5-18-2017

# Algorithmic Factorization of Polynomials over Number Fields

Christian Schulz

*Rose-Hulman Institute of Technology*

# Algorithmic Factorization of Polynomials over Number Fields

Christian Schulz

May 18, 2017

**Abstract**

The problem of exact polynomial factorization, in other words expressing a polynomial as a product of irreducible polynomials over some field, has applications in algebraic number theory. Although some algorithms for factorization over algebraic number fields are known, few are taught such general algorithms, as their use is mainly as part of the code of various computer algebra systems. This thesis provides a summary of one such algorithm, which the author has also fully implemented at `https://github.com/Whirligig231/number-field-factorization`, along with an analysis of the runtime of this algorithm. Let $k$ be the product of the degrees of the adjoined elements used to form the algebraic number field in question, let $s$ be the sum of the squares of these degrees, and let $d$ be the degree of the polynomial to be factored; then the runtime of this algorithm is found to be $O(d^4sk^2 + 2^d d^3)$.

## 1 Introduction

### 1.1 Motivation

Polynomial factorization is a problem that comes up in many contexts. From as early as high school algebra, students are introduced to some methods of factoring small polynomials and the benefits of having such a factorization. Though exact factorization is not as useful in physical contexts as approximate (numerical) factorization, it is still useful in some more abstract contexts such as algebraic number theory.

Because these methods are complex and specialized, and because most computer algebra systems support them out of the box, there is little reason for most students to investigate the problem of factoring in the general case, with polynomials of arbitrary degree. However, the problem is an interesting one to study. This thesis will break down one particular version of the factorization problem, wherein our domain consists of the set of polynomials over algebraic number fields.

## 1.2 The Problem

We begin with a few definitions. A *field* is a set of elements with defined addition, subtraction, multiplication, and division operations; the specific axioms satisfied here are not relevant, as all fields involved in this paper will consist of either some set of complex numbers with the familiar operations, or the integers modulo a prime $p$. [10] (Note that in the latter case, all nonzero integers have a multiplicative inverse mod $p$, so division may still be naturally defined.) When one field's elements are a subset of those of another field, and the former field's operations are simply restrictions of those of the latter, we say that the first field is a *subfield* of the second, or alternatively that the second is a *field extension* of the first. [8]

Usually in algebraic number theory, we tend to build larger fields from smaller ones and not the other way around. A common way to do this is what is called *adjunction.* Given a base field $F$, adjunction works by first *adjoining* a formal variable $\alpha$, creating the ring $F[\alpha]$. This ring is the set of all polynomials in $\alpha$ with coefficients from the field $F$, and the polynomials may be added, subtracted, and multiplied as one would expect. In order to make this new structure into a field, we consider polynomials that differ by a multiple of some polynomial $p(\alpha)$ to be equivalent. This is essentially the same construction as how one forms the integers modulo a prime, and as in that case, if the polynomial $p$ chosen is irreducible, every nonzero polynomial in the new ring has a multiplicative inverse, allowing us to define division and making this structure a field. This is written out in full as $F[\alpha]/\langle p(\alpha)\rangle$. This type of extension is known as a *finite simple extension* of the field $F$. The *degree* of the extension is the degree of $p$. (This process, along with the appropriate definitions, is outlined on p. 25 of [9] and p. 292 of [7].)

Note that if $F$ is a subfield of a larger field $G$ such that there exists an element $g \in G$ such that $p(g) = 0$, then we may map elements of $F[\alpha]$ into $G$ by evaluating each polynomial at $\alpha = g$, and this *evaluation homomorphism* respects the equivalence modulo the polynomial $p$. [9] Thus, the field $F[\alpha]/\langle p(\alpha)\rangle$ is isomorphic to some subfield of $G$ via evaluation at $g$. We denote this subfield of $G$ by $F(g)$. Usually we discard the formal details of a quotient ring or polynomial ring at this point and consider the two isomorphic fields to be the same, with $\alpha$ and $g$ also identified.

We may also extend an extension, creating a tower of multiple extensions over a base field. When all of the extensions involved are finite and simple, the overall extension is *finite.* [2] In the cases examined in this paper, every finite extension is isomorphic to a simple extension, but our algorithm does not rely on this.

One of the main objects of study in algebraic number theory is an *algebraic number field,* which is a finite extension of $\mathbb{Q}$. [11] Note that as $\mathbb{Q}$ is a subfield of the algebraically closed field $\mathbb{C}$, every algebraic number field is also a subfield of $\mathbb{C}$; we can let $\mathbb{C}$ play the role of the larger field $G$ above. One example would be the field of Gaussian rational numbers, which is the degree-2 extension $\mathbb{Q}(i) \cong \mathbb{Q}[x]/\langle x^2 + 1\rangle$.

The problem at hand in this paper involves polynomial rings of algebraic number fields. An example of this would be the ring $\mathbb{Q}(i)[x]$ of polynomials in a single variable $x$ whose coefficients are Gaussian rational numbers. Like any polynomial ring over a field, this ring has unique factorization; that is, any such polynomial can be factored uniquely into irreducible components (up to units, i.e. up to any nonzero constant). This unique factorization property for polynomials over an arbitrary field is proven in Theorems 25.3 and 25.4 in [7]. The problem is how to algorithmically factor polynomials over algebraic number fields. Cohen [3] gives an algorithm to solve this problem in several steps in Subsection 3.6.2. The author has spent several weeks implementing Cohen's solution in C++, using the GMP [4] and Eigen [5] libraries, in a GitHub project currently downloadable at `https://github.com/Whirligig231/number-field-factorization`. The purpose of this paper is to give a description of Cohen's process and its sub-procedures as well as a runtime analysis of each.

## 2 Description of the Algorithm

### 2.1 Preliminaries

Before going into the major steps of the algorithm, it is necessary to mention some of the details on substeps. First, some information on representation is helpful. My implementation uses a multi-precision library called GMP to represent arbitrary integers and rational numbers. GMP is designed to efficiently work with integers and rational numbers at multiple levels of precision, allocating and deallocating space as necessary. The GMP class `mpz_class` is used to represent an arbitrary integer, and `mpq_class` similarly represents an arbitrary rational number.

To represent polynomials, the author has used an ordered, array-based list of coefficients. The datatype used to represent polynomials, the class `poly`, is templated over the coefficient datatype, meaning that one can substitute in any type of coefficient (integer, rational, complex, etc.), and the compiler will automatically create a version of the polynomial code for it. Operations such as polynomial addition and multiplication are implemented straightforwardly.

The class `mod` is used to represent numbers in modular arithmetic. Each element of `mod` stores two GMP integers, one for the modulus $n$ and one for the value, which is kept reduced, i.e. in $[0, n)$. Similarly, the templated class `polymod` is used to represent polynomial quotient rings.

To avoid having to nest arbitrarily deep templates, instead of representing e.g. an element of $\mathbb{Q}(\alpha)(\beta)(\gamma)$ with the class `polymod<polymod<polymod<mpq_class>>>`, the author uses a separate class called `numberfield`. A `numberfield` instance represents an element of an arbitrary algebraic number field and contains pointers to a rational value and a `polymod<numberfield>`, as well as a number of "layers," storing how many elements have been adjoined to $\mathbb{Q}$ and thus determining which pointer is valid. If the number of layers is zero, the pointer to the rational value points to the value of the number, which is rational. Otherwise, the pointer to the `polymod` instance points to the value of the number in terms of an adjoined element to a base field.

Several small algorithms are required for the larger parts of the problem. Polynomial GCD is implemented using Euclid's algorithm; however, it should be noted that Euclid's algorithm, unadulterated, may suffer from coefficient explosion. Coefficient explosion occurs when the complexity of the rational numbers involved, i.e. the numerator and denominator, becomes unnecessarily high as the number of iterations of an algorithm increases. If for instance we attempt to perform an algorithm designed for fixed-precision numbers, such as gradient descent, using arbitrary-precision numbers, we may find that the exact numbers involved become unwieldy, such as $\frac{11047017}{2702707071}$ or similar. As such, it is helpful to ensure that the numerators and denominators will not grow out of hand. Cohen guarantees this for most of the algorithms involved in this thesis, but not for Euclid's algorithm.

An algorithm to compute the *resultant* of two polynomials is also necessary. The resultant, defined on [3] p.118, is a constant value equal to the leading coefficient of each polynomial to the power of the degree of the other, times the product of the differences between a root of one polynomial and a root of the other (counted with multiplicity). If $\alpha_1, \ldots, \alpha_m$ are the roots of a polynomial $a$ in an algebraically closed field and $\beta_1, \ldots, \beta_n$ are the roots of a polynomial $b$, then the resultant is equal to:

$$\operatorname{Res}(a, b) = l(a)^m l(b)^n \prod_{i=1}^{m} \prod_{j=1}^{n} (\alpha_i - \beta_j). \tag{1}$$

(In this thesis, the notation $l(a)$ refers to the leading coefficient of $a$.) For example, the resultant of $x^2 - 4x + 3 = (x - 1)(x - 3)$ and $x^2 + x - 6 = (x + 3)(x - 2)$ over $\mathbb{Q}$ is $(1 + 3)(1 - 2)(3 + 3)(3 - 2) = -24$.

Thus, if and only if the two polynomials have a common factor, their resultant will be zero. An algorithm called the sub-resultant algorithm, which is essentially a modified form of Euclid's algorithm, is used to compute resultants over an arbitrary UFD in terms of the coefficients and may be modified to compute polynomial GCDs without risk of coefficient explosion. This algorithm and its modification are covered in Section 3.3 of [3].

One algorithm requires the use of vectors and matrices; for this, I use the library Eigen, whose templated classes allow a large number of operations on vectors and matrices.

## 2.2 Factoring mod $p$: Berlekamp's Algorithm

Berlekamp's algorithm is an algorithm used, in general, to factor polynomials over finite fields, of which $\mathbb{Z}_p$, the field of integers mod $p$, is an important special case. Let $a$ be a squarefree polynomial with coefficients in $\mathbb{Z}_p$. To find the factors of $a$, we will find polynomials $t$ such that given any irreducible factor of $a$, $t$ differs from a multiple of that factor by a constant. This seems like a very strong condition, but Proposition 3.4.9 in Cohen [3] states that these are precisely the polynomials $t$ such that $t^p \equiv t \pmod{a}$. By Fermat's little theorem, in $\mathbb{Z}_p$, $k^p = k$ for any constant $k$, so this set is closed under constant multiplication. Also, note that due to the "Freshman's Dream" theorem, given $t_1$ and $t_2$ in the polynomial ring $\mathbb{Z}_p[x]$, we have $(t_1 + t_2)^p = t_1^p + t_2^p$, so the aforementioned set is also closed under polynomial addition. Thus, viewing the quotient ring $\mathbb{Z}_p[x]/\langle a \rangle$ as a vector space over $\mathbb{Z}_p$, the set of elements $t$ such that $t^p = t$ is a subspace of this vector space, called the *Berlekamp subalgebra.* Thus, to characterize this set computationally, we need only find a basis for its vectors. To do this, we create a square matrix $Q$ whose $k$th column contains the coefficients of the polynomial $x^{kp}$ when it is reduced mod $a$.

Then the vector $Qt$ for some polynomial $t$ (treated as a vector of coefficients) is the polynomial $t^p$ reduced mod $a$. If $(Q-I)t$, $I$ being the identity matrix, is the zero vector, this is equivalent to the condition that $t^p \equiv t \mod a$, so the kernel of the matrix $Q-I$ is precisely the Berlekamp subalgebra. Lastly, Cohen also proves that the dimension of this subalgebra is precisely the number of irreducible factors of $a$.

For example, consider the polynomial $a(x) = x^3 - 2$ in $\mathbb{Z}_5$. Then $x^5 \equiv (x^3)x^2 \equiv 2x^2$, and $x^{10} \equiv (x^3)(x^3)(x^3)x \equiv 3x$. Then our matrices are:

$$Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 3 \\ 0 & 2 & 0 \end{bmatrix}, Q - I = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 3 \\ 0 & 2 & -1 \end{bmatrix}.$$

The kernel of $Q-I$ has the basis vectors $[1\ 0\ 0]^T$ and $[0\ 3\ 1]^T$, so the two basis polynomials for the Berlekamp subalgebra are 1 and $x^2 + 3x$.

The question, of course, is whether we may find *all* of the factors of the polynomial $a$ this way. In particular, it is necessary to know that we will not encounter an edge case where every $t$ that differs by a constant from a multiple of one given factor of $a$ also differs by the same constant from some other given factor of $a$, thus preventing us from separating the two.

To see why this is not an issue, note that if a polynomial $t$ in the Berlekamp subalgebra differed by a single constant from a multiple of *every* irreducible factor of $a$, since $a$ is squarefree, it would also differ by a constant from a multiple of $a$. But this is impossible, as the degree of $t$ is less than that of $a$. So at the very least we will achieve a nontrivial splitting of $a$. We could then continue by splitting the two factors recursively. As it turns out, we do not need to recompute the Berlekamp subalgebra in the process; we may simply split a factor $a_1$ of $a$ by using vectors from the Berlekamp subalgebra of $a$ and discarding (via taking the GCD) any factors of $a$ that do not come from within $a_1$.

We may thus factor a polynomial over $\mathbb{Z}_p$ as follows. First, we compute the matrix $Q$, using modular exponentiation by squaring and reducing. Then we compute the kernel of the matrix $Q-I$ to find a basis for the Berlekamp subalgebra, using a simple alteration of Gaussian elimination. There are two ways we could now proceed. We could take every basis element $t$, try the GCD of $t-s$ with $a$ for each constant $s$, and find all of the factors of $a$ this way. However, this method has runtime proportional to $p$, as we need to try every constant $s$. Instead, we can simply pick a linear combination of the basis vectors uniformly at random, in which case Cohen claims ([3] p. 131) that a nontrivial factor will be found in at least 44% of iterations (assuming of course that one exists). The probabilistic algorithm has expected runtime proportional to the logarithm of $p$ instead of $p$ itself, so it is much better for large $p$.

Indeed, using the example given above we may find that by adding the constant 2 to $x^2 + 3x$, the GCD of the new polynomial with $x^3 - 2$ is $x + 2$. If we add the constant 4, we find by taking the GCD that $x^2 + 3x + 4$ is already a factor of $x^3 - 2$. These two factors make up the irreducible factorization of $x^3 - 2 = (x+2)(x^2+3x+4) \bmod 5$.

## 2.3 Factoring mod $p^e$: Hensel Lifting

The main idea of this portion of the algorithm is to take a factorization mod $p$ and lift it to a factorization mod $p^e$. Here *lifting* means adding multiples of $p$ such that the new factors are the same in $\mathbb{Z}_p$ but different in $\mathbb{Z}_{p^e}$. The method comes from Theorem 3.5.4, given in Cohen, and the following Algorithms 3.5.5 and 3.5.6, the two Hensel lifts.

The general idea behind the two Hensel lifts is as follows. We begin with two positive integers $p$ and $q$. These are not necessarily prime; the choice of letters is due to Cohen. In our application, $p$ and $q$ will at least always be prime powers, even though this is not required for Algorithms 3.5.5 or 3.5.6. Then assume that we have polynomials $A, B, C, U, V$ such that $C \equiv AB \pmod{p}$ and $UA + VB \equiv 1 \pmod{q}$. What the Hensel lift allows us to do is replace $A$ and $B$ with liftings thereof such that the congruences continue to hold for higher multiples of $p$ and $q$. In particular, let $r = (p, q)$ be the greatest common divisor of $p$ and $q$. Then Algorithm 3.5.5 performs some simple algebraic manipulations to give us polynomials $A'$ and $B'$, congruent to $A$ and $B$ mod $p$, such that $C \equiv A'B' \pmod{pr}$.

Note that replacing $p$ with $pr$, the conditions of Algorithm 3.5.5 are still satisfied, so we could run the algorithm again and continue multiplying $p$ by $r$ and achieving better and better liftings of $A$ and $B$. However, we can improve this process using Algorithm 3.5.6.

Algorithm 3.5.6 performs algebraic manipulations again, similar to Algorithm 3.5.5, but this time producing new polynomials $U'$ and $V'$ congruent to $U$ and $V$ mod $q$ such that $U'A' + V'B' \equiv 1 \pmod{qr}$. In doing so, note that we have replaced $q$ with $qr$ and $r$ with $r^2$. Thus, the factor by which we are lifting, if we use both algorithms in an alternating fashion, squares every time. This leads to a fairly simple procedure for lifting a factorization mod $p$ to a factorization mod $p^e$.

We will first handle the case of a squarefree polynomial with two factors, $C = AB \pmod{p}$. First, as mentioned, we must find $U$ and $V$ such that $UA + VB = 1 \pmod{p}$. This can be done with Euclid's algorithm, as $A$ and $B$ have no common factors mod $p$. Then by running Algorithm 3.5.5, we produce liftings $A'$ and $B'$ such that the factorization $C = A'B'$ holds mod $p \cdot (p, p) = p^2$. We can run Algorithm 3.5.6 to find liftings $U'$ and $V'$ such that the Euclidean equation $U'A' + V'B' = 1$ holds mod $p \cdot (p, p) = p^2$. We have now effectively replaced $p$ with $p^2$ and $A, B, U, V$ by their respective liftings $A', B', U', V'$ in the conditions for the Hensel lift.

So we can run this same Hensel lift multiple times, every time squaring the prime power over which the equation holds; after $n$ iterations of the above process, the equation $C = A'B'$ will hold mod $p^{2^n}$. So we merely select an $n$ such that $2^n \geq e$.

The question remains how we extend this algorithm to handle more than two factors. Note that it does not matter in Hensel lifting whether the polynomials $A$ and $B$ are themselves irreducible, only that they have no common factors (which is guaranteed if $C$ is squarefree). So let $C = A_1 A_2 \ldots A_n$ be a factorization of $C$ mod $p$. For $1 \leq i < n$, let $F_i = A_{i+1} A_{i+2} \ldots A_n$. Apply Hensel lifting to the equation $C = A_1 F_1 \pmod{p}$ to produce $F_1'$ and $A_1'$ such that $C = A_1' F_1' \pmod{p^e}$. Now $F_1' = A_2 F_2 \pmod{p}$, so we run the same Hensel lift on this equation. We continue, iterating, until we find the value of $F_{n-1}'$, which is congruent to $A_n$. Then $C = A_1' F_1' = A_1'(A_2' F_2') = \cdots = A_1' A_2' \ldots A_n' \pmod{p^e}$, which is exactly what we need.

For an example of this, let us consider the polynomial $x^3+2x^2+5x+3$. Modulo 7, this is congruent to $(x+2)(x+3)(x+4)$. Say we want to find a factorization of this polynomial mod $49 = 7^2$. First, we apply Hensel lifting to $x^3 + 2x^2 + 5x + 3 \equiv (x + 2)(x^2 + 5) \pmod{7}$. Only a single lift is required in this case to raise the modulus from 7 to 49; we get $x^3 + 2x^2 + 5x + 3 \equiv (x + 23)(x^2 + 28x + 47) \pmod{49}$. Next, we know that $x^2 + 28x + 47 \equiv x^2 + 5 \equiv (x + 3)(x + 4) \pmod{7}$, so we perform a second Hensel lift to lift this factorization to $(x+24)(x+4) \pmod{49}$. Thus, the complete lifted factorization is $x^3 + 2x^2 + 5x + 3 \equiv (x + 23)(x + 24)(x + 4) \pmod{49}$.

## 2.4 Bounding the Coefficients of a Polynomial over $\mathbb{Z}$

The general method we give for factoring over $\mathbb{Z}$ is to first factor modulo some prime, then to raise that factorization with Hensel lifting to a factorization mod $p^e$ for sufficiently large $e$. The purpose of this subsection is to clarify precisely what is meant by "sufficiently large." In short, we will want to find a suitable exponent $e$ such that the coefficients of the factors of $u$ are guaranteed to fall in $[-\frac{p^e-1}{2}, \frac{p^e-1}{2}]$. To do this, we turn to Cohen's Theorem 3.5.1, which states that if a polynomial $b$ divides a polynomial $a$ and has degree at most half the degree of $a$, then:

$$|b_j| \leq \binom{d-1}{j}|a| + \binom{d-1}{j-1}|l(a)|$$

where $b_j$ is the $j$th coefficient of $b$, $l(a)$ is the leading coefficient of $a$, $d$ is the degree of $b$, and $|a|$ is the $l_2$-norm of $a$, that is, the square root of the sum of the squares of its coefficients.

Note that we only care about the values of $d$ and $j$ maximizing this expression. We clearly want $d$ to be as high as possible, so we let $d$ be half the degree of $a$, rounded down.

Now we want the value of $j$ giving the highest value. If $d$ is even, this value is $\frac{d}{2}$, which maximizes the values of the two binomial coefficients in the formula. If $d$ is odd, this value is either $\frac{d+1}{2}$ or $\frac{d-1}{2}$. In the former case, the second coefficient will be larger, while in the latter case, the first coefficient will be larger. The values of the two coefficients switch in these cases; call them $c_1$ and $c_2$, with $c_1 \leq c_2$.

Now we know that $|a| \geq |l(a)|$, and all of these quantities are going to be nonnegative integers. Given that $|l(a)| \leq |a|$ and $c_1 \leq c_2$, we can state that $|l(a)|c_2 + |a|c_1 \leq |l(a)|c_1 + |a|c_2$. So to maximize $|b_j|$, we want to multiply $|a|$ by $c_2$, the larger coefficient; thus the first coefficient should be larger, and we use $j = \frac{d-1}{2}$ when $d$ is odd.

In any case, letting $j = \lfloor \frac{d}{2} \rfloor$, we get an upper bound of:

$$|b_j| \leq B = \binom{\lfloor \frac{\deg(a)}{2} \rfloor - 1}{\lfloor \frac{\deg(a)}{4} \rfloor}|a| + \binom{\lfloor \frac{\deg(a)}{2} \rfloor - 1}{\lfloor \frac{\deg(a)}{4} - 1 \rfloor}|l(a)|$$

which then applies to all coefficients $b_j$. We then find the smallest $e$ such that $p^e \geq B$ with a binary search. In order to speed up the exponentiation required for the search, note that we can first try, and store, the powers $p^{2^k}$ and then multiply these to form other powers. In doing so, we combine the binary search algorithm with exponentiation by squaring.

With this bound on the coefficients of factors of $a$, we now have the pieces to put together into an algorithm for factorization over $\mathbb{Z}$.

## 2.5 Factoring over $\mathbb{Z}$

The idea here is as follows. As explained in the previous subsections, it is comparatively easy to factor modulo prime powers. Any factorization over $\mathbb{Z}$ is valid over $\mathbb{Z}_{p^e}$, but the converse might not be true, for two reasons.

The first reason is that a factor over $\mathbb{Z}_{p^e}$ has infinitely many liftings, and at most one of them will be valid over $\mathbb{Z}$. For instance, consider the polynomial $x^3+2x^2-2x-1$. Over $\mathbb{Z}_3$, this factors as $(x-1)(x^2+1)$. But we do not want to lift $x^2+1$ to itself over $\mathbb{Z}$; rather, the factorization over $\mathbb{Z}$ is $(x-1)(x^2+3x+1)$. This shows that we cannot simply choose the lifting whose coefficients have least absolute value, and that the absolute values of coefficients of the factors are not bounded by those of the coefficients of the original polynomial.

The second reason is that irreducible polynomials over $\mathbb{Z}$ may not be irreducible over $\mathbb{Z}_{p^e}$. For instance, over $\mathbb{Z}_5$, the polynomial above splits into linear factors as $(x-1)^3$. But we cannot factor $x^2+3x+1$ over $\mathbb{Z}$. Choosing which prime power to use will not always help; for instance, the polynomial $x^4+1$ is irreducible over $\mathbb{Z}$ but splits over $\mathbb{Z}_{p^e}$ for *any* prime $p$ and positive integer $e$.

So our method will factor over $p^e$ for carefully chosen $p$ and $e$ such that the first problem disappears. Then, we will use brute force to get around the second problem, which will unfortunately (see Section 3) lead to an exponential runtime in the worst case.

Let $a$ be our input polynomial. First, we divide out by the *content* of the polynomial, which is the greatest common divisor of its coefficients. The resulting polynomial will then be *primitive,* meaning that its only constant factors are units. (Of course, we keep track of the content so that we can include it as a constant factor later on.)

To simplify things, we will guarantee that the polynomial we factor is squarefree over both $\mathbb{Z}$ and $\mathbb{Z}_p$. Note that we can compute $(a, a')$ to determine whether the polynomial is squarefree, and we can divide by this polynomial to produce a polynomial with the same factors but multiplicity 1. [12] We do so, letting $u = \frac{a}{(a,a')}$.

It is not difficult to perform some optimizations at this point. First off, it is trivial to check if $a$ has any factors of $x$, so we cast those out at this point. Secondly, as we will see later, some of our computations depend on $l(u)$, the leading coefficient of $u$. If this value is larger than the constant term of $u$, it will be more efficient to reverse $u$, switching the order of the coefficients (so e.g. $12x^2+7x+1$ would become $x^2+7x+12$). This reversal operation distributes over polynomial multiplication, so we may reverse our factors at the end to account for this and maintain our factorization.

Next, we search for a prime $p$ such that $u$ is still squarefree mod $p$. This process may take some time (see Section 3), but we can place some bounds on it that guarantee it is not the limiting factor if we use a sufficiently fast primality test such as Miller-Rabin. My implementation uses GMP's function `mpz_nextprime`, which in turn uses 25 iterations of Miller-Rabin. This is not guaranteed to have no false positives, but the chance of a composite number slipping through is bounded above by $4^{-25}$. Putting this in practical terms, if one began running one primality test per millisecond now, one would expect the first false positive to appear in the year A.D. 37695.

We can now perform Berlekamp factorization to factor our reduced polynomial $u$ over $\mathbb{Z}_p$, and use the Hensel lifts explained in Subsection 2.3 to lift this to a factorization over $\mathbb{Z}_{p^e}$, for $e$ chosen according to Subsection 2.4. After this, we will begin to test the factors we have found.

As mentioned before, we are not guaranteed that our new factors will correspond one-to-one with the integer factors of $u$. In fact, in the worst case, as will be proven in Section 3, a polynomial of degree $d$ will split linearly over $\mathbb{Z}_{p^e}$, while over $\mathbb{Z}$ it factors into two irreducible polynomials of degree $\frac{d}{2}$. So we must instead try combinations of factors to see if their product corresponds to a factor over $\mathbb{Z}$.

For efficiency's sake, we will only check the first half of the combinations. When the number of factors mod $p^e$ is odd, this means we check subsets of less than half of the factors. When the number of factors is even, we have an additional step wherein we check those subsets of size equal to half the number of factors that include the first factor. (These are the complements of the other subsets of this size.)

Once we have picked a subset, we multiply those factors to produce a polynomial $v$ and check the degree of $v$. Note that the bound we gave in Subsection 2.4 is only valid for factors of degree less than or equal to half the degree of $u$, so if the degree of $v$ is higher, we instead divide $u$ by $v$ in $\mathbb{Z}_{p^e}$ and set $v$ to this polynomial instead. Next, we shift all of the coefficients to be in the range $[-\frac{p^e-1}{2}, \frac{p^e-1}{2}]$.

Note that if $u$ is not monic, we may have introduced a constant factor to $v$ such that it will not divide $u$. This is because we produced $v$ by factoring $u$ over the field $\mathbb{Z}_p$, in which any nonzero constant is a unit. We could compute the content of $v$ to account for this, but it is faster to multiply $u$ by $l(u)$ instead. (Note that if $u$ is monic, the algorithm will keep $v$ monic, and there is no issue. Therefore any extra factor introduced to $v$ is a factor of $l(u)$.) First, as Cohen recommends ([3], p. 139), we check the divisibility of the constant terms; Cohen claims this usually clears a lot of false positives. If the polynomial $v$ passes this test, we check if it divides $l(u)u$.

Once a polynomial $v$ passes both divisibility tests, we first divide $v$ by its content. Then we can determine the multiplicity of $v$ in the original polynomial $a$ through repeated division. Lastly, we remove the factors mod $p^e$ that we used to find $v$ from the list. (Note that if we divided $u$ by $v$, we have actually multiplied the remaining factors, so we remove those instead.) We continue in this manner until the polynomial is fully factored, reverse the factors if $u$ was reversed before, include any factors of $x$ computed before, include the content of $a$ as a constant factor, and return the final list of factors.

## 2.6 Factoring over $\mathbb{Q}$

This step is trivial once we take into account Gauss's lemma, which states that a polynomial with integer coefficients that is irreducible over $\mathbb{Z}$ is irreducible over $\mathbb{Q}$. This means that in order to factor over $\mathbb{Q}$, we need only multiply by some constant in order to clear the denominators, factor the resulting polynomial over $\mathbb{Z}$, and divide out by the constant we used.

## 2.7 Factoring over $K(\alpha)$

The idea here is to make use of what are called *conjugate elements.* When we create a polynomial extension, we adjoin a root $a$ of some polynomial $p$. Assuming that our base field is a number field and that $p$ is irreducible, there is no algebraic information distinguishing $a$ from the other roots of $p$. Any algebraic equation will continue to hold if $a$ is replaced by the other roots. The expressions produced by replacing $a$ in this manner are called *conjugates* of the original expression, and the number of conjugates, counting the original expression and counting multiplicity, is the degree of $p$. For example, in $\mathbb{Q}(i)$, the conjugates of $2+3i$ are itself and its complex conjugate $2-3i$. Similarly, the (nontrivial) conjugate of $x^2+ix-3+4i$ is $x^2-ix-3-4i$.

The *norm* of an element or polynomial is the product of its conjugates. For example, the norm of $2+3i$ is $(2+3i)(2-3i)=13$. The norm is always an element of, or a polynomial over, the base field. Note that because algebraic equations remain the same under conjugation, polynomial factorization is preserved; i.e. the norm operation is a multiplicative homomorphism.

With this, we have an idea for how to factor in $K(\alpha)$. First, we take our polynomial and multiply it by each of its conjugates to produce the norm. Now, we factor the norm over $K$. It is not difficult to prove from our previous assertions that (Lemma 3.6.3 in [3]) the irreducible factors in the norm are simply the norms of the irreducible factors of the original polynomial. Thus, we can take polynomial GCDs with the original polynomial to find these factors.

There is one issue that may occur in this process: two factors of the original polynomial may have the same norm, in which case the GCD will not give us one or the other but rather both. To prevent this, we simply perform a transformation in advance, shifting the input: instead of factoring the polynomial $a$, we factor $a_k$ where $a_k(x)=a(x+k\alpha)$ for some integer $k$. Cohen's Lemma 3.6.2 guarantees that we will find a $k$ such that no two factors have the same norm. To test whether a given $k$ works, we simply check whether the norm of $a_k$ is squarefree. If not, we increment $k$ and try again.

The one missing piece to this puzzle is how to actually compute the norm of a polynomial. To do this, we can use resultants. Recall that the resultant is the product of powers of the leading coefficients with the differences between the roots of two polynomials (Equation 1 in Subsection 2.1). Equivalently, the resultant of $b$ and $a$ is a power of the leading coefficient of $b$, times the values of the polynomial $a$ evaluated at every root of $b$. We can view our polynomial $a$ over $K(\alpha)$ as a multivariate polynomial in $x$ and $\alpha$ over $K$. Then the resultant with respect to $\alpha$ of the minimal polynomial of $\alpha$ over $K$ with the polynomial $a$ is some power of 1 times several copies of $a$ with $\alpha$ replaced by each root in turn of its minimal polynomial—which are precisely its conjugate elements, so this polynomial is the norm! This can be written in mathematical notation as:

$$\mathrm{Res}_\alpha(a(x,\alpha),p(\alpha)) = 1^{\deg(a)} \prod_{\alpha':p(\alpha')=0} a(x,\alpha') = \prod_{\alpha':p(\alpha')=0} a(x,\alpha') = (\text{norm of } a(x)).$$

For example, consider the polynomial $x^2 + \sqrt{2}$ over $\mathbb{Q}(\sqrt{2})$. We can use Algorithm 3.3.7 in Cohen to compute $\operatorname{Res}_\alpha(x^2 + \alpha, \alpha^2 - 2) = x^4 - 2$. This is equal to the norm $(x^2 + \sqrt{2})(x^2 - \sqrt{2}) = x^4 - 2$.

From here the process is fairly straightforward to line out. First, we search for a shift $k$ under which the norm is squarefree. For each $k$, we perform a squarefree test by taking the GCD of the norm of $a(x + k\alpha)$ with its formal derivative. Once we find a $k$ that satisfies this condition, we factor the norm over $K$. Lastly, for every factor of the norm, we undo the coordinate shift we performed earlier and take the GCD with the original polynomial $a$ to find a factor of $a$. Apart from the standard handling of the case where $a$ is not squarefree to begin with, this algorithm will fully factor any polynomial over $K(\alpha)$. Applied recursively, this gives us a means for factoring any polynomial over any number field.

## 3 Runtime Analysis

Now that we have explained in detail the method for factoring polynomials over number fields, the other part of this thesis will be devoted to determining the runtime of this algorithm and its various sub-algorithms.

In order to do this, we must make some assumptions about atomic operations. In this thesis, we will assume that integers (and hence rational numbers) take a constant amount of storage and that their elementary operations take constant time. This assumption is not entirely accurate; in reality, both are logarithmic in the magnitude of the integer (or numerator and denominator). But this is mostly independent of the behavior in terms of the degree of the polynomial and the extensions we use, so in the end we may assume that it is largely irrelevant to the main source of complexity.

In some cases, a different algorithm is required in order to avoid coefficient explosion. In particular, Euclid's algorithm (see Subsection 2.1) should not be used for rational polynomials for this reason. Instead, we may use the sub-resultant algorithm (same subsection) to compute polynomial GCDs instead. This is however the only major case where coefficient explosion becomes a potential issue. When we are factoring mod $p$ or $p^e$, coefficient explosion is not an issue because the number of possible coefficients is bounded. When we are factoring over $\mathbb{Z}$, $\mathbb{Q}$, or $K(\alpha)$, the coefficients involved in intermediate steps are not significantly more complex than those of the original polynomial and its factors.

### 3.1 Polynomial Arithmetic

First, we must determine the runtimes of basic arithmetic operations for polynomials and elements of number fields. These are recursively defined in terms of one another, of course, so the analysis becomes somewhat complex. In these analyses, $d_1$ and $d_2$ are the degrees of the two polynomials, and $d$ is the larger of the two.

To add two polynomials, we simply perform $d$ additions in the base field. To negate a polynomial, we perform $d$ negations in the base field. Subtraction is simply negation plus addition. To multiply or divide a polynomial by a constant, we perform $d$ multiplications or divisions in the base field, respectively. To multiply two polynomials with the most obvious implementation requires $d^2$ multiplications and $d^2$ additions in the base field; the author did not feel that additional work to speed this up was merited.

Polynomial division is somewhat harder. First, we perform one division in the base field. Then we begin a loop, dividing out one coefficient at a time until the degree of the remainder is less than that of the divisor. Thus, the number of times this polynomial runs is, up to a constant, at most $|d_1 - d_2|$, as we will always remove at least one term from the remainder. Inside the loop, we perform two multiplications and $2d_1$ additions in the base field. There are two other variations on this division algorithm, but their runtimes turn out not to be significantly different. For most cases we will simply say that the runtime is $O(d^2T)$, where $d = \max(d_1, d_2)$ and $T$ is the runtime of a single multiplication or division in the base field. In cases where the difference between $d_1$ and $d_2$ is better than linear in terms of $d$, we will note this.

We now examine the runtimes of these operations in the quotient ring we use to represent a simple field extension. Addition and negation are not substantially different; we simply add or negate the respective polynomials. Multiplication begins with multiplication of the two polynomials, but now we must also divide their product by the polynomial modulus $a$ in order to reduce it modulo $a$.

Division in a field extension is implemented as a multiplication by the multiplicative inverse of the divisor. To compute the modular multiplicative inverse, we must first perform the extended Euclidean algorithm on the polynomial with the polynomial modulus. Then we do a division by a constant and lastly a polynomial division to compute the residue.

The Euclidean algorithm for polynomials, as mentioned in Subsection 2.1, is the final piece of this puzzle. The main portion of it is a loop that runs until the remainder of one polynomial divided by the other is zero. Inside this loop, we perform a polynomial division, a polynomial subtraction, and a polynomial multiplication. Careful investigation reveals that this is not as slow as it may appear. Note that the runtime of the division depends on the difference in the degrees between the two input polynomials; the sum of all of the differences involved, every time we reach this statement, is in fact linear in $d$ as opposed to quadratic. Moreover, one of the two polynomials in the multiplication is the quotient of the division, which will have degree 1 in every iteration other than (possibly) the first. So overall, the Euclidean algorithm requires $O(d^2)$ additions, subtractions, and multiplications in the base field, as well as $O(d)$ divisions. Figure 1 gives a graph of all of these recurrences.

Now we may begin to unfold these recursions. For addition, subtraction, and negation, a polynomial operation takes $d$ operations in the base field, and an operation in a field extension takes one polynomial operation over the base field whose degree is the degree of the extension (minus one). Thus, we conclude that general field addition, subtraction, and negation are $O(k)$, where $k$ is the product of the degrees of the extensions used, and general polynomial addition, subtraction, and negation are $O(dk)$, where $d$ is the degree of the (larger) polynomial and $k$ is the product of the degrees of the extensions used to form the base field.

We will write the runtimes of the other operations as a system of big-O recurrence relations in terms of $i$, the number of extensions we are building in a tower on top of $\mathbb{Q}$. Here:

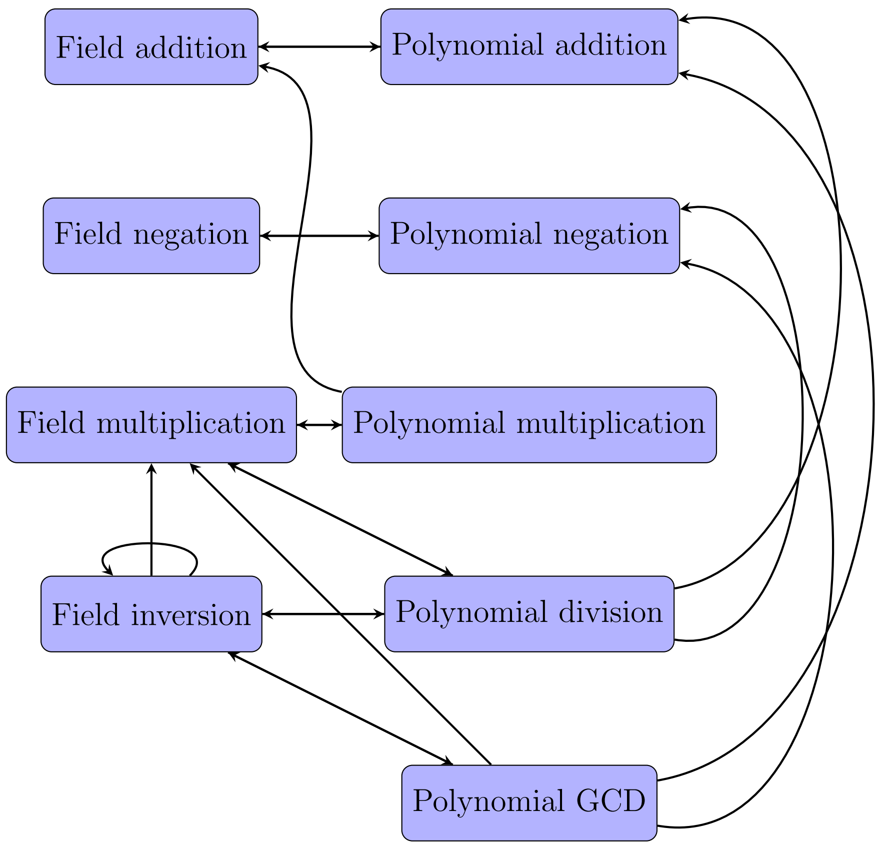

Figure 1: A graph of the recurrences for polynomial and field operations. An arrow from A to B indicates that algorithm A uses algorithm B as a subroutine.

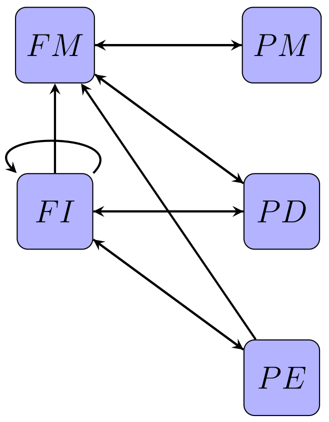

Figure 2: A graph of the recurrences for multiplication, division, and Euclid's algorithm, as formally notated.

- $d_i$ is the degree of the polynomial modulus used to define the $i$th extension above $\mathbb{Q}$ in the tower;
- $d$ is the degree of the (larger) outermost polynomial in a polynomial operation;
- $PM_i$ is the runtime of polynomial multiplication over $i$ extensions;
- $FM_i$ is the runtime of field multiplication in the $i$th extension;
- $PD_i$ is the runtime of polynomial division over $i$ extensions;
- $FI_i$ is the runtime of field inversion in the $i$th extension;
- $PE_i$ is the runtime of Euclid's algorithm with polynomials over $i$ extensions.

$$PM_i \in O\left(d^2 \prod_{j \le i} d_j + d^2 FM_i\right)$$

$$FM_i \in O\left(PM_{i-1} + PD_{i-1}\right)$$

$$PD_i \in O\left(FM_i + FI_i + dFM_i + d^2 \prod_{j \le i} d_j\right)$$

$$FI_i \in O\left(PE_{i-1} + d_i FM_{i-1} + d_i FI_{i-1} + PD_{i-1}\right)$$

$$PE_i \in O\left(d^2 \prod_{j \le i} d_j + d^2 FM_i + dFI_i\right)$$

$$FM_0, FI_0 \in O(1).$$

Figure 2 gives a diagram of these relations, as in the previous diagram but with the new notation.

First we will reduce the number of equations by substituting the polynomial runtimes into the corresponding field runtimes:

$$FM_i \in O\left(d_i \prod_{j\leq i} d_j + d_i^2 FM_{i-1} + FM_{i-1} + FI_{i-1} + d_i FM_{i-1} + d_i \prod_{j\leq i} d_j\right)$$

$$FI_i \in O\left(d_i \prod_{j\leq i} d_j + d_i^2 FM_{i-1} + d_i FI_{i-1} + d_i FM_{i-1}\right.$$

$$\left. + d_i FI_{i-1} + FM_{i-1} + FI_{i-1} + d_i FM_{i-1} + d_i \prod_{j\leq i} d_j\right)$$

$$FM_0, FI_0 \in O(1).$$

We can now combine a large number of like and dominated terms:

$$FM_i \in O\left(d_i \prod_{j\leq i} d_j + d_i^2 FM_{i-1} + FI_{i-1}\right)$$

$$FI_i \in O\left(d_i \prod_{j\leq i} d_j + d_i^2 FM_{i-1} + d_i FI_{i-1}\right)$$

$$FM_0, FI_0 \in O(1).$$

Noting the similarity between these, we rewrite the second equation in terms of the first:

$$FM_i \in O\left(d_i \prod_{j\leq i} d_j + d_i^2 FM_{i-1} + FI_{i-1}\right)$$

$$FI_i \in O(FM_i + d_i FI_{i-1})$$

$$FM_0, FI_0 \in O(1).$$

Note that due to the $d_i^2 FM_{i-1}$ term, the complexity of $FM_i$ will be at least $\prod_{j\leq i} d_j^2$. So we can remove the first term of $FM_i$, as it is dominated by the second term:

$$FM_i \in O(d_i^2 FM_{i-1} + FI_{i-1})$$

$$FI_i \in O(FM_i + d_i FI_{i-1})$$

$$FM_0, FI_0 \in O(1).$$

Note that the $d_i FI_{i-1}$ term adds a linear factor in terms of the previous level, while $FM_i$ is quadratic in terms of the previous level. Moreover, note that if we suppose that $FI_i$ is also quadratic in terms of the previous level, the $d_i FI_{i-1}$ term is dominated and can be removed. Then $FI_i$ and $FM_i$ have the same big-O complexity, and we have found a solution to the system of recurrences.

We conclude that field multiplication and inversion are both $O(k^2)$. From this, it follows that polynomial multiplication, division, and the Euclidean algorithm are $O(d^2k^2)$. Due to the fact that the power on $k$ and $d$ is 2, we will refer to these algorithms collectively as the *quadratic operations.*

## 3.2 Other Preliminary Algorithms

Note that one of the steps in our algorithm to factor over $\mathbb{Z}$ requires us to compute binomial coefficients $\binom{n}{r}$. A naive recursive implementation of this would be exponential due to the overlap in computation of various values, but if we memoize the algorithm with a global array, we need only compute each value once. To compute $\binom{n}{r}$, we will need the values of $\binom{n'}{r'}$ for all pairs such that $r' \leq r$ and $n' - r' \leq n - r$; the number of pairs is $r(n-r)$, so this algorithm is $O(r(n-r))$ with memoization.

The sub-resultant algorithm and its modification to compute GCDs (see Subsection 2.1) begin and end with various constant operations, which have a runtime of $O(dk^2)$. The centerpiece of each algorithm is a loop that divides one polynomial by another, recursively replacing the first with the second and the second with the remainder, until the remainder is a constant. This is the same loop as in the Euclidean algorithm, so we conclude that it runs fewer than $d$ times, $d$ being the higher of the two degrees. We also conclude based on our similar analysis of the Euclidean algorithm that the division loop inside this loop runs a total of $O(d)$ times and not $O(d^2)$ as a naive analysis might conclude. Inside the loop, all other operations besides the division are $O(dk^2)$. Thus, the total runtime is $O(d^2k^2)$.

The matrix kernel algorithm is a simple alteration of Gaussian elimination. The slowest part of the matrix kernel algorithm is a triply nested loop. The outermost loop runs once per column, the middle loop runs once per row, and the innermost loop runs once over all the columns past the current column in the outermost loop. This would be cubic behavior, but the two inner loops are also in one of the two sides of a branch. The first side, the side the loops are *not* in, is run once for each basis vector in the kernel, so the inner loops are run a number of times equal to the matrix's rank. There is another quadratic loop outside of the cubic loop, so overall, assuming an $n \times n$ square matrix of rank $r$, the runtime of this algorithm is $O(n^2(r+1))$.

## 3.3 Factoring mod $p$

Consider the case of using Berlekamp's algorithm to factor a polynomial $a$ of degree $d$ over $\mathbb{Z}_p$, and assume that this polynomial will eventually have $f$ irreducible factors. We will determine the runtime of this algorithm in terms of $d$, $p$, and $f$.

First, as stated before, we must compute the matrix $Q$ using modular exponentiation by squaring. We do this by first computing the polynomial $X^p \pmod a$ and then computing its powers in succession. The first step requires us to, $\lceil \log_2 p \rceil$ times, perform a multiplication (to square the current polynomial) and a division (to reduce it mod $a$), and then perform a second loop $\lceil \log_2 p \rceil$ times to add the powers indexed by powers of 2. (For instance, to compute $X^{13}$, we would first compute $X^2, X^4, X^8, X^{16}$ by squaring, and then we would use the binary expansion of 13 to compute $X^{13} = X \cdot X^4 \cdot X^8$.) As this loop has $O(\log p)$ steps, and each step is a constant number of quadratic operations, the runtime of this step is $O(d^2 \log p)$. After this, computing the successive powers is a polynomial multiplication and modular reduction (both of which are $O(d^2)$), $d$ times, for a time of $O(d^3)$.

Once these powers are put into the matrix, we compute a basis for its kernel. As determined earlier, the runtime of this algorithm is $O(n^2(r+1))$. Because the rank of the matrix is its dimension $d$ minus the nullity, which as mentioned in Subsection 2.2 is $f$, this runtime can be rewritten as $O(d^2(d-f+1))$, which is bounded by $O(d^3)$ and can thus be ignored.

The next part of the algorithm is a large loop that runs until all of the irreducible factors have been found. Inside this loop, we take one of the basis vectors $t$ from the kernel and apply it to each of the currently found factors using a shift of every possible constant $s$ mod $p$ by computing the GCD $(t-s, a)$. In the worst case, the algorithm would find all but one of the factors using the first basis vector and then fail to split the last factor until it tried the last basis vector with $p-1$ as the shift. (Later we will see that we do not need to worry about whether this bound is achieved.) Inside the innermost part of this loop, we perform a GCD operation, which is a quadratic operation.

In summary, we need to run a loop at most $f$ times, once to find each factor, in which we loop over each factor already found, leading to quadratic behavior in terms of $f$. Then we try each of the $p$ possible constants for $s$ and use it to compute a quadratic operation in terms of $d$. So in this case, the runtime of this part of the algorithm is $O(f^2 \cdot p \cdot d^2) = O(f^2 d^2 p)$.

Overall, then, the deterministic version of Berlekamp's algorithm is $O(d^3 + d^2 f^2 p)$. This is undesired, as it is exponential in the number of bits in $p$. However, the probabilistic version performs much better.

In the probabilistic version of Berlekamp's algorithm, we keep the initial $O(d^3)$ step. Then we begin a loop that again runs until all of the factors are found. This time, we pick $f$ random constants and use them to generate a linear combination $t$ of the basis vectors, which is $O(fd)$. Then, for each polynomial factor $z$ already in the list, we compute the polynomial $t^{(p-1)/2}$, reduced modulo $z$. Using modular exponentiation by squaring and reducing, this can be done in $O(d^2 \log \frac{p-1}{2}) = O(d^2 \log p)$ time. Lastly, we find the GCD with the factor, which is $O(d^2)$.

For each iteration of the outermost loop, we are guaranteed (according to [3]) that we will find a new factor with positive constant probability, as long as $p \neq 2$. Thus, the expected number of times we run the outer loop is $O(f)$. Inside the loop, we run over each factor already in the list, which results in quadratic behavior in terms of the number of factors for the total number of times this inner loop runs. Thus, the overall runtime of these loops is $O(f \cdot f \cdot d^2 \log p) = O(d^2 f^2 \log p)$.

The expected runtime of this version of Berlekamp's algorithm is $O(d^3 + d^2 f^2 \log p)$. Note that this is an improvement from $p$ to $\log p$ over the deterministic version. As mentioned earlier, this expected runtime is guaranteed as long as $p \neq 2$, so we may use it for all sufficiently large $p$ to maintain the $\log p$ behavior overall.

## 3.4 Hensel Lifting

As explained in the previous section, Hensel lifting as the author has implemented it consists of four subroutines. The first two, the single Hensel lift and quadratic Hensel lift, have no internal loops apart from those found in polynomial arithmetic. Thus, the limiting factor for runtime is the fact that both algorithms require a constant number of polynomial multiplications and divisions. So both of these algorithms are $O(d^2)$.

The multiple Hensel lift used to raise a product of two factors mod $p$ to a product mod $p^e$ works as follows. First, we run the Euclidean algorithm on the two factors to find $U$ and $V$, which is $O(d^2)$. Then, we perform $n$ iterations of the Hensel lift and quadratic Hensel lift, where $n$ satisfies $2^n \geq e$. Thus $n$ is proportional to $\log e$.

There is one final step: after this lifting is done, we may have lifted "too far" $(2^n > e)$; in order to prevent storing unnecessarily high coefficients, the author's implementation reduces the coefficients of the final lifting mod $p^e$. This means we must compute $p^e$, which could be done by squaring, but as this turns out not to be significant in the greater scope, the author's implementation does it naively, with $e$ multiplications. So the overall runtime of the multiple Hensel lift is $O(d^2 \log e + e)$.

When we have more than two factors, say $f$ factors, we simply run the multiple Hensel lift on factor pairs in succession, as stated earlier. The only additional time overhead consists of $f$ polynomial multiplications, which are dominated by the runtime of the actual Hensel lifts. So this full polynomial Hensel lift is $O(d^2 f \log e + fe)$.

## 3.5 Factoring over $\mathbb{Z}$

Factoring over $\mathbb{Z}$ is a complicated process. Most of the operations involved are simple polynomial arithmetic and thus bounded by $d^2$, but a few steps in the process are not and deserve further analysis.

The first of these complex steps involves finding an appropriate prime to use for Berlekamp's algorithm. The obvious way to do this is to continue trying prime numbers until one works. In order to analyze this, we must first put a bound on how large the prime we need eventually is. Iványi proves as Corollary 5.77 [6] that the smallest prime $p$ modulo which the polynomial is still squarefree is $O((d \log d + 2d \log |a|)^2)$, where $|a|$ is the polynomial's $l_2$-norm. Note that we must make the assumption that $|a|$ is polynomial in terms of $d$, so that $\log |a|$ and $\log d$ are equal up to a constant factor. This falls under our assumption from earlier that the rational numbers involved do not become too large; in short, we are giving an analysis of the runtime as $d$ increases but $|a|$ stays relatively fixed in comparison, which is behavior that the author expects would occur in applications as the algorithm's use is scaled up.

Thus, $p$ is $O(d^2 \log^2 d)$. The number of primes we will try to use is then $O(\pi(d^2 \log^2 d))$, which is $O(\frac{d^2 \log^2 d}{\log(d^2 \log^2 d)})$ by the prime number theorem. As $\log(d^2 \log^2 d) \geq \log d$ for large $d$, this can be relaxed to $O(d^2 \log d)$.

So we will need to test the primality of $O(d^2 \log^2 d)$ numbers bounded above by $p \in O(d^2 \log^2 d)$. The author's implementation, as stated, uses the Miller-Rabin test a constant number of times, relying on the extremely low probability of a pseudoprime.

Each Miller-Rabin test for a number $k$ is $O(\log^3 k)$, so all of the primality tests combined will be $O(d^2 \log^2 d \log^3(d^2 \log^2 d)) = O(d^2 \log^5 d)$.

Every time we find a prime, we then test if our polynomial is squarefree modulo that prime. This test is a polynomial GCD that has $O(d^2)$ runtime, and as stated it runs $O(d^2 \log d)$ times. This dominates the cost of primality testing, and the step of finding an appropriate prime is $O(d^4 \log d)$.

After finding a prime, the next major step is to run Berlekamp's algorithm, which is $O(d^3 + d^2 f^2 \log d)$. As $f \leq d$ this is dominated by the above $O(d^4 \log d)$ step.

We next need to compute the appropriate value of $e$ and perform a Hensel lift. As stated earlier, we compute the bound $B$ using binomial coefficients as, up to rounding, $B = \binom{d/2-1}{d/4}|a| + \binom{d/2-1}{d/4-1}|l(a)|$. This computation, and determining the value of $e$ such that $p^e$ is sufficiently large, together take $O(d^2)$ time and are thus insignificant.

Then $e$ is $O(\log B)$, which is $O(\log(\binom{d/2}{d/4}|a|)) = O(\log \binom{d/2}{d/4} + \log |a|)$. As assumed earlier, $\log |a|$ is at worst proportional to $d$.

By Stirling's formula or the Wallis product, we can determine the asymptotic complexity of a central binomial coefficient as $\binom{2n}{n} = O(\frac{4^n}{\sqrt{n}})$. Letting $n = \frac{d}{4}$ gives us $\log \binom{d/2}{d/4} = O(\log \frac{4^{d/4}}{\sqrt{d/4}}) = O(\log 4^{d/4}) = O(d)$. So $e$ is $O(d)$.

The Hensel lift, as mentioned in the previous subsection, is $O(d^2 f \log e + fe)$. Knowing that $f$ and $e$ are both $O(d)$, this is $O(d^3 \log d)$ and thus dominated by other terms.

The other major step is the step in which we try combinations of factors mod $p^e$ to see if any of them yield factors over $\mathbb{Z}$. At worst, we can envision a scenario where a polynomial splits into $f$ factors mod $p$, but over $\mathbb{Z}$ it has two factors, each the product of $\frac{f}{2}$ of its factors mod $p$. In this case, the algorithm will try every nonempty subset of factors of cardinality less than $\frac{f}{2}$ before finding the correct ones. The number of subsets of cardinality less than $\frac{f}{2}$ tried in this case is:

$$\sum_{r=1}^{\frac{f}{2}-1} \binom{f}{r}$$

$$= \frac{1}{2}\left(\sum_{r=0}^{\frac{f}{2}-1} \binom{f}{r} + \sum_{r=0}^{\frac{f}{2}-1} \binom{f}{r}\right) - \binom{f}{0}$$

$$= \frac{1}{2}\left(\sum_{r=0}^{\frac{f}{2}-1} \binom{f}{r} + \sum_{r=\frac{f}{2}+1}^{f} \binom{f}{f-r}\right) - 1$$

$$= \frac{1}{2}\left(\sum_{r=0}^{\frac{f}{2}-1} \binom{f}{r} + \sum_{r=\frac{f}{2}+1}^{f} \binom{f}{r}\right) - 1$$

$$= \frac{1}{2}\left(\sum_{r=0}^{f} \binom{f}{r} - \binom{f}{\frac{f}{2}}\right) - 1$$

$$
\begin{aligned}
&= O\left(\sum_{r=0}^{f} \binom{f}{r} - \binom{f}{\frac{f}{2}}\right) \\
&= O\left(2^f - \frac{4^{f/2}}{\sqrt{f/2}}\right) \\
&= O\left(2^f - \frac{2^f}{\sqrt{f}}\right) \\
&= O(2^f).
\end{aligned}
$$

This is bad news, because it is exponential in $f$. It would thus be good to know whether this bound is ever achieved. Indeed, we can construct examples of this case.

Let $p_k$ be the $k$th prime for $k > 1$. Let $a_1(x) = (x+1)(x+3)\dots(x+p_k-2)$ and $a_2(x) = (x+2)(x+4)\dots(x+p_k-1)$. Let $n = \prod_{i \le k} p_i$. Then using some simple modular calculations, we can always add a multiple of $n$ to each non-leading coefficient of $a_1$ and $a_2$ such that the coefficient is a multiple of $p_{k+1}$ but not of $p_{k+1}^2$, forming new polynomials $b_1 \equiv a_1 \pmod{n}$ and $b_2 \equiv a_2 \pmod{n}$.

Then $b_1 b_2$ is a polynomial of degree $p_k - 1$, and it is congruent to $(x+1)(x+2)\dots(x+p_k-1)$ modulo any prime less than or equal to $p_k$. For the first $k-1$ of these primes $p_i$, $b_1 b_2$ is no longer squarefree mod $p_i$, so we cannot use $p_i$ for the algorithm. We can use $p_k$ and split $b_1 b_2$ into $p_k - 1$ linear factors. But by Eisenstein's criterion with the prime $p_{k+1}$ the polynomials $b_1$ and $b_2$ are irreducible over $\mathbb{Z}$, so $b_1 b_2$ only splits into two factors over $\mathbb{Z}$, each of which has degree $\frac{p-1}{2}$. This is precisely the worst case that we have been attempting to produce.

In the worst case, then, we need to check $O(2^f)$ subsets of factors. The checking process has several steps, the first and slowest of which is simply multiplying all of the factors in the subset. The cardinality of a subset of less than half of the factors is, on average, proportional in the limit to the total number of factors mod $p$; this can be seen by noting that the half-normal distribution has finite mean and noting that the distribution of the cardinality of a subset approaches a half-normal distribution in the limit. Thus, this multiplication step cannot be said to be faster than $O(fd^2)$.

Thus, overall, we find that the process of factoring a polynomial over $\mathbb{Z}$ is $O(d^4 \log d + 2^f f d^2)$. If $f$ is small, the first term may dominate, but if the polynomial factors into many factors mod $p$, $f$ will be proportional to $d$ and cause exponential runtime. Thus, we may remove $f$ from this expression and simply say that the algorithm is $O(2^d d^3)$.

## 3.6 Factoring over $\mathbb{Q}$

As stated before, this uses Gauss's lemma. As such, the entire algorithm consists of only a single factorization over $\mathbb{Z}$ and several $O(d)$ operations. Its worst-case runtime is $O(2^d d^3)$ as with factoring over $\mathbb{Z}$.

## 3.7 Factoring over $K(\alpha)$

Note that this algorithm is a bit more generic than the previous ones; it is valid for any number field, not only for a simple extension of $\mathbb{Q}$. We will denote the degree of the polynomial by $d$, the degree of $\alpha$ over $K$ by $n$, and the product of the degrees of all extensions constructing $K(\alpha)$ from $\mathbb{Q}$ by $k$.

The first few steps of this algorithm are quadratic operations such as polynomial GCD and division. These are $O(d^2k^2)$. The first major step in the algorithm is a loop in which we search for an appropriate shift under which the polynomial norm is squarefree. The proof of Lemma 3.6.2 in [3] demonstrates that there can be at most one shift under which the norm is *not* squarefree for each quadruple of values $(i_1, i_2, j_1, j_2)$ where $1 \leq j_1, j_2 \leq e$ and $1 \leq i_1, i_2 \leq d$. Thus this loop runs at most $O(d^2n^2)$ times. Inside the loop are more operations of complexity $O(d^2k^2)$.

After the loop, we factor in the base field. We leave this runtime as a variable $F$ for now. Lastly, there is another loop of at most $d$ iterations whose body consists of operations of complexity $O(d^2k^2)$. Thus, the overall runtime of this algorithm is $O(d^4k^2n^2+F)$. When the base field is $\mathbb{Q}$, $F \in O(2^d d^3)$; otherwise, $F$ is the runtime of this same algorithm, one level down.

Thus, we have another recurrence relation. Letting $F_i$ be the runtime for factoring in extension $i$ over $\mathbb{Q}$, with $F_0$ the runtime over $\mathbb{Q}$ itself, we have $F_0 = O(2^d d^3)$. When we move up one level from $F_i$ to $F_{i+1}$, we add a term $d^4k_i^2n_i^2$ as above, where $k$ and $n$ have now been indexed by $i$. So $F_{i+1} = O(d^4n_i^2 \prod_{j=1}^{i} n_j^2 + F_i)$. We can collapse this recurrence, by weakening it slightly, to $F_i = O(d^4 \sum_{j=1}^{i} n_i^2 \prod_{j=1}^{i} n_i^2 + 2^d d^3)$. Letting $s$ denote the sum of the squares of the extension degrees and $k^2$ their product, we have $O(d^4sk^2 + 2^d d^3)$ as the final runtime of this algorithm. (Note that the second term will dominate in most cases, unless we are factoring small polynomials in large extensions.)

## 4 Conclusion

Factoring is a complex process. Many substeps are required in order to factor polynomials over a field like $\mathbb{Q}$ or a number field. Overall, the slowest step in this process is the factor-combining step allowing us to move from a finite field to an infinite field; unfortunately, the runtime of this step is exponential. As such, ways to make this step more efficient would be a welcome change to the overall procedure.

One avenue to explore would be the potential application of parallel and quantum computing to this algorithm. There are some major steps in this algorithm that could benefit from parallel processing, such as the trial of random vectors in the probabilistic version of Berlekamp's algorithm, or the trial of subsets of factors in the factor-combining step mentioned in the previous paragraph. As quantum algorithms such as Shor's have shown promise in related problems, the author suspects that a quantum approach may be helpful here as well.

Finally, one may explore how to extend this problem to that of factoring over the *ring of integers* of a number field. The ring of integers of a number field is a ring containing only those elements that are roots of polynomials with integer coefficients and leading coefficient 1. It is so called because the ring of integers of $\mathbb{Q}$ is the ring $\mathbb{Z}$, and rings of integers in some cases have an analogous relationship to their corresponding fields. Factoring over a ring of integers requires "a little extra work" according to Cohen ([3], p. 144), because Gauss's lemma does not necessarily hold in these cases; among other issues, factorization is no longer necessarily unique.

## References

[1] Margherita Barile, "Conjugate Elements," *MathWorld – A Wolfram Web Resource,* http://mathworld.wolfram.com/ConjugateElements.html.

[2] Margherita Barile, "Finite Extension," *MathWorld – A Wolfram Web Resource,* http://mathworld.wolfram.com/FiniteExtension.html.

[3] Henri Cohen, *A Course in Computational Algebraic Number Theory,* Springer-Verlag Berlin Heidelberg, 1993.

[4] Free Software Foundation, "GMP: The GNU Multiple Precision Arithmetic Library," https://gmplib.org.

[5] Gaël Guennebaud and Benoît Jacob and others, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[6] Antal Iványi, *Algorithms of Informatics vol. 1,* Kempelen Farkas Hallgatói Információs Központ, 2001.

[7] Charles C. Pinter, *A Book of Abstract Algebra,* Dover Publications, 2010.

[8] Todd Rowland and Eric W. Weisstein, "Extension Field," *MathWorld – A Wolfram Web Resource,* http://mathworld.wolfram.com/ExtensionField.html.

[9] John Swallow, *Exploratory Galois Theory,* Cambridge University Press, 2004.

[10] Eric W. Weisstein, "Field," *MathWorld – A Wolfram Web Resource,* http://mathworld.wolfram.com/Field.html.

[11] Todd Rowland and Eric W. Weisstein, "Number Field," *MathWorld – A Wolfram Web Resource,* http://mathworld.wolfram.com/NumberField.html.

[12] David Y. Y. Yun, "On square-free decomposition algorithms," *Proceedings of the third ACM symposium on Symbolic and algebraic computation,* 1976: 26-35.