn^2)$ operations.

In total this takes $O(n^3 + kpn^2 + p^n 2) = O(n^3 + rpn^2)$ operations.

□

Computational Example

Below the polynomial $r(x) = x^8 + x^6 + x^5 + x^3 + x^2 + x + 1's$ square-free part over $\mathbb{F}_2[X]$ will be factored into irreducible factors.

First we perform our square-free algorithm.

$\gcd(r(x), r'(x)) = \gcd(x^8+x^6+x^5+x^3+x^2+x+1, x^4+x^2+1) = x^4+x^2+1 = (x^2+x+1)^2$

So our square-free polynomial is now

$$u(x) = \frac{x^8 + x^6 + x^5 + x^3 + x^2 + x + 1}{x^2 + x + 1} = x^6 + x^5 + x^4 + x^3 + 1$$

(Note that we only divide by x^2+x+1, as we are in characteristic two, so we have the case of $f'(x) = g'(x)h(x)^2 + 2g(x)h(x) = g'(x)h(x)^2$, so $\gcd(f(x), f'(x)) = h(x)^2$.)

Step 1: For our polynomial $u(x)$ we generate the Berlekamp matrix as outlined in the algorithm.

$$\mathcal{Q} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Step 2: Generate a basis for the kernel of $\mathcal{Q} - I$

Basis=$\{1, x^4 + x^2\}$, so $v^{[1]}(x) = 1$ and $v^{[2]}(x) = x^4 + x$.

Step 3: Start with the non-trivial basis element represented as a polynomial, $v^{[2]}(x)$

For $s = 0$, $\gcd(x^6 + x^5 + x^4 + x^3 + 1, x^4 + x) = 1$

For $s = 1$, $\gcd(x^6 + x^5 + x^4 + x^3 + 1, x^4 + x + 1) = x^4 + x + 1$

So $x^4 + x + 1$ is an irreducible factor of $r(x)$ and

$$\frac{x^6 + x^5 + x^4 + x^3 + 1}{x^2 + x + 1} = x^2 + x + 1$$

We know this irreducible as our iterations through the basis are completed, so

$$u(x) = (x^4 + x + 1)(x^2 + x + 1)$$

as irreducible factors and we are done. If we wish to calculate the factorization of our original polynomial $r(x)$, we can simply multiply by $x^2 + x + 1$, what we divided out by, as it is irreducible as shown by the algorithm, and

$$r(x) = (x^4 + x + 1)(x^2 + x + 1)^2.$$

Cantor-Zassenhaus Algorithm

We will now introduce an alternative factoring algorithm for polynomials over finite fields, the Cantor-Zassenhaus Algorithm. This algorithm was created by David Cantor and Hans Zassenhaus in 1981 and is still widely implemented alongside the Berlekamp algorithm. A fundamental difference between the two algorithms is that the Berlekamp algorithm is deterministic. This means that given any polynomial it will provide the unique factorization eventually, regardless of how many steps it takes. The CZ algorithm on the other hand is probabilistic, in that it can effectively fail to factor a polynomial completely. The tradeoff is that probabilistic factoring algorithms tend to utilize tricks that enable them to perform tasks quicker if they do succeed. First, we must introduce the distinct degree factorization algorithm.

Algorithm 6. *Distinct Degree Factorization Algorithm (DDF)*

Input: A polynomial $g(x) \in \mathbb{F}_p[X]$ *of degree* m

Output: Polynomials $g_i(x)$ *equal to the multiple of all irreducible factors of a particular degree* i*, such that* $g_1(x) \dots g_n(x) = g(x)$ ***Procedure:***

Step 1: Check to see if $g(x)$ *is square-free. If not, then reduce it to a square-free case by setting*

$$g(x) = \frac{g(x)}{\gcd(g(x), g'(x))}$$

and store $\gcd(g(x), g'(x))$.

Step 2: Set $i = 1$ *and calculate* $x^{p^i} - x \bmod f(x)$ *using a repeated squaring algorithm (described below).*

Step 3: Set $g_i(x) = gcd(x^{p^i} - x, g(x))$*. Set* $g(x) = \frac{f(x)}{g_i(x)}$*. If* $deg(g(x)) = 0$*, terminate. If* $deg(g(x)) > 0$*, go to step 3.*

Step 3: Set $i = i + 1$ *and repeat step 2.*

Proposition 7. *This algorithm is correct in its assertions.*

Proof. This is valid due to the fact that $x^{p^m} - x$ is the product of all irreducible polynomials over $\mathbb{F}_p$ of degree $d|m$, which we will prove now by showing that all roots of each are roots of the other.

The roots of $x^{p^m} - x$ are the elements of $\mathbb{F}_{p^m}$ by proposition 5. Consider an element $\alpha \in \mathbb{F}_{p^m}$ and the extension field of $\mathbb{F}_p$, $\mathbb{F}_p(\alpha)$, then

$$[\mathbb{F}_{p^m} : \mathbb{F}_p] = [\mathbb{F}_{p^m} : \mathbb{F}_p(\alpha)][\mathbb{F}_p(\alpha) : \mathbb{F}_p].$$

Consider the minimal irreducible polynomial of α over $\mathbb{F}_p$ with degree d, then $[\mathbb{F}_p(\alpha) : \mathbb{F}_p] = d$ and from above $m = kd$, for some k which implies $d|m$.

Given a root of some irreducible polynomial $f(x)$ of degree $d|m$ over $\mathbb{F}_p$, then $\mathbb{F}_{p^d}$ is a subfield of $\mathbb{F}_{p^m}$, and thus the root is in $\mathbb{F}_{p^m}$ and thus is a root of $x^{p^m} - x$ by proposition 5.

So the statement is true and we have that $g_i(x) = \gcd(x^{p^i} - x, g(x))$ is the product of all irreducible polynomial factors of degree i of $g(x)$. We can iterate through all i such that $1 \leq i \leq m$ and divide $g(x)$ progressively to distinct degree factorize our polynomial. □

In order to compute $\gcd(x^{p^i} - x, f(x))$, note that

$$\gcd(x^{p^i} - x, f(x)) = \gcd(x^{p^i} - x \bmod f(x), f(x)).$$

This can be computationally intensive if p or m is sufficiently large. We will use a repeated squaring method to remedy this described below.

Algorithm 7. *Repeated squaring algorithm*

Input: $p^k = \sum_{i=1}^{l} b_i 2^i$, *a binary representation of* p, *where* $l = \lceil \log(p) \rceil$.

Output: $x^{p^l} \bmod f(x)$

Step 1: Set $a = x^{b_0}$, $a_0 = x$ *and* $i = 1$.

Step 2: Set $a_i = a_{i-1}^2 \bmod f(x)$. *If* $b_i = 1$, *then* $a = a_i \cdot a$.

Step 3: If $i < k$, *set* $i = i + 1$ *and go to step 2. If* $i = k$, *end algorithm.*

We want to calculate $x^{p^i} \bmod f(x)$, for each i. By reducing modulo $f(x)$ after each squaring, we can reduce how long each computation takes and prevent them from "piling" up and having to reduce many times at the end.

Then, in implementation, if we start with $i = 0$, we perform this process for each i and calculate $\gcd(f(x), x^{p^i} - x \bmod f(x))$. In the algorithm, it also only multiples by the x^{2^i} for some i if necessary, so we calculate x^{2^k} in an efficient manner. We must perform $\log(p)$ exponentiations, our indices go from $i = 1$ to k, and for each index we calculate the gcd at a cost of $O(n^2)$, yielding a total computational cost of $O(\log(p)n^2)$.

Now we can see that the computational complexity of the DDF algorithm for a polynomial of degree n is $O(\log(p)n^3)$. To calculate $gcd(x^{p^i} - x$ for a given power of i, it costs $O(\log(p)n^2)$. We have to perform at most n loops through the DDF algorithm, so that makes $O(\log(p)n^3)$ operations at most in total.

Algorithm 8. *Cantor-Zassenhaus Algorithm*

Input: A monic squarefree polynomial $g(x) \in \mathbb{F}_p[X]$ *of degree* m

Output: Irreducibles $h_i(x)$ *with* $deg(h_i(x)) > 0$ *and* $f(x) = \prod_i h_i(x)$

Step 1:For a given polynomial $g(x) \in \mathbb{F}_p[X]$ *of degree* m*, run DDF on* $g(x)$ *and denote the distinct degree factors as* $g_i(x)$ *with distinct degree* i*.*

Step 2: For each $g_i(X)$*, Pick a polynomial* $r(X) \in \mathbb{F}_p[X]$ *at random with degree less than* i*. Repeat the following steps for each* $g_i(x)$*.*

Step 3: Set $d = \frac{p^i-1}{2}$*.*

Step 4:Calculate and output $gcd(r(X)^d + 1, g_i(X))$ *and divide* $g_i(X)$ *by it, yielding a new* $g_i(X)$*.*

Step 5: Repeat steps 2 through 4 with different random r(X) until $d_i(X) = 1$*, while recording each irreducible factor.*

Step 6: Set $i = i + 1$ *and loop steps 2 through 5 until all distinct degree factorizations have been exhausted. output each DDF's factors; this the is the factorization of* f*.*

The steps after step 1 use the fact the Chinese Remainder Theorem to represent

$$R = \mathbb{F}_p[X]/g_i(x) = \mathbb{F}_p[X]/f_1(x) \times \ldots \times \mathbb{F}_p[X]/f_n(x) = (\mathbb{F}_{p^i})^n$$

where $g_i(x) = \prod_{j=1}^{n} f_j(x)$ and each $f_j(x)$ has degree i.

Theorem 4. *The Cantor-Zassenhaus algorithm factorizes a squarefree polyonomial over* $\mathbb{F}_p[X]$ *into irreducibles.*

Proof. If we know that a non-trivial polynomial is congruent to 0 in one of the fields in the direct product, then we know that polynomial is congruent to 0 modulo $g_i(x)$ and is divisible by $g_i(x)$.

Since $\mathbb{F}_{p^i}^{\times}$ is an abelian group of order p^i, then for all elements a of the group, $a^{p^i-1} = 1$. Half the elements $a \in \mathbb{F}_{p^i}^{\times}$ have order $\frac{p^i-1}{2}$ i.e. $a^{\frac{p^i-1}{2}} = -1$, so if we pick a random non-trivial element in $\mathbb{F}_p[X]$, there's a $1/2$ probability that it has this order when reduced modulo $f_j(x)$, which we will call $m = \frac{p^i-1}{2}$. Thus given

a random non-zero element $r(x) \in \mathbb{F}_p[X]$, $r(x)^m + 1$ is 0 when reduce modulo $f_i(x)$ with half probability and likewise $r(x)^m + 1$ is 2 with half probability. So when considering all of the n fields in the cartesian product, we are looking for a non-trivial zero divisor such that $r(x)^m + 1$ is 0 in some coordinate of the n-tuple under the isomorphism, but not all of them as that implies that it is divisible by $g_i(x)$.

The probability of success for this algorithm is thus $1 - \frac{1}{2^d} - \frac{1}{2^d} = 1 - \frac{1}{2^{d-1}}$, as the probability of a coordinate being 0 or 2 is independent and thus there is a $\frac{1}{2^d}$ chance that the n-tuple is all 0s or all 2s when evaluated at $r(x)^m + 1$. Repeating the process for each distinct degree factor of our polynomial will yield a complete factorization. □

Theorem 5. *The computational complexity of the Cantor-Zassenhaus algorithm is* $O(\log(p)n^3)$.

Proof. As shown earlier, the computational complexity of the distinct degree factorization algorithm on a n degree polynomial is $O(\log(p)n^3)$. Next, for a given $r(x)$, we have to exponentiate it to a power of p and calculate the $\gcd(r(x)^m + 1, f(x))$ taking $O(\log(p)n^2)$ operations, and we have at most n irreducible factors so a rough upper bound is we have to perform step 4 n times, as with sufficiently large p the probability that the random element is a zero divisor is ~ 1, giving us $O(\log(p)n^3)$ operations for all the iterations of this step as well. All other steps take $O(n^3)$ or less computations, independent of p, so we have a computational complexity of $O(\log(p)n^3)$. □

Computational Example

We will factor $f(x) = x^7 + 2x^5 + x^3 + 2x \in \mathbb{F}_3[X]$ into irreducibles using the Cantor-Zassenhaus algorithm.

Step 1:

First we perform the DDF algorithm on $f(x)$ and checking if it is squarefree, as in the Berlekamp algorithm.

$\gcd(f(x), f'(x)) = \gcd(x^7 + 2x^5 + x^3 + 2x, x^6 + x^4 + 2) = 1$, so f is squarefree and we can loop through the splitting process of the DDF algorithm.

$i = 1$, $\gcd(f(x), x^3 - x) = x^3 + 2x = f_1(x)$ and

$$g(x) = \frac{f(x)}{x^3 - x} = x^4 + 1$$

$i = 2$, $\gcd(g(x), x^9 - x) = x^4 + 1 = f_2(x)$

Step 2: For $f_1(x)$ we generate random polynomial with degree 1, $r(x) = x + 2$ with $m = 1$.

$\gcd(f_1(x), x) = x$, so set $f_1(x) = x^2 + 2$, and now $f(x) = xf_1(x)f_2(x)$.

Generate another random polynomial with degree 1, $r(x) = x+1$, and $\gcd(f_1(x), x+1) = x + 1$ and now we know that $f(x) = (x)(x + 1)(x + 2)f_2(x)$ as we divide $f_1(x)$ by $x + 1$ to receive our last linear factor.

Step 3: For $f_2(x)$ we generate random polynomial with degree less than 3, $r(x) = x^2$ with $m = (3^2 - 1)/2 = 4$.

$\gcd(f_2(x), x^8 + 1) = 1$ so we must generate another random polynomial $r(x) = x^2 + x + 2$ and $r(x)^4 + 1 = x^8 + x^7 + 2 * x^6 + x^5 + x^4 + 2x^3 + 2x^2 + 2x + 2$

$\gcd(f_2(x), x^8 + x^7 + 2 * x^6 + x^5 + x^4 + 2x^3 + 2x^2 + 2x + 2) = x^2 + 2x + 2$ and $\frac{x^4+1}{x^2+2x+2} = x^2 + x + 2$ and since these are both degree 2 polynomials we are done and $f(x) = x(x + 1)(x + 2)(x^2 + x + 2)(x^2 + 2x + 2)$.

Lattice Factorization Algorithm

Now we will present an algorithm for factoring monic polynomials in $\mathbb{Z}[X]$. Factorization of polynomials over the integers is an inherently more difficult problem than factorization over finite fields, as the integers are not a field and thus the factorization of the coefficients of a polynomial is non-trivial and in fact computationally difficult. Also, given a polynomial of degree n over a finite field of size q, there are q^n polynomials, while there are a countably infinite number of polynomials over the integers. In order to have a chance at conquering this problem in a reasonable amount of time, one must find a way to limit the number of possible irreducible polynomials to search through as potential factors. This is accomplished through a clever technique created by A.K. Lenstra in 1981 utilizing lattices and the Berlekamp algorithm in conjunction with Hensel's Lifting Lemma [3].

In order to introduce the algorithm, first we must introduce the lattice and it's determinant :

Definition 8. *A lattice L is a subset of $\mathbb{R}^n$ generated by a basis $\mathcal{B}$ of $\mathbb{R}^n$ and its linear combinations with integer coefficients:*

$$L = \{\sum_{i=0}^{n} a_i b_i | a_i \in \mathbb{Z}, b_i \in \mathcal{B}\},$$

and its determinant is the absolute value of the determinant of the bases vectors:

$$\det(L) = |det(\mathcal{B})|$$

As is well known, the determinant of a basis is the volume of the *fundamental region*,

$$F = \{\sum_{i=0}^{n} a_i b_i | a_i \in [0,1), b_i \in \mathcal{B}\},$$

i.e. $\det(L) = \text{vol}(F)$, so if one lattice L is a subset of another lattice L', then the fundamental region of L contains that of L', thus $\det(L') \leq \det(L)$.

An important result we will use relating our the determinant of a matrix to its vectors is as follows:

Theorem 6. *Hadamard's Inequality*

For a square matrix A with set of linearly independent column vectors $\mathcal{B}$,

$$|\det(A)| \leq \prod_{b_i \in \mathcal{B}} \|b_i\|$$

Proof. By dividing the matrix A by the product of all of its vector lengths, $\prod_{b_i \in \mathcal{B}} \|b_i\|$, we get a matrix U, which is made up of vectors all of length one. Since $|\det(U)| \leq 1$ through a geometric argument, as the maximal case is that the vectors of U are orthogonal i.e. they form an orthonormal basis, then we have :

$$|\det(A)| \leq \prod_{b_i \in \mathcal{B}} \|b_i\| \cdot |\det(U)| \leq \prod_{b_i \in \mathcal{B}} \|b_i\|.$$

□

Another piece of machinery we will introduce that is used in the lattice factorization algorithm is Hensel's lifting lemma.

Theorem 7 (Hensel's lifting lemma)**.** *Given a monic polynomial $f(x) \in \mathbb{Z}[X]$ such that $f(x) \equiv h_k(x)g_k(x) \pmod{p^k}$ and there exists $u(x), v(x) \in \mathbb{Z}[X]$ such that*

$$g_k(x)u(x) + h_k(x)v(x) \equiv 1 \pmod{p},$$

then there exists unique $g_{k+1}(x)$, $h_{k+1}(x)$ where:

$$g_{k+1} \equiv g_k(x) \pmod{p^k},$$

$$h_{k+1} \equiv h_k(x) \pmod{p^k},$$

such that

$$f(x) \equiv g_{k+1}(x)h_{k+1}(x) \pmod{p^{k+1}}$$

Proof. We seek are seeking $g_{k+1}(x)$ and $h_{k+1}(x)$ such that

$$g_{k+1}(x) = g_k(x) + c_k(x)p^k,$$

and

$$h_{k+1}(x) = h_k(x) + d_k(x)p^k,$$

for some $d_k(x), c_k(x)$. This implies that

$$\begin{aligned} f(x) &\equiv (g_k(x) + c_k(x)p^k)(h_k(x) + d_k(x)p^k) \pmod{p^{k+1}} \\ &\equiv g_k(x)h_k(x) + p^k(h_k(x)c_k(x) + g_k(x)d_k(x)) + c_k(x)d_k(x)p^{2k} \pmod{p^{k+1}} \\ &\equiv g_k(x)h_k(x) + p^k(h_k(x)c_k(x) + g_k(x)d_k(x)) \pmod{p^{k+1}} \end{aligned}$$

And by assumption, $f(x) - h_k(x)g_k(x) = l(x)p^k$ for some $l(x)$, so we have that

$$l(x)p^k \equiv p^k(h_k(x)c_k(x) + g_k(x)d_k(x)) \pmod{p^{k+1}}$$

, and thus,

$$l(x) \equiv h_k(x)c_k(x) + g_k(x)d_k(x) \pmod{p}.$$

Since by assumption $\gcd(h_k(x), g_k(x)) = 1 \pmod{p}$, we have $l(x)|h_k(x)$ and $l(x)|g_k(x) \pmod{p}$, and $h_k(x) = \overline{c}(x)l(x)$ and $g_k(x) = \overline{d}(x)l(x)$ for some $\overline{c}(x)$, $\overline{d}(x)$. This does not guarantee uniqueness of g_{k+1} and h_{k+1}, but if we take $c(x)$ such that $d(x)$ such that $\overline{c}(x) = h_k(x)q(x) + c(x)$ and $\overline{d}(x) = g_k(x)p(x) + d(x)$, then by the division algorithm, these remainders are unique and thus we have found our g_{k+1} and h_{k+1}.

□

This theorem says that we can extend a polynomial from a factorization over $\mathbb{F}_p[X]$ to one over $\mathbb{Z}/(p^k\mathbb{Z})[X]$ uniquely and that it keeps the equivalence relations we would expect for lesser powers of p.

In order to factor polynomials over the integers, we will use the fact that we can reduce and factor a polynomial $f(x) \in \mathbb{Z}[X]$ of degree n modulo some prime p using the Berlekamp algorithm. This gives us some information about the roots of the polynomial over $\mathbb{Z}$, and doubly functions as a test for irreducibility as if it is irreducible over $\mathbb{F}_p$, then it irreducible over $\mathbb{Z}$. We then choose an irreducible factor h of degree l in $\mathbb{F}_p[X]$ and uniquely lift the irreducible factors to h_k in the ring $\mathbb{Z}/(p^k)\mathbb{Z}[X]$. We will determine said value of k by use of the following theorem:

Theorem 8. *Given two relatively prime monic polynomials $f, h \in \mathbb{Z}[X]$ with $deg(f) = n_1$, $deg(h) = n_2$ and $n_1 \geq n_2$, such that h and f have a common divisor $h_k \in \mathbb{Z}/(p^k\mathbb{Z})[X]$, where $1 \leq deg(h_k) = n \leq n_2$, then $p^{kn} \leq \|f\|^{n_2}\|h\|^{n_1}$.*

Proof. Consider the collection of polynomials viewed as vectors,

$$\begin{aligned} v_i &= f \cdot x^i, \qquad 0 \leq i \leq n_2 - 1, \\ v_i &= h \cdot x^{i-n_2}, \qquad n_2 \leq i \leq n_1 + n_2 - 1, \end{aligned}$$

where the coefficient of the j-th power of x is the j-th coordinate in each vector v_i.

First note that our collection of vectors is linearly independent:

If

$$\sum_{i=0}^{n_1+n_2-1} a_i v_i = 0,$$

then

$$\sum_{i=0}^{n_2-1} a_i x^i \cdot f + \sum_{i=n_2}^{n_1+n_2-1} a_i x^i \cdot h = 0$$

which implies that

$$uf + vh = 0$$

for some polynomials $u, v \in \mathbb{Z}[X]$. Since $\gcd(f, h) = 1$, then $u = 0$, $v = 0$ which implies $a_i = 0$ for all i, and thus they are linearly independent. As there are $n_1 + n_2$ vectors, then this collection is a basis for $\mathbb{R}^{n_1+n_2}$ and we define our lattice L as the linear combinations with integer coefficients of the above set of vectors.

Via Hadamard's inequality, we know that $\det(L) \leq \|f\|^{n_2}\|h\|^{n_1}$. Now consider the following polynomials viewed as vectors:

$$\begin{aligned} b_i &= p^k \cdot x^i, \qquad 0 \leq i \leq n-1, \\ b_i &= h_k \cdot x^{i-n}, \qquad n \leq i \leq n_1 + n_2 - 1, \end{aligned}$$

Again, note that this collection of vectors is linearly independent:

If

$$\sum_{i=0}^{n_1+n_2-1} a_i v_i = 0,$$

then

$$\sum_{i=0}^{n_1-1} a_i p^k x^i + \sum_{i=n_2}^{n_1+n_2-1} a_i x^i \cdot h_k = 0.$$

If we reduce the equation modulo p^k we have that

$$h_k\Big(\sum_{i=n_2}^{n_1+n_2-1} a_i x^i\Big) = 0,$$

and since h_k is a non-trivial divisor of f and g, then we know that all a_i are 0 and the vectors are linearly independent. As there are $n_1 + n_2$ vectors, then this is also a basis for $\mathbb{R}^{n_1+n_2}$ and we can consider the lattice L_k generated by these vectors. Observe this lattice consists of vector representations of all polynomials in $\mathbb{Z}/(p^k\mathbb{Z})[X]$ divisible by h_k of degree less than or equal to $n_1 + n_2 - 1$. There are $n_1 + n_2 - 1$ vectors in L^k, and reduction modulo p^k of all b_i for $0 \leq i \leq n-1$ yields 0, which is trivially divisible by h_k modulo p^k. All polynomials of the form $h_k \cdot x^{i-n}$ are also divisible by h_k, so they are all divisible by h_k and thus this is all of them. $L \subseteq L_k$ as f and h are both divisible by h_k modulo p^k, and thus all the vector representations in L are also in L_k. If we represent the basis of our lattice L_k as a matrix, one can see that since h_k is monic, then our matrix is upper triangular and $\det(L_k) = p^k$ and since $L \subseteq L_k$, it follows that $p^k = \det(L_k) \leq d(L) \leq \|f\|^{n_2}\|h\|^{n_1}$ and we are done. $\square$

Using this theorem, through the contrapositive we know that an irreducible polynomial $g \in \mathbb{Z}[X]$ divides f if $p^{kn} > \|f\|^{n_2}\|g\|^{n_1}$ for some sufficiently large k and $g|f$ modulo p^k. In order to find said k, assume that $\deg(f) = n$ and $\deg(g) = m$, and due to an inequality proved by Mignotte [4, pages 1153-1157], if $g|f$, then $|g_i| \leq \binom{m}{i}\|f\|$ for all $i \in \{0, \ldots, m\}$, so $\|g\| \leq \sqrt{\binom{2m}{m}}\|f\| = B$.Note that in our algorithm below, namely step 7, we take $\deg(g) = n - 1$, in order to ensure that our k value is sufficiently large, though one could optimize said value of m in order to improve performance.

Since $\|g\|^n\|f\|^m \leq B^{2m}$, then we can choose k such that $B^{2m} < p^{kn}$. Every polynomial in L_k that doesn't divide f has length greater than B, so g is the shortest vector in L_k and we can find our irreducible factor of f by using a shortest vector algorithm.

To find our potential factors of f and construct our lattice L_k, we use the Berlekamp factoring algorithm to find our polynomials $h \in \mathbb{F}_p[X]$ with which we can lift to our sufficiently large k with respect to our bound B outlined above. By Hensel's lifting lemma, there exists a unique h_k such that $h|f \in \mathbb{F}_p[X] \implies h_k|f(\mathbb{Z}/p^k\mathbb{Z})[X]$, so by lifting h we can find our h_k such that we can construct L_k. To choose our prime p, we will choose the smallest prime dividing the determinant of f. Then the last step would be to find said shortest vector as above.

However, there does not exist a polynomial time i.e. fast, algorithm to calculate the shortest vector in a lattice, but we can find a vector that is short enough in polynomial time via the LLL algorithm, introduced in 1982 by A.K. Lenstra, H.W. Lenstra Jr., and L. Lovász [5]. This algorithm takes the basis for a lattice L and creates an orthogonal basis from it through the Graham-Schmidt process, and then uses an iterative procedure that yields a δ-LLL reduced basis for the lattice such that b_{i+1} is at most some constrained factor, related to the chosen $\delta \in (1/4, 1)$, longer than b_i for all i. In addition, given a reduced basis, then

$$\|b_1\| \leq (\frac{2}{\sqrt{4\delta - 1}})^{n-1} \cdot \lambda(L),$$

which says that the length of the first vector in our reduced basis is bounded above by a factor dependent on the chosen δ and n, the rank of the lattice, times the shortest vector in the lattice ($\lambda(L)$). In addition, the factoring algorithm utilizing the LLL algorithm uses the following relation that holds for all linearly independent set of vectors $x_1, \ldots, x_t \in L$, and any b_j in the reduced basis such that $1 \leq j \leq t$:

$$\|b_j\| \leq (\frac{2}{\sqrt{4\delta - 1}})^{n-1} \max(\|x_1\|, \ldots, \|x_t\|).$$

In practice, one chooses $\delta = \frac{3}{4}$ as a convention, and thus this becomes

$$\|b_j\| \leq 2^{\frac{n-1}{2}} \max(\|x_1\|, \ldots, \|x_t\|).$$

Our calculation of k in Step 7 below is due to the immediately above inequality combined with $\|g\|^n\|f\|^{n-1} \leq B^{2(n-1)}$.

Step 11 is the key to finding our irreducible factor h_0 of polynomial f over the integers corresponding to our h over $\mathbb{F}_p$ from the Berlekamp algorithm. We know that each b_i for $1 \leq i \leq j$ satisfies the contrapositive of inequality in theorem 8, so for such b_i, h_0 divides it, as $h_k|h_0$ modulo p^k, thus $h_0|\gcd(b_1, b_2, \ldots b_j)$. In order to show that it is indeed an equality, refer to [5, Prop 2.16]; it is an argument by degree of said gcd and h_0, and the primitivity of the gcd so that they are identical.

Below we will outline the factorization algorithm:

Algorithm 9. *Lattice Factorization Algorithm*

Input: A monic polynomial $f(x) \in \mathbb{Z}[X]$ of degree n

Output; Irreducible monic polynomials $h_1, h_2, \ldots \in \mathbb{Z}[X]$ such that $\prod_i h_i = f$

Step 1: $r = discriminant(f)$

Step 2: if $r = 0$, $f = \frac{f}{f'}$ and go to Step 1

Step 3: Else, choose p as the least prime that doesn't divide r

Step 4 Factor f via the Berlekamp algorithm into irreducibles h over $\mathbb{F}_p$

Step 5: If there exists h such that $h \equiv f \pmod{p}$, output f

Step 6: Else, pick an h from Step 4 and denote $l = deg(h)$

Step 7: $k = \lceil log_p(2^{\frac{n(n-1)}{2}}\binom{2(n-1)}{n-1}^{\frac{n}{2}}\|f\|^{2n-1})(1/l)\rceil$

Step 8: Perform Hensel's lift reiteratively on h to get $h_k \in \mathbb{Z}/(p^k)\mathbb{Z}[X]$

Step 9: Construct the basis of L_k and perform the LLL algorithm an denote the new reduced basis $L'_k = \{b_1, \ldots, b_n\}$

Step 10: $j =$ greatest integer such that $|b_j| < (p^{kl}/\|f\|^{n-1})^{\frac{1}{n}}$, $\forall b_i$ in the basis of L'_k

Step 11: $h_0 = \gcd(b_1, b_2, \ldots b_j)$

Step 12: $f = f/h_0$. output h_0. If $f = f/h_0 = 1$, done. Else, $f = f/h_0$ and return to step 5 picking another h

In order for this algorithm to be efficient, it needs a fast way to perform a lattice reduction so that one can approximate the shortest vector in a lattice.

Computational Example

Though explaining the algorithm does provide insight to how it actually works, it is much more illuminating when an explicit example is calculated. Below, we will factor the sixth degree polynomial $x^6 + 5x^4 - 2x^3 + 6x^2 - 8x - 8$ over $\mathbb{Z}[X]$.

Step 1 & 3:

$discriminant(f) = 5394456576 = 2^{13} * 3^3 * 29^3 \implies$ prime $p = 5$

Step 4 & 6:

$x^6 + 5x^4 - 2x^3 + 6x^2 - 8x - 8 \equiv (x^3 + 2x + 1)(x^3 + 3x + 2) \pmod 5$.

Choose $h = x^3 + 3x + 2 \implies l = \deg(h) = 3$

Step 7:

$k = \lceil \log_5(6^{15} * 252^3(\sqrt{194})^{11})/3 \rceil \approx \lceil 15.00 \rceil = 15$

Step 8:

Rather than explicitly perform Hensel's lift on our polynomial h to h_k, instead we will use the Pari function factorpadic(), as this is equivalent to factoring f over the p-adic integers out to p^k.

$x^6 + 5x^4 - 2x^3 + 6x^2 - 8x - 8 = ((1 + O(5^{20})) * x^3 + (2 + O(5^{20})) * x + (1 + 4 * 5 + 4 * 5^2 + 4 * 5^3 + 4 * 5^4 + 4 * 5^5 + 4 * 5^6 + 4 * 5^7 + 4 * 5^8 + 4 * 5^9 + 4 * 5^{10} + 4 * 5^{11} + 4 * 5^{12} + 4 * 5^{13} + 4 * 5^{14} + 4 * 5^{15} + 4 * 5^{16} + 4 * 5^{17} + 4 * 5^{18} + 4 * 5^{19} + O(5^{20}))) * ((1 + O(5^{20})) * x^3 + (3 + O(5^{20})) * x + (2 + O(5^{20}))) \in \mathbb{Z}_p[X]$

$\implies$

$x^6 + 5x^4 - 2x^3 + 6x^2 - 8x - 8 \equiv (x^3 + 2x + (1 + 4 * 5 + 4 * 5^2 + 4 * 5^3 + 4 * 5^4 + 4 * 5^5 + 4 * 5^6 + 4 * 5^7 + 4 * 5^8 + 4 * 5^9 + 4 * 5^{10} + 4 * 5^{11} + 4 * 5^{12} + 4 * 5^{13} + 4 * 5^{14} + 4 * 5^{15}) * (x^3 + 3x + 2) \pmod{5^{15}}$

$\implies h_k = x^3 + 2x + 152587890621$

Step 9:

$$L_k = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 5^{15} \\ 0 & 0 & 0 & 0 & 5^{15} & 0 \\ 0 & 0 & 0 & 5^{15} & 0 & 0 \\ 0 & 0 & 1 & 0 & 2 & 152587890621 \\ 0 & 1 & 0 & 2 & 152587890621 & 0 \\ 1 & 0 & 2 & 152587890621 & 0 & 0 \end{pmatrix}$$

and on L_k we run the LLL algorithm to get a reduced short vector basis L'_k=

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 2 & -4 \\ 0 & 1 & 0 & 2 & -4 & 0 \\ 1 & 0 & 2 & -4 & 0 & 0 \\ 2153649694 & 3782460387 & -3176180852 & -1049678002 & 420776097 & -583657165 \\ 5049313149 & -3954390405 & 2298432868 & 2411544719 & 217174760 & 683195595 \\ 2551803419 & 6533340669 & 4443033612 & 2859467662 & 3063068997 & 2642292904 \end{pmatrix}$$

Step 10:

We must take the $\gcd(b_1, b_2, b_3) = \gcd(x^3+2x-4, x^4+2x^2-4x, x^5+2x^3-4x^2) = x^3 + 2x - 4$ as the last three basis elements greatly exceed the bound we must adhere to, and now we have $x^6 + 5x^4 - 2x^3 + 6x^2 - 8x - 8 = (x^3 + 2x - 4)g(x)$ and by performing polynomial division $g(x) = x^3 + 3x + 2$.

$g(x) \equiv x^3 + 3x + 2 \pmod 5 \implies g(x)$ is irreducible in $\mathbb{Z}[X]$ and thus our full factorization is $x^6 + 5x^4 - 2x^3 + 6x^2 - 8x - 8 = (x^3 + 2x - 4)(x^3 + 3x + 2)$ and we're done.

References

[1] Dummit, D. and Foote, R. *Abstract Algebra, 3rd ed.*, John Wiley & Sons Inc., 2004.

[2] Knuth, D. *The Art of Computer Programming Vol. 2, 3rd ed.*, Addison-Wesley, 1998.

[3] Lenstra, A.K. *Lattices and factorization of polynomials*, 1981.

[4] Mignotte, M. *An inequality about factors of polynomials*, Math. Comp. 28, 1974.

[5] A.K. Lenstra, H.W. Lenstra Jr., L. Lovasz, *Factoring Polynomials with Rational Coefficients*, Mathematische Annalen 261, pp. 515-534, 1982.