Efficient Multiplication in Finite Field Extensions of Degree 5

Nadia El Mrabet[1], Aurore Guillevic[2,3], and Sorina Ionica[4]

[1] LIASD - Université Paris 8, France
nelmrabe@mime.univ-paris8.fr
[2] Laboratoire Chiffre, Thales Communications S.A.,
160 bd de Valmy BP 82, 92704 Colombes Cedex France
[3] Équipe crypto DI/LIENS, École Normale Supérieure, France
aurore.guillevic@fr.thalesgroup.com
[4] TANC, Inria Saclay and LIX, École Polytechnique, France
sorina.ionica@m4x.org

Abstract. Small degree extensions of finite fields are commonly used for cryptographic purposes. For extension fields of degree 2 and 3, the Karatsuba and Toom Cook formulæ perform a multiplication in the extension field using 3 and 5 multiplications in the base field, respectively. For degree 5 extensions, Montgomery has given a method to multiply two elements in the extension field with 13 base field multiplications. We propose a faster algorithm, which requires only 9 base field multiplications. Our method, based on Newton's interpolation, uses a larger number of additions than Montgomery's one but our implementation of the two methods shows that for cryptographic sizes, our algorithm is much faster.

Keywords: finite field arithmetic, implementation, interpolation

1 Introduction

Efficient implementation of a cryptosystem often relies on high performance arithmetic in a finite field $\mathbb{F}_q$ or over some extension field of small degree. In view of recent proposals for torus-based cryptography [17] and pairing-based cryptography [7], we propose a method to implement the arithmetic of the finite field $\mathbb{F}_{q^5}$, if the characteristic of $\mathbb{F}_q$ is greater than 11. Our method, based on Newton's interpolation method for multiplying two polynomials, is faster than previously known methods to perform multiplication in such fields.

We begin by enumerating several uses of degree five extension fields in cryptography. Firstly, Rubin and Silverberg [16] considered the problem of compression (i.e. representing elements in a finite field subgroup with fewer bits than classical algorithms) for extension fields in terms of *algebraic tori* $T_n(\mathbb{F}_q)$ (i.e. the elements of $\mathbb{F}_{q^n}^*$ whose norm is one down to every proper subfield of $\mathbb{F}_{q^n}/\mathbb{F}_q$). Rubin and Silverberg developed the CEILIDH cryptosystem based on $T_6(\mathbb{F}_q)$. In [17], van Dijk et al. gave applications based on $T_{30}(\mathbb{F}_q)$, such as El Gamal

encryption, El Gamal signatures and voting schemes. Van Dijk et al. proposed an implementation of $T_{30}(\mathbb{F}_q)$, based on some techniques used to implement CEILIDH. More precisely, they implemented $\mathbb{F}_{q^5}$ by using a degree 5 subfield of the degree 10 extension $\mathbb{F}_q[X]/\Phi_{11}(X)$, with $\Phi_{11}(X)$ the eleventh cyclotomic polynomial. Their operation count shows that multiplication in $\mathbb{F}_{q^5}$ requires $15M_q + 75A_q$, where M_q and A_q denote the costs of multiplication and addition in $\mathbb{F}_q$.

Secondly, we note that several families of elliptic curves having embedding degree 10, 15, 30 or 35 have been proposed for pairing-based cryptography [7,8]. These families are recommended for implementations at high security levels, i.e. 192 and 256. On such curves, we need an efficient implementation of an extension field whose degree is divisible by 5. This is generally done by using tower fields and thus requires an efficient arithmetic of $\mathbb{F}_{q^5}$. Moreover, note that the pairing values can be represented in compressed form by using algebraic tori. Up to the present, such implementations were done for supersingular curves in characteristic 3 (see [9]) and for Barreto-Naehrig curves (see [15]). The arithmetic of $T_{30}(\mathbb{F}_q)$ may be used for compressible pairings on curves with embedding degree 30, for example.

Bodrato [4] proposed a method to speed up the Toom Cook algorithm for degree 5 extension fields with characteristic 2. In [2], the multiplication in $\mathbb{F}_{q^5}$ is computed with two applications of the Karatsuba method and requires $14M_q + 34A_q$. To the best of our knowledge, the fastest known formula for computing multiplication over $\mathbb{F}_{q^5}$ with $\mathrm{char}(\mathbb{F}_q) > 5$ can be derived from Montgomery's method to multiply two five-term polynomials [14]. The complexity of his method is $13M_q + 62A_q$.

We propose a method to perform multiplication over $\mathbb{F}_{q^5}$ which relies on Newton's interpolation method. Interpolation methods require performing a certain number of divisions. Divisions are generally expensive, but we show that with Newton's interpolation, it is possible to choose the interpolation values such that we only need to perform a small number of divisions by small constants. Our operation count gives a total cost of $9M_q + 137A_q$ for a multiplication in $\mathbb{F}_{q^5}$. In order to apply Montgomery's method to multiplication in $\mathbb{F}_{q^5}$, some extra additions are needed. Even though our algorithm performs a great number of additions, our method is faster than Montgomery's one if $M_q > 18A_q$. Our method can be adapted for degree 6 and 7 extension fields. Our operation count shows that we need $11M_q + 196A_q$ for a multiplication in $\mathbb{F}_{q^6}$ and $13M_q + 271A_q$ for a multiplication in $\mathbb{F}_{q^7}$.

This paper is organized as follows: in Sect. 2 we describe Montgomery's method and estimate the number of additions in $\mathbb{F}_q$ that this method performs. In Sect. 3 we describe an efficient multiplication based on the interpolation for the field $\mathbb{F}_{p^5}$. Section 4 describes our implementation and gives experimental results. In Sect. 5 we show that our idea can be used to optimize the arithmetic of degree 6 and 7 extension fields. Finally, Sect. 6 shows that our method applies for implementations in pairing-based and torus-based cryptography, for high levels of security. In Appendix 8 we display the complete formula for inversion in $\mathbb{F}_{q^5}$.

2 Montgomery's approach

Let $\mathbb{F}_q$ be a finite field of characteristic greater than 5. Usually, an extension of degree k of $\mathbb{F}_q$ is defined by $\mathbb{F}_{q^k} = \mathbb{F}_q[X]/(P(X)\mathbb{F}_q[X])$ where $P(X) \in \mathbb{F}_q[X]$ is an irreducible polynomial of degree k. Consequently, elements of $\mathbb{F}_{q^k}$ are represented by polynomials in X, of degree at most $k-1$ and with coefficients in $\mathbb{F}_q$. Montgomery [14] proposed a Karatsuba-like formula for 5-term polynomials. We recall here his method. Let $A = a_0 + a_1X + a_2X^2 + a_3X^3 + a_4X^4$ and $B = b_0 + b_1X + b_2X^2 + b_3X^3 + b_4X^4$ in $\mathbb{F}_{q^5}$ with coefficients over $\mathbb{F}_q$.

Montgomery constructs the polynomial $C(X) = A(X) \cdot B(X)$ using the following formula

$$
\begin{aligned}
C &= (a_0 + a_1X + a_2X^2 + a_3X^3 + a_4X^4)(b_0 + b_1X + b_2X^2 + b_3X^3 + b_4X^4) \\
&= (a_0 + a_1 + a_2 + a_3 + a_4)(b_0 + b_1 + b_2 + b_3 + b_4)(X^5 - X^4 + X^3) \\
&\quad +(a_0 - a_2 - a_3 - a_4)(b_0 - b_2 - b_3 - b_4)(X^6 - 2X^5 + 2X^4 - X^3) \\
&\quad +(a_0 + a_1 + a_2 - a_4)(b_0 + b_1 + b_2 - b_4)(-X^5 + 2X^4 - 2X^3 + X^2) \\
&\quad +(a_0 + a_1 - a_3 - a_4)(b_0 + b_1 - b_3 - b_4)(X^5 - 2X^4 + X^3) \\
&\quad +(a_0 - a_2 - a_3)(b_0 - b_2 - b_3)(-X^6 + 2X^5 - X^4) \\
&\quad +(a_1 + a_2 - a_4)(b_1 + b_2 - b_4)(-X^4 + 2X^3 - X^2) \\
&\quad +(a_3 + a_4)(b_3 + b_4)(X^7 - X^6 + X^4 - X^3) \\
&\quad +(a_0 + a_1)(b_0 + b_1)(-X^5 + X^4 - X^2 + X) \\
&\quad +(a_0 - a_4)(b_0 - b_4)(-X^6 + 3X^5 - 4X^4 + 3X^3 - X^2) \\
&\quad +a_4b_4(X^8 - X^7 + X^6 - 2X^5 + 3X^4 - 3X^3 + X^2) \\
&\quad +a_3b_3(-X^7 + 2X^6 - 2X^5 + X^4) \\
&\quad +a_1b_1(X^4 - 2X^3 + 2X^2 - X) \\
&\quad +a_0b_0(X^6 - 3X^5 + 3X^4 - 2X^3 + X^2 - X + 1)
\end{aligned}
$$

The cost of these computations is $13M_q + 22A_q$. Note that in order to recover the final expression of the polynomial of degree 8, we have to re-organize the 13 lines to find its coefficients. We denote the products on each of the 13 lines by u_i, $0 \leq i \leq 12$ (i.e. $u_{12} = (a_0 + a_1 + a_2 + a_3 + a_4)(b_0 + b_1 + b_2 + b_3 + b_4)$, $u_{11} = (a_0 - a_2 - a_3 - a_4)(b_0 - b_2 - b_3 - b_4)$ etc.) By re-arranging the formula in function of the degree of X, we obtain the following expression for C

$$
\begin{aligned}
C &= u_3X^8 \\
&+ (-u_2 - u_3 + u_6)X^7 \\
&+ (u_0 + 2u_2 + u_3 - u_4 - u_6 - u_8 + u_{11})X^6 \\
&+ (-3u_0 - 2u_2 - 2u_3 + 3u_4 - u_5 + 2u_8 + u_9 - u_{10} - 2u_{11} + u_{12})X^5 \\
&+ (3u_0 + u_1 + u_2 + 3u_3 - 4u_4 + u_5 + u_6 - u_7 - u_8 - 2u_9 + 2u_{10} + 2u_{11} - u_{12})X^4 \\
&+ (-2u_0 - 2u_1 - 3u_3 + 3u_4 - u_6 + 2u_7 + u_9 - 2u_{10} - u_{11} + u_{12})X^3 \\
&+ (u_0 + 2u_1 + u_3 - u_4 - u_5 - u_7 + u_{10})X^2 \\
&+ (-u_0 - u_1 + u_5)X \\
&+ u_0
\end{aligned}
$$

Considering this expression, we can easily count hidden additions in Montgomery's formula. We have taken into account that some operations are repetitive and simplified the expression of C very carefully by hand. We obtain the following formula

$$
\begin{aligned}
C &= u_3X^8 \\
&+ (-u_2 + u_6 - u_3)X^7 \\
&+ ((u_0 + u_3 - u_4) - (u_6 - u_2) + (u_2 - u_8 + u_{11}))X^6 \\
&+ (u_3 - u_5 + u_9 - u_{10} + u_{12} - 2(u_2 - u_8 + u_{11}) - 3(u_0 + u_3 - u_4))X^5 \\
&+ (u_1 + u_2 - u_4 + u_5 + u_6 - u_7 - u_8 - u_{12} - 2(u_9 - u_{10} - u_{11}) + 3(u_0 + u_3 - u_4))X^4 \\
&+ (u_0 - u_6 + u_9 - u_{11} + u_{12} - 2(u_1 - u_7 + u_{10}) - 3(u_0 + u_3 - u_4))X^3 \\
&+ ((u_0 + u_3 - u_4) + u_1 - u_5 + (u_1 - u_7 + u_{10}))X^2 \\
&+ (-u_0 - u_1 + u_5)X \\
&+ u_0
\end{aligned}
$$

We consider that a multiplication by 3 costs one addition. This is due to the fact that $3U = 2U + U$ and that the product by 2 is only a shift in the binary decomposition (see Sect. 4.2). Our operation count shows that we need to perform 42 extra additions in order to get C. To sum up, Montgomery's method costs $13M_q + 62A_q$. Finally, in order to compute $C \mod P(X)$, we need some extra operations. Since the reduction technique is similar to the one for the multiplication method we propose, we detail it in Sect. 3.

3 Our approach

In extensions of degree 2 and 3 Karatsuba and Toom Cook multiplications are the most efficient known. For composite degree extensions (i.e. 2^i3^j, for $i, j > 0$) one can use tower field extensions [11] and apply Karatsuba and Toom Cook methods [18]. If the degree of the extension is not composite, one may use the FFT method [18].

In this paper, we are interested in efficiently computing multiplications in extension fields of degree 5. Note that the use of FFT is not interesting in this case. Indeed, during a FFT multiplication, we have to multiply by roots of unity. In the general case, q is a large random prime number and the roots of unity over $\mathbb{F}_q$ do not necessarily have a sparse representation, even after recoding. Hence multiplications by these roots are expensive.

Finally, we may use Lagrange or Newton's interpolation method to implement the multiplication in $\mathbb{F}_{q^5}$. Generally, interpolation methods have the drawback to increase the number of additions during a multiplication. Moreover, with interpolation methods we need to perform several divisions. Bajard et al. [3] study these methods and replace divisions by multiplications by large numbers. In this paper, we study Newton's interpolation and by carefully choosing our interpolation points, we perform divisions by small constants. While multiplication by large constants uses a general multiplier, we explain that divisions by small constants can be handled as 2 additions.

Note that Karatsuba and Toom Cook's formulæ can be found using Newton's interpolation by applying Newton's forward difference formula. We use the same

approach for a degree five extension, and we show that the number of extra additions is not large.

3.1 Newton's interpolation

We denote by $A(X) = a_0 + a_1 X + \ldots + a_{k-1} X^{k-1}$ and $B(X) = b_0 + b_1 X + \ldots + b_{k-1} X^{k-1}$ the expressions of A and B in $\mathbb{F}_{q^k}$. The interpolation method for the multiplication follows this steps

- Find $2k-1$ different values in $\mathbb{F}_q$ $\{\alpha_0, \alpha_1, \ldots, \alpha_{2k-2}\}$.
- Evaluate the polynomials $A(X)$ and $B(X)$ at these $2k-1$ values: $A(\alpha_0), \ldots, A(\alpha_{2k-2}), B(\alpha_0), \ldots, B(\alpha_{2k-2})$.
- Compute $C(X) = A(X) \times B(X)$ at these $2k-1$ values $C(\alpha_i) = A(\alpha_i) B(\alpha_i)$.
- Interpolate the polynomial $C(X)$ of degree $2k-2$ (with Newton's method).

Newton's interpolation constructs the polynomial $C(X)$ in the following way

$$\begin{aligned} c'_0 &= C(\alpha_0) \\ c'_1 &= (C(\alpha_1) - c'_0) \frac{1}{(\alpha_1 - \alpha_0)} \\ c'_2 &= \left((C(\alpha_2) - c'_0) \frac{1}{(\alpha_2 - \alpha_0)} - c'_1 \right) \frac{1}{(\alpha_2 - \alpha_1)} \\ &\vdots \end{aligned}$$

The reconstruction of $C(X)$ is done by

$$\begin{aligned} C(X) = c'_0 + c'_1(X - \alpha_0) + c'_2(X - \alpha_0)(X - \alpha_1) + \cdots \\ + c'_{2k-2}(X - \alpha_0)(X - \alpha_1) \ldots (X - \alpha_{2k-2}). \end{aligned}$$

This can be computed using Horner's scheme

$$C(X) = c'_0 + (X - \alpha_0) \left[c'_1 + (X - \alpha_1) \left(c'_2 + (X - \alpha_2) \left\langle \ldots \right\rangle \right) \right]$$

The global complexity of Newton's interpolation is the sum of the following operations:

1. the evaluations at α_i of $A(X)$ and $B(X)$
2. the $2k-1$ multiplications in $\mathbb{F}_q$ ($A(\alpha_i) \times B(\alpha_i)$)
3. the computation of the c'_i
4. Horner's scheme to find the expression of $C(X) = A(X) \times B(X)$ of degree $2k-1$.

3.2 Simplifying operations in Newton's interpolation

Since we want to multiply two polynomials of degree 4, we choose 9 values for the interpolation

$$\alpha_0 = 0, \alpha_1 = 1, \alpha_2 = -1, \alpha_3 = 2, \alpha_4 = -2, \alpha_5 = 4, \alpha_6 = -4, \alpha_7 = 3, \alpha_8 = \infty.$$

These values were chosen in order to minimize both the number of additions during the evaluation step and the costs of divisions by constants.

Complexity of the evaluations in α_i of A and B. For the first step, we have to evaluate $A(X)$ and $B(X)$ in the α_i. With the chosen values, evaluations of $A(X)$ and $B(X)$ are done using only additions and shifts in $\mathbb{F}_q$. Indeed, a product by a power of 2 is composed of shifts in binary base. In order to evaluate $A(X)$ at 2^j, we compute the products $a_i \times (2^j)^i$ which are shifts, and then the additions $\sum_{i=0}^{k-1} a_i (2^j)^i$ using a FFT scheme. For example, we describe the FFT scheme for the evaluation of $A(2)$ and $A(-2)$. First, we compute evaluations for even indices and odd indices, separately. Let $A_e = a_0 + a_2 \times 2^2 + (a_4 \times 2^2) \times 2^2$ and $A_o = a_1 \times 2 + (a_3 \times 2^2) \times 2$. The evaluations are then $A(2) = A_e + A_o$ and $A(-2) = A_e - A_o$.

As explained in [3], by writing down $3 = 2+1$ and $3^2 = 2^3+1$, the evaluation at 3 of $A(X)$ and $B(X)$ is composed only of shifts and additions. In practice, it is more efficient to design a direct procedure to multiply by 3 which is equivalent to one addition. This will be detailed in Sect. 4.

Adding the different costs, the evaluations of $A(X)$ and $B(X)$ have a total complexity of $48A_q$. Once we have performed the evaluations, we are able to compute the 9 multiplications $A(\alpha_i) \times B(\alpha_i)$, which are obtained with $9M_q$. The complexity of steps 1 and 2 is then $9M_q + 48A_q$.

Complexity of the computations of c'_j. The complete formulæ for computing the c'_j are

$$
\begin{aligned}
c_0 &= u_0 \\
c_1 &= u_1 - c_0 \\
c_2 &= (u_2 - c_0 + c_1)/2 \\
c_3 &= ((u_3 - c_0)/2 - c_1 - c_2)/3 \\
c_4 &= (((u_4 - c_0)/2 + c_1)/3 - c_2 + c_3)/4 \\
c_5 &= (((((u_5 - c_0)/4 - c_1)/3 - c_2)/5 - c_3)/2 - c_4)/6 \\
c_6 &= ((((((u_6 - c_0)/4 + c_1)/5 - c_2)/3 + c_3)/6 - c_4)/2 + c_5)/8 \\
c_7 &= (((((-u_7 + c_0)/3 + c_1)/2 + c_2)/4 + c_3 + c_4)/5 + c_5 - c_6)/7
\end{aligned}
$$

In order to compute the coefficients c'_j during Newton's interpolation, one has to compute divisions by differences of α_i. In a binary basis, divisions by a power of 2 are rather simple, since they are equivalent to shifts to the right, plus sometimes an addition (see below). We approximate a division by a power of 2 by $1A_q$. Among all the differences of the α_i we choose, eleven are not a power of 2. They are given in Table 1. In Sect. 4.2, our analysis of divisions by 3, 5, 7 shows that the complexity of these divisions is equivalent to $2A_q$. In order to compute the c'_j, we need $28A_q$, 11 divisions by 2, 4 or 8, and 11 divisions by 3, 5 or 7. Consequently, the complexity of computing the c'_j is $61A_q$.

Table 1. The problematic differences

$\alpha_3 - \alpha_2 = 3$	$\alpha_4 - \alpha_1 = -3$	$\alpha_5 - \alpha_1 = 3$	$\alpha_5 - \alpha_2 = 5$
$\alpha_5 - \alpha_4 = 6$	$\alpha_6 - \alpha_1 = -5$	$\alpha_6 - \alpha_2 = -3$	$\alpha_6 - \alpha_3 = -6$
$\alpha_7 - \alpha_0 = 3$	$\alpha_7 - \alpha_4 = 5$	$\alpha_7 - \alpha_6 = 7$	

Cost of the polynomial interpolation. We use Horner's scheme to find the expression of the product polynomial $C = A \times B$. More precisely, we have to compute

$$C(X) = ((((c'_8(X - \alpha_7) + c'_7)(X - \alpha_6) + c'_6)(X - \alpha_5) + c'_5) \dots$$
$$+c'_1)(X - \alpha_0) + c'_0.$$

We begin to compute from the inside to the outside. First, we compute the parenthesis $(c'_8(X - \alpha_7) + c'_7)$, next $((c'_8(X - \alpha_7) + c'_7)(X - \alpha_6) + c'_6)$, and so on.

Horner's scheme for the chosen values of α_is is composed only of shifts and additions. The total complexity of the polynomial reconstruction is $28A_q$.

Complexity of the polynomial reduction. We may use the same technique for polynomial reduction in both Montgomery's method and our interpolation method. Indeed, we may represent the finite field $\mathbb{F}_{q^5}$ using an irreducible reduction polynomial of the form $X^5 - \alpha$.

We consider q such that $q \equiv 1 \bmod 5$. Then the following result [12, Theorem 3.75] guarantees that such polynomials exist over $\mathbb{F}_q$.

Theorem 1. *[12, Theorem 3.75] Let $\mathbb{F}_{q^5}$ be a finite field, and let α be an element of $\mathbb{F}_q$. Then the binomial $X^5 - \alpha$ is irreducible in $\mathbb{F}_q[X]$ if and only if 5 divides the order e of $\alpha \in \mathbb{F}_q$, but not $(q-1)/e$.*

Moreover, in practice we may take α a small integer (such as 2 or 3). The reduction step needs 4 operations which are multiplications by α, but in practice they are computed as shifts and additions.

Table 2. Details of the operation count

Operation	Complexity
evaluation	$9M_q + 48A_q$
computation of c'_j	$61A_q$
interpolation	$28A_q$
Total	$9M_q + 137A_q$

3.3 Results and comparison

Table 3 gives the complexity of a multiplication with Montgomery's formulæ and with our interpolation formulæ.

Table 3. Cost of multiplication in $\mathbb{F}_{q^5}$

Montgomery	This work
$13M_q + 62A_q$	$9M_q + 137A_q$

We save 4 multiplications in $\mathbb{F}_q$ using interpolation whereas we add 75 additions considering Montgomery's formula. The method we propose is more efficient if a multiplication in $\mathbb{F}_q$ has a cost greater than 18 additions in $\mathbb{F}_q$. The benchmarks of the library we used show that for q prime, the ratio M_q/A_q depends on the size of q. Consequently, our method is more efficient than Montgomery's formula if $\log q > 512$. Our benchmarks are given in Sect. 4.

4 Technical details and implementation

If q is a prime power, then $a \in \mathbb{F}_q$ can be represented as a polynomial of degree k with coefficients $a_0, a_1, \ldots, a_{k-1} \in \mathbb{F}_p$ such that $p^k = q$. Then additions and divisions by small constants are performed on every coefficient. Hence in the remainder of this section, we assume that q is prime.

4.1 Cost of additions and shifts in C language

Our implementation is written in the C language. Over $\mathbb{F}_q$ with a and b of w 32-bit words, at each word addition, a carry must be taken into account. Indeed in C, an assembly instruction such as Add With Carry is not available. Algorithm 1 explains the processor behavior when performing an addition.

When computing a shift to the left written as $a \ll s$ in C, no carry appears. Hence this procedure is cheaper than an addition in $\mathbb{F}_q$. At each 32-bit word state ℓ, the instruction $r_\ell \leftarrow (a_\ell \ll s)\mathsf{Xor}(a_{\ell-1} \gg (32-s))$ is enough. It needs 1 reading because $a_{\ell-1}$ was already loaded in a register at the preceding state, 3 instructions and 1 writing, plus 2 counter imcrementations for the word a_ℓ and r_ℓ memory address. The total count is then about $8w$ instructions. To conclude, with a C implementation, the ratio Shift/Add $= 8/12 \approx 0.66$ is obtained. To improve the performance, a function which computes $a + (b \ll s)$ is provided. See details in Algorithm 2.

For the procedure in Algorithm 2 the ratio (Add with Shift)/Add is about $15/12 \approx 1.25$.

Finally, a direct multiplication by 3 is also used. As $3a = a + 2a$, this is performed as an Add with Shift, but neither the memory access for b_ℓ nor the counter incrementation for its address is needed. Hence the ratio Mult By 3 /

Algorithm 1 Addition in a prime field $\mathbb{F}_q$

INPUT : $a = a_{w-1}a_{w-2}\dots a_0 \in \mathbb{F}_q$ and $b = b_{w-1}b_{w-2}\dots b_0 \in \mathbb{F}_q$ of w 32-bit words
OUTPUT : $r = a + b = r_w r_{w-1} r_{w-2} \dots r_0$ not reduced mod q

1: $r_0 \leftarrow a_0 + b_0$
2: set carry
3: **for** $\ell \leftarrow 1, \dots, w-1$ **do**
4: $\quad \mathsf{tmp} \leftarrow a_\ell + b_\ell$ ▷ 2 readings, 1 instruction
5: $\quad r_\ell \leftarrow \mathsf{tmp} + \mathsf{carry}$ ▷ 1 instruction, 1 writing
6: $\quad$ carry update ▷ 3 instructions
7: ▷ 3 counter imcrementations for memory address of a_ℓ, b_ℓ, r_ℓ
8: **end for** ▷ 1 instruction
9: $a_w \leftarrow \mathsf{carry}$ ▷ overflow bit
10: **return** r ▷ $\approx 12w$ instructions

Algorithm 2 Shift to the left with addition in a prime field $\mathbb{F}_q$

INPUT : $a = a_{w-1}a_{w-2}\dots a_0 \in \mathbb{F}_q$ and $b = b_{w-1}b_{w-2}\dots b_0 \in \mathbb{F}_q$ of w 32-bit words, $0 < s < 32$
OUTPUT : $r = a + b2^s = a + (b \ll s) = r_w r_{w-1} r_{w-2} \dots r_0$ not reduced mod q

1: $r_0 \leftarrow a_0 + (b_0 \ll s)$
2: set carry
3: **for** $\ell \leftarrow 1, \dots, w-1$ **do**
4: $\quad \mathsf{tmp} \leftarrow a_\ell + (b_\ell \ll s)\mathsf{Xor}(b_{\ell-1} \gg (32-s))$ ▷ 2 readings, 4 instructions
5: $\quad r_\ell \leftarrow \mathsf{tmp} + \mathsf{carry}$ ▷ 1 instruction, 1 writing
6: $\quad$ carry update ▷ 3 instructions
7: ▷ 3 counter incrementations
8: **end for** ▷ 1 instruction
9: $a_w \leftarrow \mathsf{carry} + (b_{w-1} \gg (32-s))$ ▷ overflow bit
10: **return** r ▷ $\approx 15w$ instructions

Add is $13/12 \approx 1.08$. Practical results are given in Table 4. The benchmarks are close to the theoretical results. The compiler and processor type do not influence too much the timing results.

4.2 Division by small constants

Division by 2, 4 and 8 in a prime field $\mathbb{F}_q$. Let $a \in \mathbb{F}_q$. If the last significant bit of a is 0, a is even and computing $a/2$ is just a shift to the right. Otherwise, a is odd but as q is a large prime, q is odd; hence $a+q$ is even and $a/2 = (a+q)/2$ with a shift. There remains a slight detail : $a + q$ may induce a bit overflow. Indeed, the modular integers are normally smaller than q. If q is of $32w$ bits, $a+q$ may be of $32w+1$ bits. To avoid that, we shift a and q before adding them. To finish we add the carry loss in the shift. Writing $a = 2a' + 1$, $q = 2q' + 1$, a' is a shifted of one bit to the right, q' is the same for q. The result is obtained as $a/2 = (a+q)/2 = a' + q' + 1$.

Following the same idea, division by 4 or 8 is a shift with sometimes an addition. We write $a = 4a' + r_a$, $r_a \in \{0, 1, 2, 3\}$ and $q = 4q' + r_q$, $r_q \in \{1, 3\}$. If

Table 4. Theoretical and practical ratio Operation/Addition without modular reduction

Operation	Ratio Operation/Add								
	Theoretical	Our implementation in a prime field $\mathbb{F}_q$							
$\log q$		160	192	256	384	512	768	1024	1536
Shift to the left	0.66	0.50	0.50	0.50	0.48	0.46	0.42	0.47	0.42
Shift and Add	1.25	1.11	1.13	1.18	1.32	1.23	1.26	1.34	1.33
Multiplication by 3	1.08	0.88	0.89	1.02	0.97	1.00	1.00	1.08	1.09

$r_a = 0$, then the shift of two bits is enough; otherwise the value of $a/4$ is given in Table 5.

Table 5. Division by 4

	$r_a = 1$	$r_a = 2$	$r_a = 3$
$r_q = 1$	$a' + 3q' + 1$	$a' + 2q' + 1$	$a' + q' + 1$
$r_q = 3$	$a' + q' + 1$	$a' + 2q' + 2$	$a' + 3q' + 3$

Note that a', q' are just shifts of two bits of a and q, respectively. Moreover, $q', 2q'$ and $3q'$ can be precomputed. For division by 8, we follow the same method, considering that $a = 8a' + r_a$, $r_a \in \{0, 1, \ldots, 7\}$ and $q = 8q' + r_q$, $r_q \in \{1, 3, 5, 7\}$. Benchmarks are given in Table 7.

Divisions by 3, 5 and 7 in a prime field $\mathbb{F}_q$. For these cases, shifts are not possible. We present a detailed division by 3, and give the main idea for 5 and 7.

We write a basic division of a by 3, considering that a is composed of 32-bit words. For each word, the possible remainders are 0, 1 or 2 (10 in binary base). This leads to a 33 or 34 bit word to the next state if the remainder is not zero. Fortunately, we know that $2^{32} = 3 \cdot \texttt{0x55555555} + 1$ and $2 \cdot 2^{32} = 3 \cdot \texttt{0xaaaaaaaa} + 2$. This leads to Algorithm 3.

Now $a = 3a' + r_a$. If $r_a = 0$ then $a/3 = a'$. If not, write $q = 3q' + r_q$ (which can be precomputed). The result of the division is computed as explained in Table 6.

Table 6. Division by 3

	$r_a = 1$	$r_a = 2$
$r_q = 1$	$a' + 2q' + 1$	$a' + q' + 1$
$r_q = 2$	$a' + q' + 1$	$a' + 2q' + 2$

Algorithm 3 Division by 3 in a prime field $\mathbb{F}_q$

INPUT : $a = a_{w-1}a_{w-2}\dots a_0 \in \mathbb{F}_q$ of w 32-bit words
OUTPUT : $a' = a/3 = a'_{w-1}a'_{w-2}\dots a'_0$ and r such that $a = 3a' + r$

```
 1: a'_{w-1} ← a_{w-1} div 3; carry ← a_{w-1} mod 3
 2: for ℓ ← w − 2, ..., 0 do
 3:     if carry = 0 then                                          ▷ 1
 4:         a'_ℓ ← a_ℓ div 3; carry ← a_ℓ mod 3                    ▷ 1/3 3
 5:     else                                                       ▷ 1
 6:         if carry = 1 then                                      ▷ 2/3 1
 7:             a'_ℓ ← 0x55555555 + (a_ℓ div 3)                    ▷ 1/3 4
 8:             carry ← carry + (a_ℓ mod 3)                        ▷ 1/3 2
 9:         else                                                   ▷ 2/3 1
10:             a'_ℓ ← 0xaaaaaaaa + (a_ℓ div 3)                    ▷ 1/3 4
11:             carry ← carry + (a_ℓ mod 3)                        ▷ 1/3 2
12:         end if
13:         if carry ⩾ 3 then                                      ▷ 2/3 1
14:             carry ← carry −3; a'_ℓ ← a'_ℓ + 1                  ▷ 2/3 4
15:         end if
16:     end if
17: end for                                  ▷ 2 counter imcrementations + 1
18: return (a', carry)                                  ▷ ≈ 15w instructions
```

Cost of the divisions by small constants Division by 3 is is carefully detailed in Algorithm 3. Our counting shows that in Algorithm 3 we perform around $15w$ processor instructions. Since in $\frac{2}{3}$ of cases we have to add p' (as explained in Table 6), we have on average the ratio

$$\mathrm{Div}_q/A_q \simeq 2.$$

where Div_q denotes the cost of division by 3, 5 or 7. However, this number is just an approximation since the exact costs depends on

- the type of the compiler,
- the compiler directives,
- the number of cycles required for each processor instruction,
- the pipeline depth into the processor,
- the cache memory, etc.

Moreover, there are some conditional jumps in Algorithm 3. In an implementation, they may be replaced by access to a table indexed by the remainder's value. Table 7 gives a practical estimation of the ratio division/addition.

The idea for division by 5 or 7 is the same, except that computing a'_ℓ needs different values (see Table 8).

4.3 Implementation results

We implemented Montgomery's formula and our multiplication in C in order to compare them. The subfield $\mathbb{F}_q$ is simply built with $q \equiv 1 \mod 5$ a large random

Table 7. Theoretical and practical ratio Division by a small constant/Add

Operation	Ratio Operation/Add							
	Theoretical	Our implementation in a prime field $\mathbb{F}_q$						
$\log q$		160	192	256	384	512	768	1024
Division by 2	1	1.00	0.82	0.85	0.84	1.01	0.62	0.76
Division by 3	2	1.57	1.10	1.37	1.42	1.44	1.60	1.59
Division by 4	1	1.00	1.00	0.79	0.92	0.88	1.03	1.07
Division by 5	2	1.69	1.41	1.47	1.58	1.74	1.62	2.01
Division by 6	2	2.76	2.15	2.00	2.14	2.34	2.14	2.36
Division by 7	2	2.15	1.52	1.89	1.62	1.76	1.88	1.96
Division by 8	1	1.36	1.05	1.04	1.01	1.04	0.89	1.21

Table 8. Constants for division by 5 and 7

division by 5	division by 7
$2^{32} = 5 \cdot \texttt{0x33333333} + 1$	$2^{32} = 7 \cdot \texttt{0x24924924} + 4$
$2 \cdot 2^{32} = 5 \cdot \texttt{0x66666666} + 2$	$2 \cdot 2^{32} = 7 \cdot \texttt{0x49249249} + 1$
$3 \cdot 2^{32} = 5 \cdot \texttt{0x99999999} + 3$	$3 \cdot 2^{32} = 7 \cdot \texttt{0x6db6db6d} + 5$
$4 \cdot 2^{32} = 5 \cdot \texttt{0xcccccccc} + 4$	$4 \cdot 2^{32} = 7 \cdot \texttt{0x92492492} + 2$
	$5 \cdot 2^{32} = 7 \cdot \texttt{0xb6db6db6} + 6$
	$6 \cdot 2^{32} = 7 \cdot \texttt{0xdb6db6db} + 3$

prime number of cryptographic size, from 160 to 1536 bits. Our benchmarks on Montgomery's algorithm and our method use the same prime numbers q. The modular library implementing the arithmetic of $\mathbb{F}_q$ is LibCryptoLCH [5] and uses the Montgomery representation to perform a modular multiplication (see chapter 14 of [13]). This library is also written in C. Parameters such as maximum moduli size and size of words are set at compilation. We used a gcc compiler with `-O2` optimization directive. The code was running on a Pentium 64 bits 3GHz under Linux, Ubuntu 10.10. The reduction step (mod $X^5 - \alpha$) is done at each multiplication. Degree 5 extensions $\mathbb{F}_q[X]/(X^5 - \alpha)$ with very small α such as $\alpha = 2$ were found.

Depending on the size of q, the cost of a M_q in terms of A_q increases as shown in Table 9.

Table 9. Ratio M_q/A_q for different sizes of q

$\log q$	160	256	384	512	768	1024	1536
32 bits	5.2	7.1	12.1	16.1	26.6	36.3	50.0
64 bits	3.9	5.7	6.9	9.3	16.6	19.4	32.1

On this 64 bit processor, our formula is better than Montgomery's one for $\log q$ greater than 512, as shown in Figure 1. Implementation of additions in the

base field library is not optimized, so the ratio M_q/A_q takes quite small values for small sizes of q. The timing ratio between Montgomery's method and our algorithm is shown in Table 10.

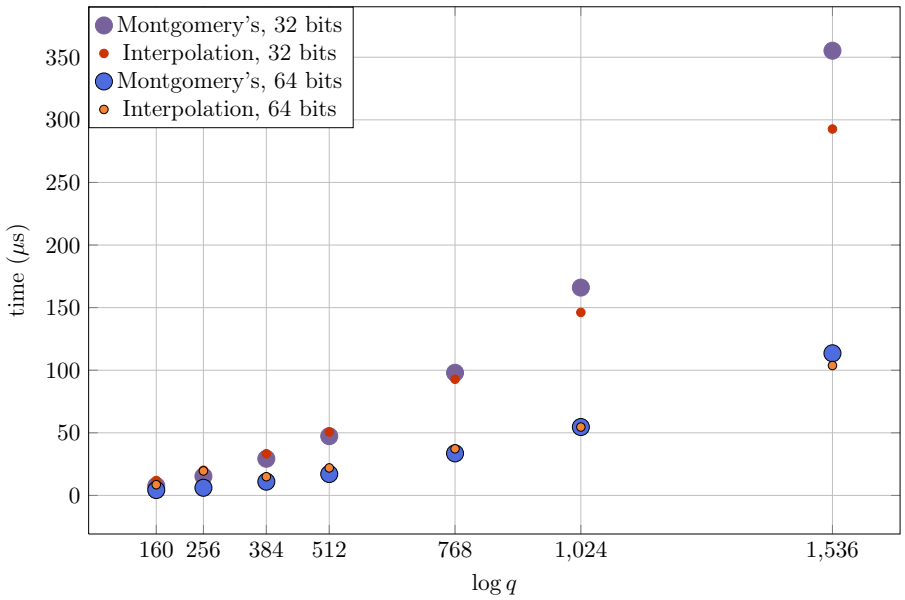

Fig. 1. Implementation results

Table 10. Timing ratio: Montgomery's formula results/ our's

$\log q$	160	256	384	512	768	1024	1536
32-bit words, theory	0.70	0.76	0.89	0.96	**1.08**	**1.15**	**1.21**
32-bit words, practice	0.64	0.75	0.88	0.93	**1.05**	**1.13**	**1.21**
64 bit words, theory	0.65	0.72	0.76	0.82	**1.00**	**1.01**	**1.12**
64 bit words, practice	0.49	0.31	0.73	0.77	0.90	**1.00**	**1.09**

5 Results and comparison for quintic and sextic polynomials

We use the same approach by interpolation to compute multiplication in extension fields of degree 6 and 7. Our results for an extension of degree 6 are the following

- the complexity of the evaluation step is $11M_q + 80A_q$,
- the complexity of the computation of the c'_i is $36A_q + 22\text{Div}_q$, where Div_q denotes the cost of division by 3, 5 or 7,
- the complexity of the Horner's scheme is $39A_q$.

As explained in Sect. 3.2, we count a division by 3, 5, 7 and 11 as $2A_q$. We perform 21 divisions by 2, 13 by 3, 6 by 5 and 3 by 7. Thus the total complexity of our multiplication is $11M_q + 199A_q$. We compare our results to Devegili et al. [6] most efficient result, and to Montgomery's one. The comparison is given in Table 11, where $M_{\mathbb{Z}}$ denotes the cost of multiplication by a small constant. Our method needs a smaller number of additions than Devegili et al.'s one.

Our results for an extension of degree 7 are the following

- the complexity of the evaluation step is $13M_q + 89A_q$,
- the complexity of the computation of the c'_i is $55A_q + 36\text{Div}_q$,
- the complexity of the Horner's scheme is $65A_q$.

Estimating the cost of a division by 3, 5, 7 and 11 by $2A_q$, the total complexity of our multiplication is $13M_q + 281A_q$. We perform here 31 divisions by 2 or power of 2, 21 divisions by 3, 9 by 5, 5 by 7 and one by 11. The comparison is given in Table 11. Our method is more efficient than Mongomery's one if the ratio M_q/A_q is greater than 8.5.

Table 11. Complexity of different method of 6-term and 7-term multiplications

Method	Devegili et al. [6]	Montgomery	this work
6-term	$11M_q + 93M_{\mathbb{Z}} + 236A_q$	$17M_q + 161A_q$	$11M_q + 199A_q$
7-term	–	$22M_q + 205A_q$	$13M_q + 281A_q$

6 Cryptographic use

We claim that our method is useful for cryptographic use in pairing-based cryptography and torus-based cryptography. We give in Tables 12 and 13 recommended security levels and the corresponding sizes of the field $\mathbb{F}_q$ for these applications.

Note that in order to achieve the 192 and 256 bit security levels, the size of the extension field $\mathbb{F}_{q^k}$ has to be within the range 8000-10000 and 14000-18000, respectively. The parameters given in Table 12 correspond to known families of pairing-friendly elliptic curves and the choices were made taking into account recommendations in [8]. Since our method is faster than Montgomery's formula if $\log q > 512$, our algorithm is interesting for implementations on these curves.

For torus-based cryptography, our method may be interesting for example when implementing $T_{30}(\mathbb{F}_q)$ for applications suggested in [17]. The choice of

Table 12. Pairing-based cryptography

Embedding degree	ρ-value	Security level	Size of q	Extension field
10	1.5	192	800	$\mathbb{F}_{q^{10}}$ (8000 bits)
15	1.5	192	576	$\mathbb{F}_{q^{15}}$ (8640 bits)
15	1.5	256	768	$\mathbb{F}_{q^{15}}$ (11520 bits)
20	1.375	256	704	$\mathbb{F}_{q^{20}}$ (14080 bits)
30	1.5	256	768	$\mathbb{F}_{q^{30}}$ (23040 bits)

Table 13. Torus-based cryptography

Security level	Torus	Size of $\mathbb{F}_q$	Size of extension field
128	$T_{30}(\mathbb{F}_q)$	102	$\mathbb{F}_{q^{30}}$ (3072 bits)
256	$T_{30}(\mathbb{F}_q)$	512	$\mathbb{F}_{q^{30}}$ (15360 bits)

parameters in Table 13 is done according to recommendations [1,17]. We suppose that the following tower of extensions is chosen when implementing $\mathbb{F}_{q^{30}}$

$$\mathbb{F}_{q^2} \subset \mathbb{F}_{q^6} \subset \mathbb{F}_{q^{30}}.$$

Then the cost of a multiplication in $\mathbb{F}_{q^6}$ is $15M_q + 72A_q$ using Karatsuba and Toom Cook algorithms, while the cost of an addition is $6A_q$. Our method is efficient if $M_{q^6}/A_{q^6} > 18$. This ratio depends on the value of M_q/A_q, which obviously depends on the type of the processor chosen. For example, using data in Table 9, we obtain $M_{q^6}/A_{q^6} \simeq 53.66$ for an implementation at 256 bit security level using a 32 bit architecture.

Finally, note that the arithmetic of $T_{30}(\mathbb{F}_q) \subset \mathbb{F}_{q^{30}}$ may be used to compress pairing values for curves with embedding degree 30. The size of q for such curves is given in Table 12.

7 Conclusion

We proposed an efficient arithmetic for the field $\mathbb{F}_{q^5}$, using a multiplication by interpolation. Our idea to use Newton's method of interpolation requires some divisions by small constants which are not a power of two but we have shown that these divisions have a slight cost. Our method can be applied to 6 and 7 degree extensions. In each case, the number of multiplications over the base field is smaller than the one in other known methods. The number of additions is larger but for cryptographic sizes of q, as shown in our implementation, our method is faster.

Acknowledgments

This work was supported in part by the French ANR-09-VERS-016 BEST Project. The authors express their gratitude to Jean Claude Bajard, Renaud Dubois and

Damien Vergnaud for helpful comments and careful proofreading and to Nicolas Guillermin for giving hints on the implementation part. The authors thank the anonymous reviewers of the Africacrypt conference for their useful comments.

References

1. Recommendations for Key Management, 2007. Special Publication 800-57 Part 1.
2. R. Avanzi and E. Cesena. Trace Zero Varieties over Fields of Characteristic 2 for Cryptographic Applications. In J. Chaumine, J. Hirschfield, and R. Rolland, editors, *Proceedings of SAGA 2007, The first Symposium on Algebraic Geometry and its Applications*, Number Theory and its Applications, pages 188–215. World Scientific, 2007.
3. J.C. Bajard, L. Imbert, and Ch. Negre. Arithmetic operations in finite fields of medium prime characteristic using the Lagrange representation. *IEEE Transactions on Computers*, 55(9):1167–1177, September 2006.
4. M. Bodrato. Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0. In *WAIFI 2007*, volume 4547 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2007.
5. Thales Communications. LibCryptoLCH Librairie cryptographique du Laboratoire Chiffre, 2011.
6. A. J. Devegili, C. Ó hÉigeartaigh, M. Scott, and R. Dahab. Multiplication and squaring on pairing-friendly fields. Cryptology ePrint Archive, Report 2006/471, 2006. `http://eprint.iacr.org/`.
7. D. Freeman. Constructing pairing-friendly elliptic curves with embedding degree 10. In F. Hess, S. Pauli, and M. Pohst, editors, *Algorithmic Number Theory Symposium-ANTS-VII*, volume 4076 of *Lecture Notes in Computer Science*, pages 452–465. Springer, 2006.
8. D. Freeman, M. Scott, and E. Teske. A taxonomy of pairing-friendly elliptic curves. *Journal of Cryptology*, 23:224–280, 2010.
9. R. Granger, D. Page, and N. Smart. On small characteristic algebraic tori in pairing based cryptography. *LMS Journal of Computation and Mathematics*, (9):64–85, 2006.
10. T. Itoh and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. *Info. and Comp.*, 78(3):171–177, 1988.
11. N. Koblitz and A. Menezes. Pairing-based cryptography at high security levels. In Nigel P. Smart, editor, *IMA Int. Conf.*, volume 3796 of *Lectures Notes in Computer Science*, pages 13–36, 2005.
12. R. Lidl and H. Niederreiter. *Finite Fields*. 2nd ed., Cambridge University Press, 1997.
13. A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptology*. CRC Press, 2001.
14. P. L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Transactions on Computers*, 54(3):362–369, 2005.
15. M. Naehrig, P. Barreto, and P. Schwabe. On compressible pairings and their computation. In S. Vaudenay, editor, *Progress in Cryptology AFRICACRYPT 2008*, volume 5023 of *LNCS*, pages 371–388. Springer.
16. K. Rubin and A. Silverberg. Torus-Based Cryptography. In D. Boneh, editor, *Advances in Cryptology CRYPTO 2003*, volume 2729 of *LNCS*, pages 349–365. Springer, 2003.

17. M. van Dijk, R. Granger, D. Page, K. Rubin, A. Silverberg, M. Stam, and D. Woodruff. Practical cryptography in high dimensional tori. volume 3494. Springer Verlag, 2005.
18. J. Von ZurGathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2003.

8 Appendix: Details for inversion

We have also counted the number of M_q for and inversion in $\mathbb{F}_{q^5}$. We consider that $5 \mid q-1$ and $\mathbb{F}_{p^5}$ is built as $\mathbb{F}_q[X]/(X^5-\alpha)$. Following the idea of T. Itoh and S. Tsujii [10] we perform an inversion in $\mathbb{F}_{q^5}$ as

$$a^{-1} = \frac{a^q\ a^{q^2} a^{q^3} a^{q^4}}{a\quad a^q\ a^{q^2} a^{q^3} a^{q^4}} = \frac{\bar{a}}{\mathrm{Norm}_{\mathbb{F}_{q^5}/\mathbb{F}_q}(a)}$$

In order to compute a^{q^i}, we need the values of X^{q^i}, $1 \leqslant i \leqslant 4$. We have $X^{q^i} = X^{q^i-1}X = X^{5\frac{q^i-1}{5}}X = X^{5\frac{q-1}{5}(1+q+\ldots+q^{i-1})}X = \alpha^{\frac{q-1}{5}(1+q+\ldots+q^{i-1})}X$. Let $\mu = \alpha^{(q-1)/5}$. Note that $\mu^5 = 1$ and that $\mu \neq 1$. Hence μ is a root of the polynomial $1+T+T^2+T^3+T^4$. Since $\mu \in \mathbb{F}_q$, we have $\mu^{q^i} = \mu$ and $\mu^{1+q+\ldots+q^{i-1}} = \mu^i$. So $X^{q^i} = \mu^i X$. Finally, for $1 \leqslant j \leqslant 4$, we have $(X^{q^i})^j = \mu^{ij \mod 5} X^j$. By writing $a = a_0 + a_1X + a_2X^2 + a_3X^3 + a_4X^4$, we obtain

$$a^{q^i} = a_0 + a_1\mu X + a_2\mu^{2i \mod 5}X^2 + a_3\mu^{3i \mod 5}X^3 + a_4\mu^{4i \mod 5}X^4,\ 1 \leqslant i \leqslant 4$$

Then we compute the numerator of the expression a^{-1} above.

$a^q\ a^{q^2} a^{q^3} a^{q^4} \mod 1+\mu+\mu^2+\mu^3+\mu^4 =$
$a_4^4X^{16}$
$- a_3a_4^3X^{15}$
$+ (-a_2a_4^3 + a_3^2a_4^2)X^{14}$
$+ (-a_1a_4^3 + 2a_2a_3a_4^2 - a_3^3a_4)X^{13}$
$+ (-a_0a_4^3 + 2a_1a_3a_4^2 + a_2^2a_4^2 - 3a_2a_3^2a_4 + a_3^4)X^{12}$
$+ (-3a_0a_3a_4^2 - 3a_1a_2a_4^2 + 2a_1a_3^2a_4 + 2a_2^2a_3a_4 - a_2a_3^3)X^{11}$
$+ (2a_0a_2a_4^2 + 2a_0a_3^2a_4 + a_1^2a_4^2 - a_1a_2a_3a_4 - a_1a_3^3 - a_2^3a_4 + a_2^2a_3^2)X^{10}$
$+ (2a_0a_1a_4^2 - a_0a_2a_3a_4 - a_0a_3^3 - 3a_1^2a_3a_4 + 2a_1a_2^2a_4 + 2a_1a_2a_3^2 - a_2^3a_3)X^9$
$+ (a_0^2a_4^2 - a_0a_1a_3a_4 - 3a_0a_2^2a_4 + 2a_0a_2a_3^2 + 2a_1^2a_2a_4 + a_1^2a_3^2 - 3a_1a_2^2a_3 + a_2^4)X^8$
$+ (2a_0^2a_3a_4 - a_0a_1a_2a_4 - 3a_0a_1a_3^2 + 2a_0a_2^2a_3 - a_1^3a_4 + 2a_1^2a_2a_3 - a_1a_2^3)X^7$
$+ (2a_0^2a_2a_4 + a_0^2a_3^2 + 2a_0a_1^2a_4 - a_0a_1a_2a_3 - a_0a_2^3 - a_1^3a_3 + a_1^2a_2^2)X^6$
$+ (-3a_0^2a_1a_4 - 3a_0^2a_2a_3 + 2a_0a_1^2a_3 + 2a_0a_1a_2^2 - a_1^3a_2)X^5$
$+ (-a_0^3a_4 + 2a_0^2a_1a_3 + a_0^2a_2^2 - 3a_0a_1^2a_2 + a_1^4)X^4$
$+ (-a_0^3a_3 + 2a_0^2a_1a_2 - a_0a_1^3)X^3$
$+ (-a_0^3a_2 + a_0^2a_1^2)X^2$
$- a_0^3a_1X$
$+ a_0^4$

We simplify this formula as follows

$$\begin{aligned}
\bar{a} = {} & (a_0^2(2a_1a_3 - a_0a_4 + a_2^2) + a_1^2(a_1^2 - 3a_0a_2 - 3a_3a_4\alpha) + a_2^2(2a_1a_4 - a_2a_3)\alpha \\
& + a_3^2(2a_1a_2 - a_0a_3)\alpha + a_4^2(2a_0a_1 + \alpha(a_3^2 - a_2a_4))\alpha - a_0a_2a_3a_4\alpha)X^4 \\
& + (a_0^2(2a_1a_2 - a_0a_3) + a_1^2(-a_0a_1 + 2a_2a_4\alpha) + a_2^2(a_2^2 - 3a_0a_4 - 3a_1a_3)\alpha \\
& + a_3^2(a_1^2 + 2a_0a_2 - a_3a_4\alpha)\alpha + a_4^2(a_0^2 + (2a_2a_3 - a_1a_4)\alpha)\alpha - a0a_1a_3a_4\alpha)X^3 \\
& + (a_0^2(a_1^2 - a_0a_2 + 2a_3a_4\alpha) + a_1^2(2a_2a_3 - a_1a_4)\alpha + a_2^2(2a_0a_3 - a_1a_2)\alpha \\
& + a_3^2(-3a_0a_1 + (a_3^2 - 3a_2a_4\alpha)\alpha) + a_4^2(a_2^2 - a_0a_4 + 2a_1a_3)\alpha^2 - a_0a_1a_2a_4\alpha)X^2 \\
& + (a_0^2(-a_0a_1 + (2a_2a_4 + a_3^2)\alpha) + a_1^2(2a_0a_4 - a_1a_3 + a_2^2)\alpha + a_2^2(-a_0a_2 + 2a_3a_4\alpha)\alpha \\
& + a_3^2(2a_1a_4 - a_2a_3)\alpha^2 + a_4^2(-3a_0a_3 - 3a_1a_2 + a_4^2\alpha)\alpha^2 - a_0a_1a_2a_3\alpha)X \\
& + (a_0^2(a_0^2 + (-3a_1a_4 - 3a_2a_3)\alpha) + a_1^2(2a_0a_3 - a_1a_2)\alpha + a_2^2(2a_0a_1 - a_2a_4\alpha)\alpha \\
& + a_3^2(2a_0a_4 - a_1a_3 + a_2^2)\alpha^2 + a_4^2(2a_0a_2 + a_1^2 - a_3a_4\alpha)\alpha^2 - a_1a_2a_3a_4\alpha^2)
\end{aligned}$$

We precompute $a_0^2, a_1^2, a_2^2, a_3^2, a_4^2$ and a_0a_1, a_0a_2, a_0a_3, a_0a_4, a_1a_2, a_1a_3, a_1a_4, a_2a_3, a_2a_4, a_3a_4. This leads to $5S_q+10M_q$. With this method, computing $\bar{a}$ needs $6M_q$ for each coefficient, hence $30M_q$ altogether. To compute the norm as $a\cdot\bar{a}$, we need an extra cost of $5M_q$. Indeed, by writing $\bar{a} = \bar{a}_0+\bar{a}_1X+\bar{a}_2X^2+\bar{a}_3X^3+\bar{a}_4X^4$, we have $a \cdot \bar{a} = a_0\bar{a}_0 + \alpha(a_1\bar{a_4} + a_2\bar{a}_3 + a_3\bar{a}_2 + a_4\bar{a}_1)$. The total count is $I_{q^5} = 45M_q + 5S_q + I_q$. In [2], Avanzi and Cesena report a cost of $50M_q + I_q$.