Abstract. We present a new algorithm for computing \( m \)-th roots over the finite field \( \mathbb{F}_q \), where \( q = p^n \), with \( p \) a prime, and \( m \) any positive integer. In the particular case \( m = 2 \), the cost of the new algorithm is an expected \( O(M(n) \log(p) + C(n) \log(n)) \) operations in \( \mathbb{F}_p \), where \( M(n) \) and \( C(n) \) are bounds for the cost of polynomial multiplication and modular polynomial composition. Known results give \( M(n) = O(n \log(n) \log \log(n)) \) and \( C(n) = O(n^{1.67}) \), so our algorithm is subquadratic in \( n \).

1. Introduction

Beside its intrinsic interest, computing \( m \)-th roots over finite fields (for \( m \) an integer at least equal to 2) has found many applications in computer science. Our own interest comes from elliptic and hyperelliptic curve cryptography; there, square root computations show up in pairing-based cryptography [3] or point-counting problems [8].

Our result in this paper is a new algorithm for computing \( m \)-th roots in a degree \( n \) extension \( \mathbb{F}_q \) of the prime field \( \mathbb{F}_p \), with \( p \) a prime. Our emphasis is on the case where \( p \) is thought to be small, and the degree \( n \) grows. Roughly speaking, we reduce the problem to \( m \)-th root extraction in a lower degree extension of \( \mathbb{F}_p \) (when \( m = 2 \), we actually reduce the problem to square root extraction over \( \mathbb{F}_p \) itself).

Our complexity model. It is possible to describe the algorithm in an abstract manner, independently of the choice of a basis of \( \mathbb{F}_q \) over \( \mathbb{F}_p \). However, to give concrete complexity estimates, we have to decide which representation we use, the most usual choices being monomial and normal bases. We choose to use a monomial basis, since, in particular, our implementation is based on the library NTL [22], which uses this representation. Thus, the finite field \( \mathbb{F}_q \) is represented as \( \mathbb{F}_p[X]/\langle f \rangle \), for some monic irreducible polynomial \( f \in \mathbb{F}_p[X] \) of degree \( n \); elements of \( \mathbb{F}_q \) are represented as polynomials in \( \mathbb{F}_p[X] \) of degree less than \( n \). We will briefly mention the normal basis representation later on.

The costs of all algorithms are measured in number of operations \(+, \times, \div\) in the base field \( \mathbb{F}_p \) (that is, we are using an algebraic complexity model) — at the end of this introduction, we discuss how our results can be stated in the Boolean model, especially in light of results by Umans [25] and Kedlaya and Umans [13].

We shall denote upper bounds for the cost of polynomial multiplication and modular composition, respectively, by \( M(n) \) and \( C(n) \). This means that over any field...
K, we can multiply polynomials of degree n in K[X] in M(n) base field operations, and that we can compute f(g) mod h in C(n) operations in K, when f, g, h are degree n polynomials. We additionally require that both M and C are super-linear functions, as in [26, Chapter 8], and that M(n) = O(C(n)). In particular, since we work in the monomial basis, multiplications and inversions in F_q can be done in, respectively, O(M(n)) and O(M(n) log(n)) operations in F_p; again see [26].

The best known bound for M(n) is O(n log(n) log log(n)), achieved by using Fast Fourier Transform [19, 5]. The most well-known bound for C(n) is O(n^{(\omega+1)/2}), due to Brent and Kung [4], where \omega is such that matrices of size n over any field K can be multiplied in O(n^\omega) operations in K; this estimate assumes that \omega > 2, otherwise some logarithmic terms may appear. Using the algorithm of Coppersmith and Winograd [6], we can take \omega < 2.37 and thus C(n) = O(n^{1.69}); an algorithm by Huang and Pan [10] actually achieves a slightly better exponent of 1.67, by means of rectangular matrix multiplication.

Main result. We will focus in this paper on the case of t-th root extraction, where t is a prime divisor of \( q - 1 \); the general case of m-th root extraction, with m arbitrary, can easily be reduced to this case (see the discussion after Theorem 1.1).

The core of our algorithm is a reduction of t-th root extraction in \( F_q \) to t-th root extraction in an extension of \( F_p \) of smaller degree. Our algorithm is probabilistic of Las Vegas type, so its running time is given as an expected number of operations. With this convention, our main result is the following.

**Theorem 1.1.** Let t be a prime factor of \( q - 1 \), with \( q = p^n \), and let s be the order of p in \( \mathbb{Z}/t\mathbb{Z} \). Given \( a \in F_q^* \), one can decide if a is a t-th power in \( F_q^* \), and if so compute one of its t-th roots, by means of the following operations:

- an expected \( O(sM(n) \log(p) + C(n) \log(n)) \) operations in \( F_p \);
- a t-th root extraction in \( F_{p^s} \).

Thus, we replace the t-th root extraction in a degree n extension by a t-th root extraction in an extension of degree \( s \leq \min(n, t) \). The extension degree \( s \) is the largest one for which \( t \) still divides \( p^s - 1 \), so iterating the process does not bring any improvement: the t-th root extraction in \( F_{p^s} \) must be dealt with by another algorithm. The smaller \( s \) is, the better.

A useful special case is \( t = 2 \), that is, we are taking square roots; the assumption that \( t \) divides \( q - 1 \) is then satisfied for all odd primes \( p \) and all \( n \). In this case, we have \( s = 1 \), so the second step amounts to square root extraction in \( F_p \). Since this can be done in \( O(\log(p)) \) expected operations in \( F_p \), the total running time of the algorithm is an expected \( O(M(n) \log(p) + C(n) \log(n)) \) operations in \( F_p \).

A previous algorithm by Kaltofen and Shoup [12] allows one to compute t-th roots in \( F_{p^s} \) in expected time \( O((M(t)M(n) \log(p) + tC(n) + C(t)M(n)) \log(n)) \); we discuss it further in the next section. This algorithm requires no assumption on \( t \), so it can be used in our algorithm in the case \( s > 1 \), for t-th root extraction in \( F_{p^s} \). Then, its expected running time is \( O((M(t)M(s) \log(p) + tC(s) + C(t)M(s)) \log(s)) \).

The strategy of using Theorem 1.1 to reduce from \( F_q \) to \( F_{p^s} \) then using the Kaltofen-Shoup algorithm over \( F_{p^s} \) is never more expensive than using the Kaltofen-Shoup algorithm directly over \( F_q \). For \( t = O(1) \), both strategies are within a constant factor; but even for the smallest case \( t = 2 \), our algorithm has advantages (as explained in the last section). For larger \( t \), the gap in our favor will increase for cases when \( s \) is small (such as when \( t \) divides \( p - 1 \), corresponding to \( s = 1 \)).
Finally, let us go back to the remark above, that for any \( m \), one can reduce \( m \)-th root extraction of \( a \in \mathbb{F}_q^* \) to computing \( t \)-th roots, with \( t \) dividing \( q - 1 \); this is well known; see for instance [1] Chapter 7.3. We write \( m = vw \) with \( (v, q - 1) = 1 \) and \( t \mid q - 1 \) for every prime divisor \( t \) of \( u \), and we assume that \( a \) is indeed an \( m \)-th power.

- We first compute the \( v \)-th root \( a_0 \) of \( a \) as \( a_0 = a^{v^{-1} \mod q - 1} \) by computing the inverse \( \ell \) of \( v \mod q - 1 \), and computing an \( \ell \)-th power in \( \mathbb{F}_q \). This takes \( O(nM(n) \log(p)) \) operations in \( \mathbb{F}_p \).
- Let \( u = \prod_{i=1}^d m_i^{\alpha_i} \) be the prime factorization of \( u \), which we assume is given to us. Then, for \( k = 1, \ldots, \alpha_1 \), we compute an \( m_1 \)-th root \( a_k \) of \( a_{k-1} \) using Theorem 1.1 so that \( a_{\alpha_1} \) is an \( m_1^{\alpha_1} \)-th root of \( a_0 \).

One should be careful in the choice of the \( m_1 \)-th roots (which are not unique), so as to ensure that each \( a_k \) is indeed an \( u/m_1^{\alpha_1} \)-th power: if the given \( a_k \) is not such a power, we can multiply it by a \( m_1 \)-th root of unity until we find a suitable one. The root of unity can be found by the algorithm of Theorem 1.1.

Once we know \( a_{\alpha_1} \), the same process can be applied to compute an \( m_2^{\alpha_2} \)-th root of \( a_{\alpha_1} \), and so on.

The first step, taking a root of order \( v \), may actually be the bottleneck of this scheme. When \( v \) is small compared to \( n \), it may be better to use here, as well the algorithm by Kaltofen and Shoup mentioned above.

**In a Boolean model.** In our algebraic model, when \( n \) grows, the bottleneck of our algorithm (or of the Kaltofen-Shoup algorithm) is modular composition, since there is currently no known algorithm with cost quasi-linear in \( n \).

If we analyze running times in a Boolean model (counting bit or word operations), we may get a better result: Kedlaya and Umans [13], following previous work by Umans [25], give an algorithm with a Boolean cost that grows like \( n^{1+\varepsilon \log(p)\log^{1+o(1)}} \) for modular composition in degree \( n \) over \( \mathbb{F}_p \), for any given \( \varepsilon > 0 \). They also show that the minimal polynomial of an element of \( \mathbb{F}_q = \mathbb{F}_p^n \) can be computed for the same cost.

The algorithms of the following sections can be analyzed in the Boolean model without difficulty (for definiteness, over a Boolean RAM with logarithmic access cost — the Kedlaya-Umans algorithm uses table lookup in large tables). The only differences are in the cost of modular composition over \( \mathbb{F}_p \), as well as minimal polynomial computation in \( \mathbb{F}_q \), for which we use the results by Kedlaya and Umans. Using the fact that arithmetic in \( \mathbb{F}_p \) can be done in Boolean time \( \log(p)^{1+o(1)} \), the running time reported in Theorem 1.1 then becomes \( O(sM(n)\log(p)^{2+o(1)} + n^{1+\varepsilon \log(p)\log^{1+o(1)}}) \) bit operations, for any \( \varepsilon > 0 \); this admits the upper bound \( O(sn^{1+\varepsilon \log(p)\log^{1+o(1)}}) \). With respect to the extension degree \( n \), this is close to being linear time.

From the practical point of view, however, we did not use the Kedlaya-Umans algorithm in our experiments, since we do not have a competitive implementation of it (and currently know of no such implementation).

**Organization of the paper.** The next section reviews and discusses known algorithms; Section 3 gives the details of the root extraction algorithm and some experimental results. Throughout the paper, \( (\mathbb{F}_q^*)^t \) denotes the set of \( t \)-th powers in \( \mathbb{F}_q^* \).
2. Previous work

Let \( t \) be a prime factor of \( q - 1 \). In the rest of this section, we discuss previous algorithms for \( t \)-th root extraction, with a special focus on the case \( t = 2 \) (square roots), which has attracted the most attention in the literature. Note that our assumptions exclude the case of \( p \)-th root extraction in characteristic \( p \).

We shall see in Section 3 that given a prime \( t \) as above, the cost of testing for \( t \)-th power is always dominated by the \( t \)-th root extraction; thus, for an input \( a \in \mathbb{F}_q^* \), we always assume that \( a \in (\mathbb{F}_q^*)^t \).

All algorithms discussed below rely on some form of exponentiation in \( \mathbb{F}_q \), or in an extension of \( \mathbb{F}_q \), with exponents that grow linearly with \( q \). As a result, a direct implementation using binary powering uses \( \mathcal{O}(\log(q)) \) multiplications in \( \mathbb{F}_q \), that is, \( \mathcal{O}(nM(n) \log(p)) \) operations in \( \mathbb{F}_p \). Even using fast multiplication, this is quadratic in \( n \); alternative techniques should be used to perform the exponentiation, when possible.

**Some special cases of square root computation.** If \( G \) is a group with an odd order \( s \), then the mapping \( f : G \rightarrow G, f(a) = a^2 \) is an automorphism of \( G \); hence, every element \( a \in G \) has a unique square root, which is \( a^{(s+1)/2} \). Thus, if \( q \equiv 3 \pmod{4} \), the square root of any \( a \in (\mathbb{F}_q^*)^2 \) is \( a^{(q+1)/4} \); this is because \( (\mathbb{F}_q^*)^2 \) is a group of odd order \( (q-1)/2 \).

More complex schemes allow one to compute square roots for some increasingly restricted classes of prime powers \( q \). The following table summarizes results known to us; in each case, the algorithm uses \( \mathcal{O}(1) \) exponentiations and \( \mathcal{O}(1) \) additions / multiplications in \( \mathbb{F}_q \). The table indicates what exponents are used in the exponentiations.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( q )</th>
<th>exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>folklore</td>
<td>3 (mod 4)</td>
<td>((q + 1)/4)</td>
</tr>
<tr>
<td>Atkin</td>
<td>5 (mod 8)</td>
<td>((q - 5)/8)</td>
</tr>
<tr>
<td>Müller ([15])</td>
<td>9 (mod 16)</td>
<td>((q - 1)/4 ) and ((q - 9)/16)</td>
</tr>
<tr>
<td>Kong et al. ([14])</td>
<td>9 (mod 16)</td>
<td>((q - 9)/8 ) and ((q - 9)/16)</td>
</tr>
</tbody>
</table>

As was said above, using a direct binary powering approach to exponentiation, all these algorithms use \( \mathcal{O}(nM(n) \log(p)) \) operations in \( \mathbb{F}_p \). Some extensions to higher roots have appeared in the literature, for example, to cube roots in the case where \( q = 7 \pmod{9} \) \([17]\), with similar running times.

**Cipolla’s square root algorithm.** To compute the square root of \( a \in (\mathbb{F}_q^*)^2 \), Cipolla’s algorithm uses an element \( b \) in \( \mathbb{F}_q \) such that \( b^2 - 4a \) is not a square in \( \mathbb{F}_q \). Then, the polynomial \( f(Y) = Y^2 - bY + a \) is irreducible over \( \mathbb{F}_q \), hence \( \mathbb{k} = \mathbb{F}_q[Y]/(f) \) is a field. Let \( y \) be the residue class of \( Y \) modulo \( (f) \). Since \( f \) is the minimal polynomial of \( y \) over \( \mathbb{F}_q \), \( N_{\mathbb{k}/\mathbb{F}_q}(y) = a \), ensuring that \( \sqrt{a} = Y^{(q+1)/2} \mod (Y^2 - bY + a) \).

Finding a quadratic nonresidue of the form \( b^2 - 4a \) by choosing a random \( b \in \mathbb{F}_q \) requires an expected \( \mathcal{O}(1) \) attempts \([1]\), page 158. The quadratic residue test, and
the norm computation take $O(M(n) \log(n) + \log(p))$ and $O(nM(n) \log(p))$ multiplications in $\mathbb{F}_p$, respectively. Therefore, the cost of the algorithm is an expected $O(nM(n) \log(p))$ operations in $\mathbb{F}_p$.

Algorithms extending Cipolla’s to the computation of $t$-th roots in $\mathbb{F}_p$, where $t$ is a prime factor of $p - 1$, are in [27] [28] [16].

**The Tonelli-Shanks algorithm.** We will describe the algorithm in the case of square roots, although the ideas extend to higher orders. Tonelli’s algorithm [23] and Shanks’ improvement [20] use discrete logarithms to reduce the problem to a square roots, although the ideas extend to higher orders. Tonelli’s algorithm [23]

We will describe the algorithm in the case of $\mathbb{F}_q^*$, where $q$ is a prime of order $\ell$. Let $q = 1 = 2^r \ell$, with $(\ell, 2) = 1$ and let $H$ be the unique subgroup of $\mathbb{F}_q^*$ of order $\ell$. Assume we find a quadratic nonresidue $g \in \mathbb{F}_q^*$; then, the square root of $a \in \mathbb{F}_q^*$ can be computed as follows: we can express $a$ as $g^s h \in g^s H$ by solving a discrete logarithm in $\mathbb{F}_q^*/H$; $s$ is necessarily even, so that $\sqrt{a} = g^{s/2} h^{(\ell - 1)/2}$.

According to [18], the discrete logarithm requires $O(r^2 M(n))$ multiplications in $\mathbb{F}_p$; all other steps take $O(nM(n) \log(p))$ operations in $\mathbb{F}_p$. Hence, the expected running time of the algorithm is $O((r^2 + n \log(p)) M(n))$ operations in $\mathbb{F}_p$. Thus, the efficiency of this algorithm depends on the structure of $\mathbb{F}_q^*$: there exists an infinite sequence of primes for which the cost is $O(n^2 M(n) \log(p)^2)$; see [24]. An extension to the computation of cube roots modulo $p$ is in [17], with a similar cost analysis.

**Improving the exponentiation.** All algorithms seen previously use at best $O(nM(n) \log(p))$ operations in $\mathbb{F}_p$, because of the exponentiation. Using ideas going back to [11], Barreto et al. [3] observed that for some of the cases seen above, the exponentiation can be improved, giving a cost subquadratic in $n$.

For instance, when taking square roots with $q = 3 \pmod{4}$, the exponentiation $a^{(q+1)/4}$ can be reduced to computing $a^{1+u+\cdots+u^{(n-3)/2}}$, with $u = p^2$, plus two (cheap) exponentiations with exponents $p(p - 1)$ and $(p + 1)/4$. The special form of the exponent $1 + u + \cdots + u^{(n-3)/2}$ makes it possible to apply a binary powering approach, involving $O(\log(n))$ multiplications and exponentiations, with exponents that are powers of $p$.

Further examples for square roots are discussed in [14] [9], covering the entries of Table 1. These references assume a normal basis representation; using (as we do) the monomial basis and modular composition techniques (which will be explained in the next section), the costs become $O(M(n) \log(p) + C(n) \log(n))$. Some cases of higher index roots are in [2]: if $t$ is a factor of $p - 1$, such that $t^2$ does not divide $p - 1$, and if $\gcd(n, t) = 1$, then $t$-th root extraction can be done using $O(tM(n) \log(p) + C(n) \log(n))$ operations in $\mathbb{F}_p$.

**Kaltofen and Shoup’s algorithm.** Finally, we mention what is, as far as we know, the only algorithm achieving an expected subquadratic running time in $n$ (using the monomial basis representation), without any assumption on $p$.

Consider a factor $t$ of $q - 1$. To compute a $t$-th root of $a \in (\mathbb{F}_q^*)^t$, the idea is simply to factor $Y^t - a \in \mathbb{F}_q[Y]$ using polynomial factorization techniques. Since we know that $a$ is a $t$-th power, this polynomial splits into linear factors, so we can use an Equal Degree Factorization (EDF) algorithm.
A specialized EDF algorithm, dedicated to the case of high-degree extension of a given base field, was proposed by Kaltofen and Shoup [12]. It mainly reduces to the computation of a trace-like quantity \( b + b^p + \cdots + b^{p^{n-1}} \), where \( b \) is a random element in \( \mathbb{F}_q[Y]/(Y^t - a) \). Using a binary powering technique similar to the one of the previous paragraph, this results in an expected running time of \( O((\ell M(n) \log(p) + t C(n) + C(t) M(n)) \log(n)) \) operations in \( \mathbb{F}_p \); remark that this estimate is faster than what is stated in [12] by a factor \( \log(t) \), since here we only need one root, instead of the whole factorization. In the particular case \( t = 2 \), this becomes \( O((\ell M(n) \log(p) + C(n)) \log(n)) \). This achieves a running time subquadratic in \( n \).

This idea actually allows one to compute a \( t \)-th root, for arbitrary \( t \): starting from the polynomial \( Y^t - a \), we apply the above algorithm to \( \gcd(Y^t - a, Y^q - Y) \); computing \( Y^q \) modulo \( Y^t - a \) can be done by the same binary powering techniques.

3. A new root extraction algorithm

In this section, we focus on \( t \)-th root extraction in \( \mathbb{F}_q \), for \( t \) a prime dividing \( q - 1 \) (as we saw in Section 1 \( m \)-th root extraction, for an integer \( m \geq 2 \), reduces to taking \( t \)-th roots, where \( t \) is a prime factor of \( m \) dividing \( q - 1 \)).

The algorithm we present uses the trace \( \mathbb{F}_q \rightarrow \mathbb{F}_{q'} \), for some subfield \( \mathbb{F}_{q'} \subset \mathbb{F}_q \) to reduce \( t \)-th root extraction in \( \mathbb{F}_q \) to \( t \)-th root extraction in \( \mathbb{F}_{q'} \). We assume as before that the field \( \mathbb{F}_q \) is represented by a quotient \( \mathbb{F}_p[X]/(f) \), with \( f(X) \in \mathbb{F}_p[X] \) a monic irreducible polynomial of degree \( n \). We let \( x \) be the residue class of \( X \) modulo \( (f) \).

Since we will handle both \( \mathbb{F}_q \) and \( \mathbb{F}_{q'} \), conversions may be needed. We recall that the minimal polynomial \( g \in \mathbb{F}_p[Z] \) of an element \( b \in \mathbb{F}_q \) can be computed in \( O(C(n) + M(n) \log(n)) \) operations in \( \mathbb{F}_p \) [21]. Then, \( \mathbb{F}_{q'} = \mathbb{F}_p[Z]/(g) \) is a subfield of \( \mathbb{F}_q = \mathbb{F}_p[X]/(f) \); given \( r \in \mathbb{F}_{q'} \), written as a polynomial in \( Z \), we obtain its representation on the monomial basis of \( \mathbb{F}_q \) by means of a modular composition, in time \( C(n) \). We will write this operation \( \text{Embed}(r, \mathbb{F}_q) \). Note that when \( b \) is in \( \mathbb{F}_p \), all these operations are actually free.

3.1. An auxiliary algorithm. We first discuss a binary powering algorithm to solve the following problem. Starting from \( \lambda \in \mathbb{F}_q \), we are going to compute

\[
\alpha_i(\lambda) = \lambda^{1+p^s} + \lambda^{1+p^s+p^{2s}} + \cdots + \lambda^{1+p^s+p^{2s}+\cdots+p^{is}}
\]

for given integers \( i, s > 0 \). This question is similar to (but distinct from) some exponentiations and trace-like computations we discussed before; our solution will be a similar binary powering approach, which will perform \( O(\log(i)) \) multiplications and exponentiations by powers of \( p \). Let

\[
\xi_i = x^{p^{is}}, \quad \zeta_i = \lambda^{p^{is}+p^{2s}+\cdots+p^{is}} \quad \text{and} \quad \delta_i = \lambda^{p^{is}+p^{2s}+\cdots+p^{is}},
\]

where all quantities are computed in \( \mathbb{F}_{q'} \), that is, modulo \( f \); for simplicity, in this paragraph, we will write \( \alpha_i, \zeta_i \) and \( \delta_i \). Note that \( \alpha_i = \lambda \delta_i \), and that we have the following relations:

\[
\xi_1 = x^{p^s}, \quad \zeta_1 = \lambda^{p^s}, \quad \delta_1 = \lambda^{p^s}
\]
and

\[ \xi_i = \begin{cases} \xi_i^{p^{i/2}} & \text{if } i \text{ is even}, \\ \xi_{i-1}^p & \text{if } i \text{ is odd}, \end{cases} \quad \zeta_i = \begin{cases} \xi_i^{p^{i/2}} \xi_{i-1}^{p^{i/2}} & \text{if } i \text{ is even}, \\ \xi_{i-1} \xi_i^p & \text{if } i \text{ is odd}, \end{cases} \]

\[ \delta_i = \begin{cases} \delta_{i/2} + \zeta_i / 2 \delta_{i/2} & \text{if } i \text{ is even}, \\ \delta_{i-1} + \zeta_i & \text{if } i \text{ is odd}. \end{cases} \]

Because we are working in a monomial basis, computing exponentiations to powers of \( p \) is not a trivial task; we will perform them using the following modular composition technique from [7].

Take \( j \geq 0 \) and \( r \in \mathbb{F}_q \), and let \( R \) and \( \Xi_j \) be the canonical preimages of, respectively, \( r \) and \( \xi_j \) in \( \mathbb{F}_p[X] \); then, we have

\[ r^{p^j} = R(\Xi_j) \mod f; \]

see for instance [26, Chapter 14.7]. We will simply write this as \( r(\xi_j) \), and note that it can be computed using one modular composition, in time \( C(n) \). These remarks give us the following recursive algorithm, where we assume that \( \xi_1 = x^{p^s} \) and \( \zeta_1 = \lambda^{p^s} \) are already known.

### Algorithm 1. XiZetaDelta(\( \lambda, i, \xi_1, \zeta_1 \))

**Input:** \( \lambda \), a positive integer \( i \), \( \xi_1 = x^{p^s}, \zeta_1 = \lambda^{p^s} \)

**Output:** \( \xi_i, \zeta_i, \delta_i \)

1. if \( i = 1 \) then
2. return \( \xi_1, \zeta_1, \zeta_1 \)
3. end if
4. \( j \leftarrow \lfloor i/2 \rfloor \)
5. \( \xi_j, \zeta_j, \delta_j \leftarrow \text{XiZetaDelta}(\lambda, j, \xi_1, \zeta_1) \)
6. \( \xi_{2j} \leftarrow \xi_j(\xi_j) \)
7. \( \zeta_{2j} \leftarrow \zeta_j \cdot \zeta_j(\xi_j) \)
8. \( \delta_{2j} \leftarrow \delta_j + \zeta_j \delta_j(\xi_j) \)
9. if \( i \) is even then
10. return \( \xi_{2j}, \zeta_{2j}, \delta_{2j} \)
11. end if
12. \( \xi_i \leftarrow \xi_{2j}(\xi_1) \)
13. \( \zeta_i \leftarrow \zeta_1 \cdot \zeta_{2j}(\xi_1) \)
14. \( \delta_i \leftarrow \delta_{2j} + \zeta_i \)
15. return \( \xi_i, \zeta_i, \delta_i \)

We deduce the following algorithm for computing \( \alpha_i(\lambda) \).

### Algorithm 2. Alpha(\( \lambda, i \))

**Input:** \( \lambda \), a positive integer \( i \)

**Output:** \( \alpha_i \)

1. \( \xi_1 \leftarrow x^{p^s} \)
2. \( \zeta_1 \leftarrow \lambda^{p^s} \)
3. \( \xi_i, \zeta_i, \delta_i \leftarrow \text{XiZetaDelta}(\lambda, i, \xi_1, \zeta_1) \)
4. return \( \lambda \delta_i \)
Proposition 3.1. Algorithm \( \square \) computes \( \alpha_i(\lambda) \) using \( O(C(n) \log(is) + M(n) \log(p)) \) operations in \( \mathbb{F}_p \).

Proof. To compute \( x^{p^s} \) and \( \lambda^{p^s} \) we first compute \( x^p \) and \( \lambda^p \) using \( O(\log(p)) \) multiplications in \( \mathbb{F}_q \), and then do \( O(\log(s)) \) modular compositions modulo \( f \). The depth of the recursion in Algorithm \( \square \) is \( O(\log(i)) \); each recursive call involves \( O(1) \) additions, multiplications and modular compositions modulo \( f \), for a total time of \( O(C(n)) \) per recursive call.

As said before, the algorithm can also be implemented using a normal basis representation. Then, exponentiations to powers of \( p \) become trivial, but multiplication becomes more difficult. We leave these considerations to the reader.

3.2. Taking \( t \)-th roots. We will now give our root extraction algorithm. As said before, we now let \( t \) be a prime factor of \( q - 1 \), and we let \( s \) be the order of \( p \) in \( \mathbb{Z}/t\mathbb{Z} \). Then \( s \) divides \( n \), say \( n = st \).

We first explain how to test for \( t \)-th powers. Testing whether \( a \in \mathbb{F}_q^* \) is a \( t \)-th power is equivalent to testing whether \( a^{(q-1)/t} = 1 \). Let \( \zeta = a^{(p^s-1)/t} \); then \( a^{(q-1)/t} = \zeta^{1+q^{s-1} + \cdots + q^{(t-1)s-1}} \). Computing \( \zeta \) requires \( O(sM(n) \log(p)) \), and computing \( 1^{1+q^{s-1} + \cdots + q^{(t-1)s-1}} \) using Algorithm \( \square \) requires \( O(C(n) \log(n) + M(n) \log(p)) \) operations in \( \mathbb{F}_p \). Therefore, testing for a \( t \)-th power takes \( O(C(n) \log(n) + sM(n) \log(p)) \) operations in \( \mathbb{F}_p \).

In the particular case when \( t \) divides \( p-1 \), we can actually do better: we have \( a^{(q-1)/t} = \text{res}(f, a)^{(p-1)/t} \), where \( \text{res}(\cdot, \cdot) \) is the resultant function. The resultant can be computed using \( O(M(n) \log(n)) \) operations in \( \mathbb{F}_p \), so the whole test can be done using \( O(M(n) \log(n) + \log(p)) \) operations in \( \mathbb{F}_p \).

In any case, we can now assume that we are given \( a \in (\mathbb{F}_q^*)^t \), with \( t \)-th root \( \gamma \in \mathbb{F}_q^* \). Defining \( \beta = T_{\mathbb{F}_q/\mathbb{F}_q'}(\gamma) \), where \( T_{\mathbb{F}_q/\mathbb{F}_q'} : \mathbb{F}_q \to \mathbb{F}_q' \) is the trace linear form and \( q' = p^s \), we have

\[
\beta = \sum_{i=0}^{t-1} \gamma^{p^{is}} = \gamma(1 + \gamma^{p^{s-1}} + \gamma^{p^{2s-1}} + \cdots + \gamma^{p^{(t-1)s-1}}) = \gamma(1 + a^{(p^s-1)/t} + a^{(p^{2s-1})/t} + \cdots + a^{(p^{(t-1)s-1})/t}).
\]

Let \( b = 1 + a^{(p^s-1)/t} + a^{(p^{2s-1})/t} + \cdots + a^{(p^{(t-1)s-1})/t} \), so that equation \( \square \) gives \( \beta = \gamma b \). Taking the \( t \)-th power in both sides results in the equation \( \beta^t = ab^t \) over \( \mathbb{F}_q' \). Since we know \( a \), and we can compute \( b \), we can thus determine \( \beta \) by \( t \)-th root extraction in \( \mathbb{F}_q' \). Then, if we assume that \( b \neq 0 \) (or equivalently that \( \beta \neq 0 \)), we deduce \( \gamma = \beta b^{-1} \); to resolve the issue that \( \beta \) may be zero, we will replace \( a \) by \( a' = ac^t \), for a random element \( c \in \mathbb{F}_q^* \).

Computing the \( t \)-th root of \( a'b^t \) in \( \mathbb{F}_q' \) is done as follows. We first compute the minimal polynomial \( g \in \mathbb{F}_p[Z] \) of \( a'b^t \), and let \( z \) be the residue class of \( Z \) in \( \mathbb{F}_p[Z]/(g) \). Then, we compute a \( t \)-th root \( r \) of \( z \) in \( \mathbb{F}_p[Z]/(g) \), and embed \( r \) in \( \mathbb{F}_q \). The computation of \( r \) is done by a black-box \( t \)-th root extraction algorithm, denoted by \( r \mapsto r^{1/t} \).
It remains to explain how to compute $b$ efficiently. Let $\lambda = a^{(p^s-1)/t}$; then, one verifies that $b = 1 + \lambda + \alpha_{\ell-2}(\lambda)$, so we can use the algorithm of the previous subsection. Putting this all together, we obtain the following algorithm:

**Algorithm 3.** $t$-th root in $\mathbb{F}_q^*$

**Input:** $a \in (\mathbb{F}_q^*)^t$

**Output:** a $t$-th root of $a$

1. $s \leftarrow$ the order of $p$ in $\mathbb{Z}/t\mathbb{Z}$
2. $\ell \leftarrow n/s$
3. repeat
4. choose a random $c \in \mathbb{F}_q$
5. $a^t \leftarrow ac^{\ell^t}$
6. $\lambda \leftarrow a^{(p^s-1)/t}$
7. $b \leftarrow 1 + \lambda + \text{Alpha}(\lambda, \ell - 2)$
8. until $b \neq 0$
9. $g \leftarrow \text{MinimalPolynomial}(a'b^t)$
10. $\beta \leftarrow z^{1/t}$ in $\mathbb{F}_p[Z]/(g)$
11. return $\text{Embed}(\beta, \mathbb{F}_q)b^{-1}c^{-1}$

The following proposition proves Theorem 1.1.

**Proposition 3.2.** Algorithm 3 computes a $t$-th root of $a$ using an expected $O(sM(n)\log(p) + C(n)\log(n))$ operations in $\mathbb{F}_p$, plus a $t$-th root extraction in $\mathbb{F}_{q'}$.

**Proof.** Note first that $\beta = 0$ means that $T_{\mathbb{F}_q/\mathbb{F}_{q'}}(\gamma c) = 0$. There are $q/q'$ values of $c$ for which this is the case, and hence the probability of having a zero trace is $(q/q')/q = 1/q' \leq 1/2$. So we expect to have to choose $O(1)$ elements in $\mathbb{F}_q$ before exiting the repeat … until loop. Each pass in the loop uses $O(sM(n)\log(p))$ operations in $\mathbb{F}_p$ to compute $a'$ and $\lambda$, and $O(C(n)\log(n) + M(n)\log(p))$ operations in $\mathbb{F}_p$ to compute $b$.

Given $a'$ and $b$, one obtains $b^t$ and $a'b^t$ using another $O(sM(n)\log(p))$ operations in $\mathbb{F}_p$; then, computing $g$ takes time $O(C(n) + M(n)\log(n))$.

After the black-box call to $t$-th root extraction modulo $g$, embedding $\beta$ in $\mathbb{F}_q$ takes time $C(n)$. We can then deduce $\gamma$ by two divisions in $\mathbb{F}_q$, using $O(M(n)\log(n))$ operations in $\mathbb{F}_p$; this is negligible compared to the cost of all modular compositions. \hfill \Box

### 3.3. Experimental results.

We have implemented our root extraction algorithm, in the case $m = 2$ (that is, we are taking square roots); our implementation is based on Shoup’s NTL [22]. The experiments in this section were done for the “small” and “large” characteristics $p = 449$ and $p = 348975609381470925634534573457497$, and different values of the extension degree, on a 2.40GHz Intel Xeon. They show that the behavior of our algorithm hardly depends on the characteristic.

Figure 1 compares our algorithm to Cipolla’s and Tonelli-Shanks’ algorithms over $\mathbb{F}_q$, with $q = p^n$, for different values of the extension degree $n$. For each $n$, the running time is averaged over five runs, with input being random squares in $\mathbb{F}_q$.

Remember that the bottleneck in Cipolla’s and Tonelli-Shanks’ algorithms is the exponentiation, which takes $O(nM(n)\log(p))$ operation in $\mathbb{F}_p$. As it turns out,
NTL’s implementation of modular composition has \( \omega = 2 \); this means that with this implementation we have \( C(n) = O(n^2) \), and our algorithm takes expected time \( O(M(n) \log(p) + n^2 \log(n)) \). Although this implementation is not subquadratic in \( n \), it remains faster than Cipolla’s and Tonelli-Shanks’ algorithms, in theory and in practice.

Next, Figure 2 compares our NTL implementation of the EDF algorithm proposed by Kaltofen and Shoup, and our square root algorithm (note that the range of reachable degrees is much larger that in the first figure). We ran the algorithms for five random elements for each extension degree. At this scale, we observe irregularities in the averaged running time of the Kaltofen-Shoup algorithm, due to its probabilistic behavior.

The vertical dashed lines and the green line respectively show the running time range, and the average running time, of Kaltofen and Shoup’s algorithm. In the case of our algorithm (the red graph), the vertical ranges are invisible because the deviation from the average is \( \approx 10^{-2} \) seconds.

This time, the results are closer. Nevertheless, it appears that the running time of our algorithm behaves more “smoothly”, in the sense that random choices seem to have less influence. This is indeed the case. The random choice in Kaltofen and
Shoup's algorithm succeeds with probability about $1/2$; in case of failure, we have to run the whole algorithm again. In our case, our choice of an element $c$ in $\mathbb{F}_q^*$ fails with probability $1/p \ll 1/2$; then, there is still randomness involved in the $t$-th root extraction in $\mathbb{F}_p$, but this step was negligible in the range of parameters where our experiments were conducted.

Another way to express this is to compare the standard deviations in the running times of both algorithms. In the case of Kaltofen-Shoup’s algorithm, the standard deviation is about $1/\sqrt{2}$ of the average running time of the whole algorithm. For our algorithm, the standard deviation is no more than $1/\sqrt{p}$ of the average running time of the trace-like computation (which is the dominant part), plus $1/\sqrt{2}$ of the average running time of the root extraction in $\mathbb{F}_p$ (which is cheap).

Finally, we mention that the crossover point between our algorithm and the previous ones varies with $p$, but is usually small: around $n = 20$ to $n = 30$ for small $p$ (say less than 500) and around $n = 10$ to $n = 20$ for larger values of $p$. For very small degrees, the Tonelli-Shanks algorithm was the fastest in our experiments. Note that for such small $n$, big-O analyses lose some significance; this makes it difficult to get accurate theoretical estimates for the behaviour of the various algorithms in this range of degrees.

References


Department of Computer Science, University of Western Ontario
E-mail address: jdoliska@uwo.ca

Department of Computer Science, University of Western Ontario
E-mail address: eschost@uwo.ca