Computing in Degree 2^k-Extensions of Finite Fields of Odd Characteristic

Javad Doliskani
jdoliska@uwo.ca

Éric Schost
eschost@uwo.ca

Abstract

We show how to perform basic operations (arithmetic, square roots, computing isomorphisms) over finite fields of the form $\mathbb{F}_{q^{2^k}}$ in essentially linear time.

Mathematics Subject Classification 2010. Primary 11Y16, 12Y05, Secondary 68W30.

1 Introduction

Factoring polynomials and constructing irreducible polynomials are two fundamental operations for finite field arithmetic. As of now, there exists no deterministic polynomial time algorithm for these questions in general, but in some cases better answers can be found. In this note, we discuss algorithmic questions arising in such a special case: the construction of, and computations with, extensions of degree $n = 2^k$ of a base field, say $\mathbb{F}_q$, with q a power of an odd prime p. In other words, we are interested with the complexity of computing in the quadratic closure of $\mathbb{F}_q$.

There exists a well-known construction of such extensions [13, Th. VI.9.1], which was already put to use algorithmically in [17]: if $q = 1 \bmod 4$, then for any quadratic non-residue $\alpha \in \mathbb{F}_q$, the polynomial $X^{2^k} - \alpha \in \mathbb{F}_q[X]$ is irreducible for any $k \geq 0$, and allows us to construct $\mathbb{F}_{q^{2^k}}$. If $q = 3 \bmod 4$, we first construct a degree-two extension $\mathbb{F}_{q'}$ of $\mathbb{F}_q$, which will thus satisfy $q' = 1 \bmod 4$; this is done by remarking that $X^2 + 1 \in \mathbb{F}_q[X]$ is irreducible, so that we can construct $\mathbb{F}_{q'}$ as $\mathbb{F}_q[X]/\langle X^2 + 1\rangle$.

In this note, taking this remark as a starting point, we give fast algorithms for operations such as multiplication and inversion, trace and norm computation, and most importantly square root computation in $\mathbb{F}_{q^{2^k}}$ (see below for our motivation), as well as isomorphism computation.

We do not make any assumption on the way $\mathbb{F}_q$ is represented: the running time of our algorithms is estimated by counting operations $(+, \times, \div)$ in $\mathbb{F}_q$ at unit cost. The cost of most algorithms will be expressed in terms of the cost of polynomial multiplication. Explicitly, we let $\mathsf{M} : \mathbb{N} \to \mathbb{N}$ be such that degree n polynomials over any ring R can be multiplied in $\mathsf{M}(n)$ operations in R, and such that $\mathsf{M}(n)/n$ is non-decreasing (this will be referred to as *super-linearity*). Using the algorithm of [3], we can take $\mathsf{M}(n) = O(n \log(n) \log\log(n))$.

In view of the discussion above, we will assume that $q = 1 \bmod 4$: if this is not the case, replacing $\mathbb{F}_q$ by $\mathbb{F}_{q'}$ as explained above only induces a constant overhead, since all operations

$(+, \times, \div)$ in $\mathbb{F}_{q'}$ can be done using $O(1)$ operations in $\mathbb{F}_q$. Besides, we will assume that the non-quadratic residue α is given; otherwise, such an α can be found by testing an expected $O(1)$ random elements in $\mathbb{F}_q$ for quadratic residuosity. This remains a core non-deterministic component of the construction, since finding α in a polynomial-time deterministic manner is a well-known open question. Some algorithms below are non-deterministic (Las Vegas) as well; for such algorithms, we give the expected running time.

Theorem 1. *Suppose that $q = 1 \bmod 4$. Given a non-quadratic residue $\alpha \in \mathbb{F}_q$, for $k \geq 0$ and $n = 2^k$, the running times for computations in $\mathbb{F}_{q^n}$ reported in Table 1 hold.*

Operation	Cost
addition / subtraction	$O(n)$
multiplication	$\mathsf{M}(n) + O(n)$
inversion	$O(\mathsf{M}(n))$
Frobenius	$O(n + \log(q))$
norm	$O(\mathsf{M}(n))$
trace	1
quadratic residuosity	$O(\mathsf{M}(n) + \log(q))$
square root	$O(\mathsf{M}(n) \log(nq))$ (expected)
isomorphism	$O(n + \log(n) \log(q))$ (expected)

Table 1: Costs for computations in $\mathbb{F}_{q^n}$, with $n = 2^k$.

This paper can be seen as an analogue of [5], which discusses these questions for Artin-Schreier extensions: the problems we consider, the techniques we use, and the applications (dealing with torsion points of some genus 1 or 2 Jacobians; see below) are similar. More precisely, the recursive techniques used here are similar to those in the reference in terms of exploiting the special structure of the tower and the defining polynomials.

For some questions, such as Frobenius or isomorphism computation, we refer to the next section for a precise description of the operation we perform; however, we mention that in all cases, we use a polynomial basis representation for all computations. In all cases, the combined size of input and output is $O(n)$ elements in $\mathbb{F}_q$, and using fast multiplication, all running times reported here are quasi-linear[1] in n.

Some results in the above table are straightforward (such as addition / subtraction or trace computation) and some are well-known (such as multiplication). Our focus in this note is actually on square root computation, and to the best of our knowledge, this is the first time that such results appear.

The straightforward approaches to square root extraction, using for instance Cipolla's or the Tonelli-Shanks algorithms [18, 4, 15] require a number of multiplications in $\mathbb{F}_{q^n}$ proportional to $\log(q^n)$; the cost is then $O(n\mathsf{M}(n)\log(q))$ operations in $\mathbb{F}_q$, which is at best quadratic in n. It is possible to compute square roots faster, using fast algorithms for *modular composition*, which is the operation that consists in computing $F(G) \bmod H$, given some

[1]An algorithm is quasi-linear time in n if it has complexity $O(n \log^k n)$ for a constant k.

univariate polynomials F, G, H. Let us denote by $\mathsf{C}(n)$ the cost of this operation, when F, G, H have all degree at most n. Then, the algorithms of [11, 6] compute square roots in $\mathbb{F}_{q^n}$ using $O(\mathsf{M}(n)\log(q) + \mathsf{C}(n)\log(n))$ operations in $\mathbb{F}_q$.

In our model, where operations in $\mathbb{F}_q$ are counted at unit cost, the best known bound on $\mathsf{C}(n)$ is $\mathsf{C}(n) = O(\sqrt{n}\mathsf{M}(n) + n^{(\omega+1)/2})$, where $2 \le \omega \le 3$ is such that matrices of size m over $\mathbb{F}_q$ can be multiplied in $O(m^\omega)$ operations [2]. Thus, depending on ω, and neglecting logarithmic factors, the cost (with respect to n) of the corresponding square root algorithms ranges between $O(n^{3/2})$ and $O(n^2)$. We remark however that, in a boolean model (on a RAM, using an explicit boolean representation of the elements in $\mathbb{F}_q$, and counting boolean operations at unit cost), Kedlaya and Umans gave in [12] an algorithm of cost $n^{1+\varepsilon}\log(q)^{1+o(1)}$ for modular composition in $\mathbb{F}_q$, for any $\varepsilon > 0$. In that model, algorithms such as those in [11, 6] are thus close to linear time.

None of the algorithms mentioned above is dedicated to the specific case we consider, where the extension degree n is a power of two; few references consider explicitly this particular case [7, 19].

The algorithm of [7] gives a quadratic residuosity test that uses $O(\log(n))$ Frobenius computations and multiplications in $\mathbb{F}_{q^n}$, and $O(\log(q))$ multiplications in $\mathbb{F}_q$. This reference does not specify how to represent the extensions of $\mathbb{F}_q$, and what algorithms should be used for basic operations. Using the algorithms given below for arithmetic and Frobenius computations, the cost of their quadratic residuosity test algorithm is $O((\mathsf{M}(n) + \log(q))\log(n))$ multiplications in $\mathbb{F}_q$; this is slightly slower than our result. The authors of [7] also give algorithms for quadratic residuosity and square root computation for degree 2^k-extensions in their later work [19]. They removed the computation overlap between quadratic residuosity test and square root, but the algorithms still run in times quadratic in n (the quadratic residuosity test being the bottleneck).

This work was inspired by computations with Jacobians of curves of genus 2 over finite fields. Given a genus 2 curve C defined over $\mathbb{F}_p$, the algorithm of [10] computes the cardinality of the Jacobian J of C, following Schoof's elliptic curve point counting algorithm [14]. This involves in particular the computation of successive divisors of 2^k-torsion in J, by means of successive divisions by two in J. Such a division by two boils down to several arithmetic operations, and four square root extractions; thus, the divisors we are computing are defined over the quadratic closure of $\mathbb{F}_p$, or of a small extension of $\mathbb{F}_p$.

In other words, for dealing with 2^k-torsion, the algorithm of [10] relies entirely on the operations above, arithmetic operations and square root extraction. At the time of writing [10], the authors relied on a variant of the Kaltofen-Shoup algorithm [11] with running time $O(\mathsf{M}(n)\log(q) + \mathsf{C}(n)\log(n))$, which was a severe bottleneck; our new algorithms completely alleviate this issue.

After proving Theorem 1 in Section 2, we present in Section 3 some experiments that confirm that the practical interest of our quasi-linear algorithms for the point-counting problem above.

Acknowledgements. The authors are supported by NSERC and the Canada Research Chairs program.

2 Proof of the complexity statements

In this section, we prove the results stated in Table 1. The tower of fields $\mathbb{F}_q, \mathbb{F}_{q^2}, \ldots \mathbb{F}_{q^{2^k}}, \ldots$ will be written

$$\mathbb{L}_0 \subset \mathbb{L}_1 \subset \cdots \subset \mathbb{L}_k \subset \cdots,$$

with $\mathbb{L}_k = \mathbb{F}_{q^{2^k}}$ for all k.

2.1 Representing the fields $\mathbb{L}_k$

The tower of fields $\mathbb{L}_0, \mathbb{L}_1, \ldots$ can be represented in two fashions, using univariate or multivariate representations (see as well [5], for a similar discussion for Artin-Schreier extensions).

Let $\alpha \in \mathbb{F}_q$ be a fixed non-quadratic residue. Define the sequence of polynomials $T_1, T_2, \ldots$ in indeterminates $X_1, X_2, \ldots$ given by

$$T_1 = X_1^2 - \alpha \quad \text{and} \quad T_k = X_k^2 - X_{k-1} \quad \text{for } k > 1,$$

as well as the polynomials $P_1, P_2, \ldots$ given by

$$P_k = X_k^{2^k} - \alpha \quad \text{for } k \geq 1.$$

The discussion in the introduction, as well as the one in [13], shows that for all $k \geq 0$ we have the equality between ideals

$$\langle T_1, \ldots, T_k \rangle = \langle P_k, X_{k-1} - X_k^2, \ldots, X_1 - X_k^{2^{k-1}} \rangle, \tag{1}$$

that this ideal (call it A_k) is prime, and that we have

$$\mathbb{L}_k \simeq \mathbb{F}_q[X_1, \ldots, X_k]/A_k.$$

For $k \geq 1$, let x_k be the image of X_k in the residue class ring $\mathbb{F}_q[X_1, \ldots, X_k]/A_k$. Due to the natural embedding of $\mathbb{F}_q[X_1, \ldots, X_k]/A_k$ into $\mathbb{F}_q[X_1, \ldots, X_{k+1}]/A_{k+1}$, x_k can be unequivocally seen as an element of $\mathbb{F}_{q^{2^\ell}}$ for $\ell \geq k$ (and thus as an element of the quadratic closure of $\mathbb{F}_q$).

On the left-hand side of (1), we have a Gröbner basis of A_k for the lexicographic order $X_1 < \cdots < X_k$, whereas on the right we have a Gröbner basis for the lexicographic order $X_k < \cdots < X_1$. Corresponding to these two bases of A_k, the elements of $\mathbb{L}_k$ can be represented as polynomials in $x_1, \ldots, x_k$ of degree at most 1 in each variable (and coefficients in $\mathbb{F}_q$), or as polynomials in x_k of degree less than 2^k.

By default, we will use the univariate representation; an element γ of $\mathbb{L}_k$ will thus be written as $\gamma = G(x_k)$, for some polynomial $G \in \mathbb{F}_q[X_k]$ of degree less than 2^k.

We will not explicitly need to convert to multivariate polynomials, but we will often do one step of such a conversion, switching between univariate and bivariate bases. Indeed, for all $k \geq 1$, we have

$$\mathbb{L}_k \simeq \mathbb{F}_q[X_k]/\langle P_k(X_k) \rangle \simeq \mathbb{F}_q[X_{k-1}, X_k]/\langle P_{k-1}(X_{k-1}), X_k^2 - X_{k-1} \rangle.$$

The $\mathbb{F}_q$-monomial basis of $\mathbb{F}_{q^{2^k}}$ associated to the left-hand side is

$$1,\ x_k,\ \ldots,\ x_k^{2^k-1},$$

whereas the one associated to the right-hand side is

$$1,\ x_{k-1},\ \dots,\ x_{k-1}^{2^{k-1}-1},\ x_k,\ x_{k-1}x_k,\ \dots,\ x_{k-1}^{2^{k-1}-1}x_k.$$

The change-of-basis from the univariate basis to the bivariate one amounts to writing an expression $G(x_k)$, with G of degree less than 2^k, as

$$G(x_k) = G_0(x_k^2) + x_k G_1(x_k^2) = G_0(x_{k-1}) + x_k G_1(x_{k-1}),$$

with G_0 and G_1 of degrees less than 2^{k-1}. This does not require any arithmetic operation; the same holds for the converse change-of-basis. Continuing this way on G_0 and G_1, we could convert between univariate and multivariate bases without arithmetic operations if needed.

2.2 Arithmetic operations

In this subsection, we discuss the cost of arithmetic operations $(+, \times, \div)$ in $\mathbb{L}_k$, for some $k \geq 0$. In all that follows, we write $n = [\mathbb{L}_k : \mathbb{F}_q]$, that is, $n = 2^k$.

Addition and subtraction take time $O(n)$. For multiplication, using the univariate basis leads to a cost of $\mathsf{M}(n) + O(n)$ operations in $\mathbb{F}_q$, where the $O(n)$ term accounts for reduction modulo P_k (this takes linear time, since P_k is a binomial). Note that a non-trivial (and slightly less efficient) approach using the multivariate representation is in [1].

For inversion, in the univariate basis, a natural idea is to use the fast extended GCD algorithm for univariate polynomials [8, Ch. 11], resulting in a running time $O(\mathsf{M}(n)\log(n))$. However, better can be done by using the tower structure of the fields $\mathbb{L}_k$. Indeed, consider $\gamma = G(x_k) \in \mathbb{L}_k^\times$. Then, writing $G(x_k) = G_0(x_{k-1}) + x_k G_1(x_{k-1})$, we have

$$\begin{aligned}\frac{1}{G(x_k)} &= \frac{1}{G_0(x_{k-1}) + x_k G_1(x_{k-1})}\\ &= \frac{G_0(x_{k-1}) - x_k G_1(x_{k-1})}{G_0(x_{k-1})^2 - x_{k-1} G_1(x_{k-1})^2}.\end{aligned}$$

Therefore, computing an inverse in $\mathbb{L}_k$ amounts to $O(1)$ additions and multiplications in $\mathbb{L}_k$, and one inversion in $\mathbb{L}_{k-1}$. This gives a recursive algorithm with cost $T(k) = T(k-1) + O(\mathsf{M}(2^k))$; the super-linearity of M implies that the running time is $O(\mathsf{M}(2^k)) = O(\mathsf{M}(n))$ operations in $\mathbb{F}_q$.

2.3 Frobenius computation

Let γ be in $\mathbb{L}_k$, and let $r = q^d$ for some positive integer d. We explain here how to compute $\gamma^r \in \mathbb{L}_k$; as above, we write $n = 2^k$.

We start with the case $\gamma = x_k$: writing $r = nu + v$ with $0 \leq v < n$, and using the fact that $x_k^n = \alpha$, we obtain $x_k^r = \alpha^u x_k^v$. The constant $\alpha^u = \alpha^{u \bmod (q-1)}$ can be computed in $O(\log(q))$ multiplications in $\mathbb{F}_q$ by repeated squaring. Although our focus is on counting $\mathbb{F}_q$-operations, we also mention how to compute $u \bmod (q-1)$ and v efficiently: first of all, we compute $\rho = r \bmod n(q-1)$ by repeated squaring; then, $u \bmod (q-1)$ and v are respectively the quotient and remainder in the division of ρ by n. They can be computed with a boolean cost polynomial (and actually, quasi-linear) in $\log(d)\log(nq)$, since the bottleneck is an exponentation with exponent $O(d\log(q))$, modulo an integer of bit size $O(\log(nq))$.

For a general γ of the form $\gamma = G(x_k)$, writing $G(x_k) = g_{n-1}x_k^{n-1} + \cdots + g_1x_k + g_0$, we have

$$\begin{aligned}\gamma^r &= g_{n-1}(x_k^r)^{n-1} + \cdots + g_1(x_k^r) + g_0\\ &= g_{n-1}(\alpha^u x_k^v)^{n-1} + \cdots + g_1(\alpha^u x_k^v) + g_0.\end{aligned}$$

Knowing α^u and v, computing γ^r amounts to compute the first n powers of $\alpha^u x_k^v$ and substituting them in G. Since $x_k^n = \alpha$, these powers are all monomials in x_k, and can be computed successively in $O(n)$ multiplications in $\mathbb{F}_q$. Therefore, computing the Frobenius takes a total of $O(n + \log(q))$ operations in $\mathbb{F}_q$.

2.4 Trace, norm and quadratic residuosity test

The norm $N_{\mathbb{L}_k/\mathbb{F}_q}$ and the trace $T_{\mathbb{L}_k/\mathbb{F}_q}$ are easy to compute, using transitivity. Indeed, for $\gamma \in \mathbb{L}_k$, we have

$$N_{\mathbb{L}_k/\mathbb{F}_q}(\gamma) = N_{\mathbb{L}_1/\mathbb{F}_q}(N_{\mathbb{L}_2/\mathbb{L}_1}(\cdots N_{\mathbb{L}_k/\mathbb{L}_{k-1}}(\gamma)))$$

and

$$T_{\mathbb{L}_k/\mathbb{F}_q}(\gamma) = T_{\mathbb{L}_1/\mathbb{F}_q}(T_{\mathbb{L}_2/\mathbb{L}_1}(\cdots T_{\mathbb{L}_k/\mathbb{L}_{k-1}}(\gamma))).$$

Write as before $\gamma = G(x_k)$ and $G = G_0(x_{k-1}) + x_kG_1(x_{k-1})$. Then, we have

$$N_{\mathbb{L}_k/\mathbb{L}_{k-1}}(\gamma) = G_0(x_{k-1})^2 - x_{k-1}G_1(x_{k-1})^2$$

and

$$T_{\mathbb{L}_k/\mathbb{L}_{k-1}}(\gamma) = 2G_0(x_{k-1}),$$

since in the quadratic extension $\mathbb{L}_k$ of $\mathbb{L}_{k-1}$ generated by $x_k^2 - x_{k-1}$, the norm (resp. trace) of γ is the product (resp. sum) of γ and its conjugate $\gamma' = G_0(x_{k-1}) - x_kG_1(x_{k-1})$.

To compute the norm of γ, this gives a recursive algorithm using one recursive call and $O(1)$ multiplications in each extension; using the super-linearity of M (as for inversion), the total is $O(\mathsf{M}(n))$ operations in $\mathbb{F}_q$. For the trace, by transitivity, we obtain $T_{\mathbb{L}_k/\mathbb{F}_q}(\gamma) = ng_0$ where g_0 is the constant term of G; this could also have been deduced from the fact that the trace of $\gamma = G(x_k)$ in the extension $\mathbb{L}_k = \mathbb{F}_q[X_k]/\langle P_k\rangle$ is the coefficient of X_k^{n-1} in GP_k' mod P_k. At any rate, the trace is computed using 1 multiplication in $\mathbb{F}_q$

Finally, to check if γ is a quadratic residue we compute $\gamma^{(q^n-1)/2} = N_{\mathbb{L}_k/\mathbb{F}_q}(\gamma)^{(q-1)/2}$; this takes $O(\mathsf{M}(n) + \log(q))$ operations in $\mathbb{F}_q$ (using repeated squaring for the exponentiation).

Let us briefly comment on alternative derivations of the norm and the trace; they are slightly less efficient, but these ideas will allow us to compute square roots in the next subsection (the underlying idea is not new; it appears for instance in [9, 7]). Fix n and γ as above and for $m \geq 0$, define

$$N_m(\gamma) = \gamma^{1+q+q^2+\cdots+q^{m-1}}$$

to be the *m-norm* of γ. Similarly, the *m-trace* of γ is defined to be $T_m(\gamma) = \gamma + \gamma^q + \cdots + \gamma^{q^{m-1}}$ (this is called a pseudo-trace in [5]). For $m = n$, we recover the standard norm and trace.

Writing

$$\zeta_m = \gamma^{q+q^2+\cdots+q^m},$$

we have $N_m(\gamma) = \gamma\zeta_{m-1}$. The element ζ_m can be computed efficiently by means of the formulas

$$\zeta_1 = \gamma^q \quad \text{and} \quad \zeta_m = \begin{cases} \zeta_{m/2}\zeta_{m/2}^{q^{m/2}} & \text{if } m \text{ is even} \\ \zeta_1\zeta_{m-1}^q & \text{if } m \text{ is odd.} \end{cases} \tag{2}$$

(We could compute $N_m(\gamma)$ directly using a similar recurrence, but we will reuse the ζ_m in the next subsection.) Computing ζ_1 is done by means of one Frobenius computation. Deducing ζ_m from either $\zeta_{m/2}$ or $\zeta_{(m-1)/2}$ takes $O(1)$ Frobenius and multiplications. Thus, the total for ζ_m, and thus $N_m(\gamma)$, is $O(\log(m))$ Frobenius and multiplications in $\mathbb{L}_k$, which amounts to $O(\mathsf{M}(n)\log(m) + \log(q)\log(m))$ operations in $\mathbb{F}_q$.

If needed, the m-trace can be computed similarly to the m-norm by the following recursion:

$$T_1 = \gamma \quad \text{and} \quad T_m = \begin{cases} T_{m/2} + T_{m/2}^{q^{m/2}} & \text{if } m \text{ is even} \\ T_1 + T_{m-1}^q & \text{if } m \text{ is odd.} \end{cases}$$

Therefore, $T_m(\gamma)$ can be computed using $O(\log(m))$ Frobenius and additions, hence the overall running time $O(n\log(m) + \log(q)\log(m))$.

2.5 Taking square roots

In this subsection, we review the idea presented in [6] to compute square roots, and adapt it to our situation, where computing a Frobenius is cheap.

Let $\delta \in \mathbb{L}_k^\times$ be given, assume that δ is a square and let $\gamma \in \mathbb{L}_k$ be an (unknown) square root of it. Define $\beta \in \mathbb{F}_q$ as the (unknown) quantity

$$\begin{aligned} \beta = T_{\mathbb{L}_k/\mathbb{F}_q}(\gamma) = \sum_{i=0}^{n-1} \gamma^{q^i} &= \gamma(1 + \gamma^{q-1} + \gamma^{q^2-1} + \cdots + \gamma^{q^{n-1}-1}) \\ &= \gamma(1 + \delta^{(q-1)/2} + \delta^{(q^2-1)/2} + \cdots + \delta^{(q^{n-1}-1)/2}) \\ &= \gamma\eta, \end{aligned} \tag{3}$$

with

$$\eta = 1 + \delta^{(q-1)/2} + \delta^{(q^2-1)/2} + \cdots + \delta^{(q^{n-1}-1)/2}.$$

We may assume $\eta \neq 0$; otherwise, we can replace δ by $\delta' = \delta c^2$ for a random element $c \in \mathbb{L}_k^\times$. We expect to have $T_{\mathbb{F}_q/\mathbb{F}_{q'}}(\gamma c) \neq 0$ after $O(1)$ trials: There are q^{2^k}/q values of c for which the trace is zero, and hence the probability of having a non-zero trace is $1 - (q^{2^k}/q)/q^{2^k} = 1 - 1/q \geq 1/2$. Squaring both sides of Eq. (3) results in the quadratic equation $\beta^2 = \delta\eta^2$ over $\mathbb{F}_q$.

Provided η is known, β can be deduced from this quadratic equation, and finally γ as $\gamma = \beta\eta^{-1}$. Computing β from the above quadratic equation takes an expected $O(\log(q))$ operations in $\mathbb{F}_q$ [8, Ch. 14.5], so that computing η is the key to computing γ efficiently. This can be done as follows.

Let $\lambda \in \mathbb{L}_k$ be defined by $\lambda = \delta^{(q-1)/2}$; then, η is given by

$$\eta = 1 + \lambda + \lambda^{1+q} + \lambda^{1+q+q^2} + \cdots + \lambda^{1+q+q^2+\cdots+q^{n-2}}.$$

For $m \geq 0$, define

$$\varepsilon_m = \lambda^q + \lambda^{q+q^2} + \cdots + \lambda^{q+q^2+\cdots+q^m},$$

so that $\eta = 1+\lambda+\lambda\varepsilon_{n-2}$, and recall as well the definition of $\zeta_m = \lambda^{q+q^2+\cdots+q^m}$. Then, similar to the recurrence relation given in (2) for ζ, the following holds for ε:

$$\varepsilon_1 = \lambda^q \quad \text{and} \quad \varepsilon_m = \begin{cases} \varepsilon_{m/2} + \zeta_{m/2}\varepsilon_{m/2}^{q^{m/2}} & \text{if } m \text{ is even} \\ \varepsilon_{m-1} + \zeta_m & \text{if } m \text{ is odd} \end{cases} \tag{4}$$

Computing λ takes $O(\mathsf{M}(n)\log(q))$ operations in $\mathbb{F}_q$; then, we obtain the initial values ζ_1 and ε_1 using $O(1)$ Frobenius. Assume, inductively, that we have computed ε_m, ζ_m. Then, using Eqs. (2) and (4), ζ_{2m} and ε_{2m}, or ζ_{2m+1} and ε_{2m+1}, can be computed using $O(1)$ Frobenius and $O(1)$ multiplications in $\mathbb{F}_{q^n}$. Therefore, ε_n, and altogether $\eta = 1+\lambda+\lambda\varepsilon_{n-2}$ can be computed using $O(\mathsf{M}(n)\log(q) + \mathsf{M}(n)\log(n)) = O(\mathsf{M}(n)\log(nq))$ operations in $\mathbb{F}_q$. We have seen that deducing β takes an expected $O(\log(q))$ operations in $\mathbb{F}_q$, and the cost of computing γ is negligible compared to the computation of η. Thus, the overall running time is an expected $O(\mathsf{M}(n)\log(nq))$ operations in $\mathbb{F}_q$.

2.6 Computing embeddings

We finally consider the problem of computing embeddings and isomorphisms between two different "towers" defining the quadratic closure of $\mathbb{F}_q$. Consider $\mathbb{L}_k = \mathbb{F}_q[X_k]/\langle P_k\rangle$ and $\mathbb{L}'_j = \mathbb{F}_q[Y_j]/\langle Q_j\rangle$, k, j positive integers, where we write

$$P_k = X_k^{2^k} - \alpha \quad \text{and} \quad Q_j = Y_j^{2^j} - \beta,$$

for some non-quadratic residues α, β in $\mathbb{F}_q$. Assuming for instance that $k \leq j$, $\mathbb{L}_k$ can be identified as a subfield of $\mathbb{L}'_j$; we show how to compute an embedding $\phi : \mathbb{L}_k \hookrightarrow \mathbb{L}'_j$ efficiently. As before, we denote by x_k the image of X_k in $\mathbb{L}_k$, and by y_j the image of Y_j in $\mathbb{L}'_j$; we write $n = 2^k$ and $m = 2^j$.

The idea is straightforward: we first find a root ρ of P_k in $\mathbb{L}'_j$; then, the mapping $\phi : \mathbb{L}_k \to \mathbb{L}'_j$ given by $\phi(G(x_k)) = G(\rho) \bmod Q_j$ is well-defined and gives an isomorphism of $\mathbb{L}_k$ onto its image. To find the root ρ of P_k in $\mathbb{L}'_j$, one has to take k successive square roots of α in $\mathbb{L}'_j$. For this, we could use the algorithm of the previous subsection, but since we start from $\alpha \in \mathbb{F}_q$, better can be done.

Let us look at a slightly more general question: given $\mu \in \mathbb{F}_q$ and an even integer $\ell \geq 0$, compute a square root of μy_j^ℓ:

- if μ is a square in $\mathbb{F}_q$, say $\mu = \nu^2$, $\nu y_j^{\ell/2}$ is a square root of μy_j^ℓ;
- else, μ/β is a square in $\mathbb{F}_q$, say $\mu/\beta = \nu^2$; then $\nu y_j^{2^{j-1}+\ell/2}$ is a square root of μy_j^ℓ.

Since we start with $\ell = 0$, we can repeat the process at least j times, and thus in particular at least k times; the cost is thus that of $O(k) = O(\log(n))$ quadratic residuosity tests, square-root computations and arithmetic operations in $\mathbb{F}_q$; this uses an expected $O(\log(q)\log(n))$ operations in $\mathbb{F}_q$. Since the root ρ we obtain is of the form $\rho = \mu y_j^\ell$ for some $\ell < m$, computing $G(\rho) \mod Q_j$, for some G of degree less than n (and thus than m), takes $O(m)$ multiplications in $\mathbb{F}_q$ (as was the case for Frobenius computation).

For isomorphism computation, taking $k = j$, and thus $m = n$, gives the claimed bound $O(n + \log(q)\log(n))$ operations in $\mathbb{F}_q$.

3 Experiments

We conclude this section with experiments using an implementation of our algorithms based on NTL [16]. All running times are obtained on an Intel Xeon CPU. In all cases, we start from the base field $\mathbb{F}_p$.

First, we consider square root computation. For computing square roots in an extension $\mathbb{F}_{p^n}$, without assumption on n, we used in [6] modular polynomial composition, resulting in the running time $O(\mathsf{M}(n)\log(p) + \mathsf{C}(n)\log(n))$. In cases where n is a power of two, the results in this paper are superior in terms of complexity; Figure 1 confirms that this is also the case in practice. In that figure, we take the "random" prime $p = 348975609381470925634534573457497$ already used in [6], and different values of the extension degree n, that are always powers of two; see below for the reasons behind our choice of such a large value of p.

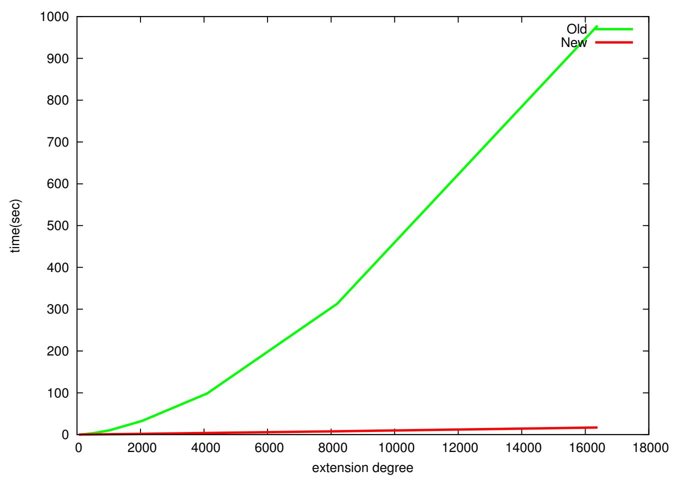

Figure 1: The new square root algorithm vs. the one in [6]

Table 2 gives timings (in seconds) for the genus 2 point-counting application described in the introduction. The table describes the various ingredients involved in computing successive 2^k-torsion divisors in the Jacobian of the (randomly chosen) curve C:

$$\begin{aligned} y^2 \;=\; & x^5 + 67412365472663169119085380769732137727\, x^4 \\ & + 132706051439871719391705031627238584248\, x^3 \\ & + 150906984006321211274278480789580538770\, x^2 \\ & + 5602222826077482782805347307759926224\, x \\ & + 157456212652423046465778673243920804193. \end{aligned}$$

over $\mathbb{F}_p$ with $p = 2^{127} - 1$ (this 127 bit prime is the one used in the records described in [10]). Once these divisors are known, they are used to compute the cardinality of the Jacobian J of C modulo 2^k (more exactly, we compute the image modulo 2^k of the characteristic polynomial of the Frobenius endomorphism on J); this last step is not detailed here, we will refer to it as the *search* step.

In Table 2, the degree e_k is the degree of the field extension over which the successive divisors are defined. There are two main rows: the first one gives the timings for computing all required square roots (which give us the required 2^k-torsion divisors); the second row gives the timing for the search step, which involves arithmetic operations in the current extension of $\mathbb{F}_p$. For a more precise profiling, each of the two main rows is divided into three subrows labelled with I, II, and III: I denotes the original Gaudry and Schost implementation [10]; II and III denote the same implementation but using the algorithm in [6] and the one in Section 2.5 respectively. All square root algorithms in this table are probabilistic.

In previous implementations, square root computation was a clear bottleneck; with our new algorithm, it has now become a minor component of the running time.

index 2^k		2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}
degree e_k		2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
square roots	I	0.2	0.4	1.2	3.5	11	33	109	365	1262	4466	16246	60689
	II	0.2	0.5	1.2	2.9	8	23	73	232	734	2309	7368	23604
	III	0.1	0.2	0.5	1.1	2	5	11	25	53	114	246	523
search step	I	0.5	1.1	2.8	6.5	14	32	73	164	368	816	2020	4827
	II	0.4	1.0	2.3	5.4	12	27	62	139	309	657	1609	3740
	III	0.4	0.9	2.0	4.5	11	24	53	119	267	598	1402	3297

Table 2: Timings for lifting 2^k-torsion

References

[1] A. Bostan, M. F. I. Chowdhury, J. van der Hoeven, and É. Schost. Homotopy methods for multiplication modulo triangular sets. *Journal of Symbolic Computation*, 46(12):1378–1402, 2011.

[2] R. P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. *Journal of the Association for Computing Machinery*, 25(4):581–595, 1978.

[3] D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28(7):693–701, 1991.

[4] M. Cipolla. Un metodo per la risoluzione della congruenza di secondo grado. *Napoli Rend.*, 9:153–163, 1903.

[5] L. De Feo and É. Schost. Fast arithmetics in Artin-Schreier towers over finite fields. *Journal of Symbolic Computation*, 47(7):771–792, 2012.

[6] J. Doliskani and É. Schost. Taking roots over high extensions of finite fields. *Mathematics of Computation*, 2012. To appear.

[7] W. Feng, Y. Nogami, and Y. Morikawa. A fast square root computation using the Frobenius mapping. In *Information and Communications Security*, volume 2836 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2003.

[8] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, second edition, 2003.

[9] J. von zur Gathen and V. Shoup. Computing Frobenius maps and factoring polynomials. *Comput. Complexity*, 2(3):187–224, 1992.

[10] P. Gaudry and É. Schost. Genus 2 point counting over prime fields. *Journal of Symbolic Computation*, 47(4):368 – 400, 2012.

[11] E. Kaltofen and V. Shoup. Fast polynomial factorization over high algebraic extensions of finite fields. In *ISSAC'97*, pages 184–188. ACM, 1997.

[12] K. S. Kedlaya and C. Umans. Fast polynomial factorization and modular composition. *SIAM J. Computing*, 40(6):1767–1802, 2011.

[13] S. Lang. *Algebra*, volume 211 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, third edition, 2002.

[14] R. Schoof. Elliptic curves over finite fields and the computation of square roots mod p. *Mathematics of Computation*, 44:483–494, 1985.

[15] D. Shanks. Five number-theoretic algorithms. In *Proceedings of the Second Manitoba Conference on Numerical Mathematics*, pages 51–70, 1972.

[16] V. Shoup. A library for doing number theory (NTL). `http://www.shoup.net/ntl/`.

[17] V. Shoup. Fast construction of irreducible polynomials over finite fields. *Journal of Symbolic Computation*, 17(5):371–391, 1994.

[18] A. Tonelli. Bemerkung über die Auflösung quadratischer Congruenzen. *Göttinger Nachrichten*, pages 344–346, 1891.

[19] Feng Wang, Yasuyuki Nogami, and Yoshitaka Morikawa. An efficient square root computation in finite fields $GF(p^{2^d})$. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E88-A(10):2792–2799, October 2005.