Smoothness Testing of Polynomials over Finite Fields

Jean-François Biasse and Michael J. Jacobson Jr.

Department of Computer Science, University of Calgary
2500 University Drive NW
Calgary, Alberta, Canada T2N 1N4

Abstract

We present an analysis of Bernstein's batch integer smoothness test when applied to the case of polynomials over a finite field $\mathbb{F}_q$. We compare the performance of our algorithm with the standard method based on distinct degree factorization from both an analytical and a practical point of view. Our results show that the batch test offers no advantage asymptotically, and that it offers practical improvements only in a few rare cases.

1 Introduction

Smoothness testing is an essential part of many modern algorithms, including index-calculus algorithms for a variety of applications. Algorithms for integer factorization, discrete logarithms in finite fields of large characteristic, and computing class groups and fundamental units of number fields require smoothness testing of integers. Testing polynomials over finite fields for t-smoothness, i.e., determining whether all irreducible factors have degree less than or equal to t, is also important in other settings. For example, the relation search performed to solve the discrete logarithm problem in the Jacobian of a genus g hyperelliptic curve over a finite field $\mathbb{F}_q$ by the Enge-Gaudry method [7] requires testing of a large amount of degree-g polynomials over $\mathbb{F}_q$ for t-smoothness. Smoothness testing of polynomials is also used in the cofactorization stage of sieving algorithms in function fields (see [11] for a survey mentioning their relevance to the discrete logarithm in finite fields). The sieve selects candidate polynomials over $\mathbb{F}_2$ that are likely to be smooth, and then these are rigorously tested for smoothness. The discrete logarithm problem in the Jacobian of a hyperelliptic curve over $\mathbb{F}_q$ can also be solved by using sieving methods [14] in an analogous manner.

In [2], Bernstein presented an algorithm to test the smoothness of a batch of integers that runs in time $O\left(b(\log b)^2 \log\log b\right)$, where b is the total number of input bits. This represents a significant improvements over the elliptic curve method described by Lenstra [9] which is conjectured to work in time $O(be^{\sqrt{(2+o(1))\log(b)\log\log(b)}})$. This method has been used successfully in a number of contexts, for example by Bisson and Sutherland as part of an algorithm for computing the endomorphism ring of an elliptic curve over a finite field [4] and by the authors for computing class groups in quadratic fields [3].

Bernstein's algorithm can be adapted easily for smoothness testing of polynomials over finite fields. However, it is not clear how much of a practical impact the resulting algorithm would have because, unlike the integer case, testing a polynomial over $\mathbb{F}_q$ for t-smoothness can be done in polynomial time with respect to the input, as described by Coppersmith [5]. Several variants of Coppersmith's method exist and are used in practice; we refer to the one described by Jacobson, Menezes and Stein [8]. When using "schoolbook" polynomial arithmetic, the number of field multiplications required for this method is in $O(d^2 t\log q + d^{2+\epsilon})$ where d is the degree of the polynomial to be tested. With asymptotically faster algorithms, this improves to $O(d^{1+\epsilon}t\log q)$.

The main idea of Bernstein's algorithm is to test a batch of polynomials for smoothness simultaneously. The product of these polynomials is first computed, followed by a "remainder tree", resulting in the product of all irreducible of degree t or less modulo each individual polynomial. Those that result in zero are t-smooth. The point is that much of the arithmetic is done with very large degree polynomials where the asymptotically fastest algorithms for polynomial arithmetic work best. When compared with Coppersmith's method, where the single polynomial operands have relatively small degree, the hope is that the use of these asymptotically faster algorithms results in an improved amortized cost.

In this paper, we describe our adaptation of Bernstein's algorithm for smoothness testing of polynomials over $\mathbb{F}_q$ and compare its performance with that of Jacobson, Menezes and Stein [8]. We show that the amortized number of field operations is in $O\left(dt\log(q) + d^{1+\epsilon}\right)$, almost the same as that of the standard method. We present numerical results obtained with a C++ implementation based on the libraries GMP, GF2X, and NTL confirming our analysis (implementation is available upon request). We test our algorithm on a number of examples of practical relevance and show that the batch algorithm does not offer an improvement.

In Section 2, we briefly review the main polynomial multiplication and remainder algorithms and their complexities in terms of field operations. We recall Coppersmith's smoothness test, as described in [8] in Section 3, and give a complexity analysis. In the next section we describe our adaptation of Bernstein's algorithm to polynomials over finite fields, followed by its complexity analysis. We conclude with numerical results demonstrating the algorithm's performance in practice.

2 Arithmetic of polynomials

Let $A, B \in \mathbb{F}_q[x]$ where $\deg(A) = a$, $\deg(B) = b$ with $a \geq b$, and $q = p^m$ where p is a prime. We express operation costs in terms of the number of multiplications in $\mathbb{F}_q$, as a function of a and b, required to perform the operation.

Most implementations of polynomial arithmetic use multiple algorithms, selecting the most efficient one based on the degrees of the operands. Our subsequent analysis considers three algorithms, the basic "schoolbook" method, the Karatsuba method, and a sub-quadratic complexity algorithm using fast Fourier transform (FFT).

The "schoolbook" method has the following costs:

- computing AB requires $(a+1)(b+1)$ multiplications in $\mathbb{F}_q$;
- computing $A \bmod B$ requires $(b+1)(a-b+1)$ multiplications in $\mathbb{F}_q$.

Asymptotically, both algorithms have complexity $O(d^2)$. The Karatsuba method requires $O(d^{\log_2 3})$ field multiplications. The Schönhage-Strassen method [12] based on the Fast Fourier Transform (FFT) requires $O(d\log(d)\log\log(d))$ field multiplications where $d = \max(a, b)$.

Fast multiplication allows a fast remainder operation via Newton division running in time $2.5\mathcal{M}(2(a-b+1)) + \mathcal{M}(2b)$ where $\mathcal{M}(d)$ is the cost of the multiplication between two degree-d polynomials. In particular, the cost of reducing a degree-$2d$ polynomial modulo a degree-d polynomial is approximately $3.5\mathcal{M}(d)$. Combined with the FFT, this gives us a remainder algorithm running in time $O(d\log(d)\log\log(d))$ where $d = \max(b, a-b)$.

Asymptotically, we will write the cost of multiplication and remainder computation as $O(d^\theta)$ for some real constant theta $1 < \theta \leq 2$ depending on the particular algorithm used. For the schoolbook algorithms we have $\theta = 2$, for Karatsuba $\theta = \log_2 3$, and for FFT we can take $\theta > 1$ arbitrarily small. Similarly, we will use $O(d^\epsilon)$ to denote logarithmic functions of d.

3 Smoothness Testing of Single Divisors

To assess the impact of our batch smoothness test for polynomials, we need a rigorous analysis of Coppersmith's method that incorporates practical improvements such as those described by Jacobson, Menezes and Stein [8]. In this section, we remind the reader of this method and provide an analysis of the cost according to the framework defined in Section 2.

Suppose we have a polynomial N of degree d and that we want to determine whether N is t-smooth. A well-known fact about polynomials over $\mathbb{F}_q$ is that $x^{q^i} - x$ is equal to the product of all irreducible polynomials of degree dividing i (see for example [10]). This observation can be used in an analogue manner to efficient distinct-degree factorization algorithms for an efficient smoothness-testing algorithm as follows:

1. Let $l = \lfloor t/2 \rfloor$. Compute $H = (x^{q^{l+1}} - x)(x^{q^{l+1}} - x) \dots (x^{q^t} - x) \pmod N$.
2. Compute $H = H^d \pmod N$.
3. If $H = 0$, then N is t-smooth.

If N is t-smooth and square-free, then $H = 0$ after Step 1, since H is divisible by all polynomials of degree $\leq t$. The second step checks whether H is divisible by all polynomials of degree $\leq t$ with multiplicity $\deg(N)$, so any factors of N occurring to high powers will be detected.

The smoothness-testing algorithm is presented in Algorithm 1. In order to facilitate the subsequent analysis, this algorithm is presented in detail.

Algorithm 1 Coppersmith Smoothness Test

Input: $N \in \mathbb{F}_q[x]$ with $\deg(N) = d$, $t \in \mathbb{Z}$
Output: "yes" if N is t-smooth, "no" otherwise
1: {Compute $P = x^{q^{\lfloor t/2 \rfloor}} \pmod N$}
2: Set $l = \lfloor t/2 \rfloor$.
3: $P = x^q \pmod N$)
4: **for** $i = 1$ to l **do**
5: Compute $P = P^q \pmod N$
6: **end for**
7: {Compute $H = (x^{q^{l+1}} - x)(x^{q^{l+1}} - x) \dots (x^{q^t} - x) \pmod N$.}
8: Let $H = 1$.
9: **for** $i = l + 1$ to t **do**
10: $P = P^q \pmod N$. {Here $P = x^{q^i} \pmod N$}
11: $Q = P - x$ {Here $Q = (x^{q^i} - x) \pmod N$}
12: Make Q monic.
13: $H = HQ \pmod N$ {Here $H = (x^{q^{l+1}} - x) \dots (x^{q^i} - x) \pmod N$}
14: If $H = 0$ go to Step 18.
15: **end for**
16: {Compute $H = H^d \pmod N$}
17: $H = H^d \pmod N$
18: **return** "yes" if $H = 0$, "no" otherwise

Theorem 3.1. *Algorithm 1 requires $O(d^\theta t \log q + d^{\theta+\epsilon})$ multiplications in $\mathbb{F}_q$, assuming multiplication and remainder of polynomials of degree d require $O(d^\theta)$ multiplications in $\mathbb{F}_q$.*

Proof. We have that $\deg(N) = d$.

- Steps 2-6 consist of $l = \lfloor t/2 \rfloor$ exponentiations to the power of q modulo N. Each of these costs $O(d^\theta \log q)$ multiplications, for a total of $O(d^\theta t \log q)$ multiplications.

- Steps 8-15 consist of $t - l = \lceil t/2 \rceil$ iterations, each of which performs:
  - one qth power modulo N ($O(d^\theta \log q)$ multiplications),
  - making Q monic ($d - 1$ field multiplications),
  - one multiplication modulo N ($O(d^\theta)$ multiplications)

  The total is $O(d^\theta t \log q)$.

- Step 17 performs a dth power modulo N, costing $O(\log(d)d^\theta)$ field multiplications.

Thus, the total number of multiplications in Algorithm 1 is in $O(d^\theta t \log q + d^\theta \log d)$ and the result follows. □

Depending on which version of multiplication and remainder is used, we obtain the following corollaries.

Corollary 3.2. *If schoolbook arithmetic is used, Algorithm 1 requires $O(d^2 t \log q + d^{2+\epsilon})$ field multiplications.*

Corollary 3.3. *If Karatsuba arithmetic is used, Algorithm 1 requires $O(d^{\log_2 3} t \log q + d^{\log_2(3)+\epsilon})$ field multiplications.*

Corollary 3.4. *If FFT arithmetic is used, Algorithm 1 requires $O(d^{1+\epsilon} t \log q + d^{1+\epsilon})$ field multiplications.*

4 Batch smoothness test of polynomials

We now present Bernstein's batch smoothness test [2] applied to polynomials over a finite field. Let $P_1, \ldots, P_m \in \mathbb{F}_q[x]$ be the irreducible polynomials of degree at most t, and $N_1, \ldots, N_k$ be polynomials that we want to test for $t-$smoothness. Note that the algorithm will work for any set of irreducible polynomials — for example, when solving the discrete logarithm problem in the Jacobian of a hyperelliptic curve we would only take irreducibles that split or ramify. The algorithm determines which of the N_i factor completely over the set of irreducibles.

The batch smoothness test starts with the computation of $P = P_1 \ldots P_m$ by means of a product tree structure. To compute this tree, we begin with the products of pairs of leaves of the tree and recursively compute the products of pairs of elements one level higher in the tree until we reach the root, which equals P. This process is depicted in Figure 1.

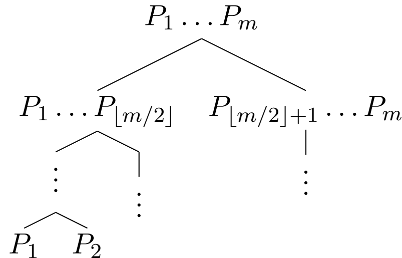


Figure 1: Illustration of product tree to compute $P = P_1 \ldots P_m$.

Note that in practice the product tree is not implemented recursively; instead, all nodes in the tree are stored in an array and index arithmetic is used to find the parent of the two children being multiplied. Note also that in the context of an index-calculus algorithm this computation need only be done once at the beginning when the factor base is computed.

Given P, we then compute $P \bmod N_1, \ldots, P \mod N_k$ by computing a remainder tree. We first compute the product tree of $N_1, \ldots, N_k$ as described above, and replace the root $N_1 \ldots N_k$ by $P \bmod N_1 \ldots N_k$, Then, using the fact that

$$P \mod N = (P \mod NM) \bmod N, \tag{1}$$

for $N, M \in \mathbb{F}_q[x]$, we recursively replace each node's children with the value stored in the node modulo the value stored in the child. At the end, Equation 1 guarantees that the leaves in the tree will contain $P \bmod N_i$ for every $i \leq k$. This process is illustrated in Figure 2.

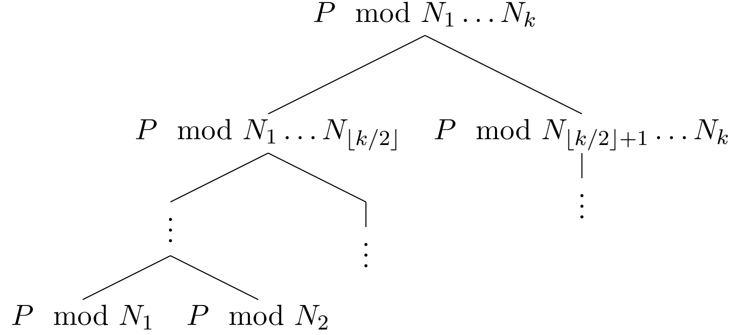


Figure 2: Illustration of the remainder tree of P and $N_1, \ldots, N_k$.

At the end of this process, if a leaf node stores zero, then we know that the original value is smooth with respect to the P_i and square-free. To account for higher multiplicities of the P_i, we could raise the values of each non-zero leaf node ($P \bmod N_i$) to the power of an exponent at least as large as $\deg(N_i)$, as in the algorithm from the previous section. In fact, to really amortize the cost of this operation, it is even better to perform the exponentiation at the root of the remainder tree, where the degree of the polynomials involved is the highest (thus exploiting asymptotically fast arithmetic). Therefore, prior to computing the remainder tree, we could update P by

$$P \leftarrow P^e \bmod N_1 \cdots N_k,$$

where e is an exponent at least as large as $\max_i \deg(N_i)$. This method returns all smooth values without false positive. Based on an idea of Coppersmith [5], we use a variant that avoids the exponentiation, but which can return false positive. It consists of calculating $N_i' \cdot P \bmod N_i$ at each non-zero leave to account for multiple roots in N_i. Note that this operation could in fact be performed on the root, but we expect it to produce too many false positive whithout improving the theoretical complexity.

Most index-calculus algorithms make use of "large prime" variants, where terms that are completely t-smooth except for a small number of factors of degree less than a given large prime bound are also useful. In order to detect such partially-smooth polynomials, we remove all the factors of degree at most t from N_i by computing $N_i / \gcd((P \bmod N_i)^{\deg(N_i)}, N_i)$. If this quantity has degree at most our given large prime bound, then we accept N_i as being partially smooth.

This method is summarized in Algorithm 2.

5 Complexity analysis

To compare the computational cost of Algorithm 2 (which tests a batch of polynomials) with that of Algorithm 1 (which tests one), we provide an analysis of the amortized cost of testing a batch of k polynomials of degree d over $\mathbb{F}_q$. We incorporate the crossover points between schoolbook, Karatsuba, and FFT-based polynomial multiplication in our analysis. Indeed, in practical applications, d is not necessarily very large, so it is interesting to assess how the algorithm performs

Algorithm 2 Batch smoothness test

Input: $\mathcal{B} = \{P_1, \cdots, P_m\}$, $N_1, \cdots, N_k$, large prime bound t_{lp} (no large primes if $t_{lp} = 0$).
Output: List L of $\mathcal{B}$-smooth N_i.

1: Compute $P = P_1 \dots P_m$ using a product tree.
2: Compute $P \mod N_1, \dots, P \mod N_k$ using a remainder tree.
3: $L = \{\}$.
4: **for** $i = 1$ to k **do**
5: $\quad y_i = P \mod N_i$, $y_i \leftarrow N_i' \cdot y_i \bmod N_i$.
6: $\quad$ **if** $y_i = 0$ or ($t_{lp} > 0$ and $\deg(N_i/\gcd(y_i, N_i)) \leq t_lp$) **then**
7: $\quad\quad L \leftarrow L \cup \{N_i\}$.
8: $\quad$ **end if**
9: **end for**
10: **return** L

in this setting. Let d_0 the degree for which Karatsuba multiplication becomes faster than the schoolbook one and d_1 the crossover point between FFT and Karatsuba. The crossover points can be observed experimentally, as shown in Section 6.

5.1 Cost of the product tree

The cost of the batch smoothness test is amortized, which means that if we test a batch of k polynomials, we divide the total cost by k.

Theorem 5.1. *The amortized cost of calculating the product tree of a batch of k polynomials of degree d over $\mathbb{F}_q$ is in*

$$O\left(d\left(d_0 + d_1^{\log_2(3)-1} - d_0^{\log_2(3)-1} + \frac{(kd)^\epsilon - d_1^\epsilon}{2^\epsilon - 1}\right)\right).$$

Proof. We begin by analyzing the cost of the steps using "schoolbook" multiplication. We denote by $i \geq 1$ the level of the product tree we are calculating, where $i = 1$ corresponds to the leaves. At each level, we perform $\frac{k}{2^i}$ multiplications between polynomials of degree $2^{i-1}d$. We switch to Karatsuba when $2^i d = d_0$, that is when $i = i_0 := \log\left(\left\lceil \frac{d_0}{d} \right\rceil\right)$. The cost of the computation of a level $i \leq i_0$ of the product tree is in

$$O\left(\frac{k}{2^i}\left(2^i d\right)^2\right) \subseteq O\left(k 2^i d^2\right).$$

The combined cost of all these levels is in

$$O\left(kd^2 \sum_{i=1}^{\log\left(\left\lceil \frac{d_0}{d} \right\rceil\right)} 2^i\right) \subseteq O\left(kd^2 2^{\log\left(\left\lceil \frac{d_0}{d} \right\rceil\right)}\right) \subseteq O(kdd_0).$$

When $i_0 \leq i < i_1 := \log\left(\left\lceil \frac{d_1}{d} \right\rceil\right)$, we use Karatsuba multiplication, and the cost of a level of the tree is in

$$O\left(\frac{k}{2^i}\left(2^i d\right)^\theta\right) \subseteq O\left(kd^\theta \left(2^{\theta-1}\right)^i\right),$$

where $\theta = \log_2(3)$. The combined cost of all these level is in

$$O\left(kd^\theta \left(\sum_{i=\log\left(\left\lceil \frac{d_0}{d} \right\rceil\right)}^{\log\left(\left\lceil \frac{d_1}{d} \right\rceil\right)} \left(2^{\theta-1}\right)^i\right)\right) \subseteq O\left(kd\left(\frac{d_1^{\theta-1} - d_0^{\theta-1}}{2^{\theta-1} - 1}\right)\right)$$

When $i \geq i_1$, we use FFT multiplication and the cost of computing one level of the product tree is in

$$O\left(\frac{k}{2^i}\left(2^i d\right)^\theta\right) \subseteq O\left(kd^\theta\left(2^{\theta-1}\right)^i\right),$$

where $\theta = 1+\epsilon$. The combined cost of all these level is in

$$O\left(kd^\theta\left(\sum_{i=\log\left(\left\lceil\frac{d_1}{d}\right\rceil\right)}^{\log(k)}\left(2^{\theta-1}\right)^i\right)\right) \subseteq O\left(kd\left(\frac{(kd)^{\theta-1}-d_1^{\theta-1}}{2^{\theta-1}-1}\right)\right)$$

The result follows by adding the cost of the computation of the levels $i \leq i_0$, $i_0 < i \leq i_1$ and $i \geq i_1$ and dividing by k to get the amortized cost. □

As this occurs in particular for polynomials in $\mathbb{F}_{2^e}[X]$, it is interesting to consider the case where no implementation of the FFT multiplication is available.

Corollary 5.2. *With the notations of Theorem 5.1, the amortized cost of calculating the product tree of a batch of k polynomials of degree d over $\mathbb{F}_q$ with only schoolbook and Karatsuba multiplication is in*

$$O\left(d\left(d_0+(kd)^{\log_2(3)-1}-d_0^{\log_2(3)-1}\right)\right).$$

Proof. The proof immediately follows from that of Theorem 5.1. It suffices to remove the levels $i_0 < i \leq i_1$. □

5.2 Cost of the remainder tree

At the level $i < \log(k)$ of the remainder tree, we reduce $\frac{k}{2^i}$ degree $2^i d$ polynomials modulo $2^{i-1}d$ polynomials. As stated in Section 2, this has the same cost as performing $\frac{k}{2^i}$ multiplications between degree-$2^{i-1}d$ polynomials. The computation of the root of the remainder tree (which comes first), consists of the reduction of $P = P_1 \cdots P_m$ modulo $N_1 \cdots N_k$. Since the cost of the product tree and remainder tree increases with k, we only consider the case $\deg(P) \geq kd$, as any other batch size would not be optimal. Then the amortized cost of the computation of the root of the remainder tree is in

$$O\left(\frac{\deg(P)^\theta}{k}\right),$$

where $\theta = 1+\epsilon$ if FFT multiplication is available, and $\theta = \log_2(3)$ otherwise. The total cost of the remaining levels is the same as that of the product tree. The last operation consists of k multiplications $N_i' \cdot P \bmod N_i$ between degree d polynomials. Depending on the size of d, this is in the same complexity class as the levels of the product tree using either plain multiplication, Karatsuba, or FFT. It therefore does not appear as an extra term in the overall complexity.

Theorem 5.3. *The amortized cost of calculating the remainder tree of a batch of k polynomials of degree d over $\mathbb{F}_q$ is in*

$$O\left(\frac{\deg(P)^{1+\epsilon}}{k}+d\left(d_0+d_1^{\log_2(3)-1}-d_0^{\log_2(3)-1}+\frac{(kd)^\epsilon-d_1^\epsilon}{2^\epsilon-1}\right)\right).$$

As previously, we can easily derive an analogue when only schoolbook and Karatsuba multiplication are available.

Corollary 5.4. *If only schoolbook and Karatsuba multiplications are used, the amortized cost of calculating the remainder tree of a batch of k polynomials of degree d over $\mathbb{F}_q$ is in*

$$O\left(\frac{\deg(P)^{\log_2(3)}}{k}+d\left(d_0+(kd)^{\log_2(3)-1}-d_0^{\log_2(3)-1}\right)\right).$$

5.3 Optimal size of batch

Let us find the optimal value of k. Whether we use FFT or not, the cost function has the shape

$$c(k) = A/k + Bk^{\theta-1} + C,$$

where $A = \deg(P)^\theta$, $B = \frac{d^\theta}{2^{\theta-1}-1}$, C does not depends on k and $\theta = 1+\epsilon$ if we use FFT multiplication, $\theta = \log_2(3)$ otherwise. A critical point for this function is attained for

$$k_{opt} = \left(\frac{A}{(\theta-1)B}\right)^{1/\theta} \in O\left(\frac{\deg(P)}{d}\right).$$

5.4 Overall cost

By combining the optimmal size of batch with the expression of the cost of the remainder tree as a function of k, we naturally get the overall cost.

Theorem 5.5. *The amortized cost of Algorithm 2 is given by*

$$O\left(d\deg(P)^\epsilon + d\left(d_0 + d_1^{\log_2(3)-1} - d_0^{\log_2(3)-1} + \frac{(\deg(P))^\epsilon - d_1^\epsilon}{2^\epsilon - 1}\right)\right)$$

when FFT multiplication is available.

Corollary 5.6. *If only schoolbook and Karatsuba multiplications are used, the amortized cost of Algorithm 2 is in*

$$O\left(d\deg(P)^{\log_2(3)-1} + d\left(d_0 - d_0^{\log_2(3)-1}\right)\right).$$

Thus, as a function of d,, the batch algorithm has roughly the same asymptotic complexity as the single polynomial test. We also see that for $d \le d_0$, where the single polynomial test uses only schoolbook arithmetic but the batch algorithm may use Karatsuba or FFT, the batch algorithm is also not expected to offer much improvment. In particular, the costs given in Theorem 5.5 and its corollary both have a term of the form dd_0 which, for d close to d_0 is d^2. Although some of this is cancelled by negative terms in the cost functions, we still expect the overall cost to be closer to d^2. The situation is similar for values of d close to the FFT threshold d_1.

The dependency in t (where t is the bound on the degree of the factor base elements) is hidden in the term in $\deg(P)$. Asymptotically, as $q \to \infty$, we have $\deg(P) \in O(q^t)$. When Algorithm 2 is used with FFT multiplication, we have a term in $\deg(P)^\epsilon$. Here, the ϵ represents the logarithmic terms in the FFT complexity, so $\deg(P)^\epsilon$ is roughly $O(\log(q^t)\log\log(q^t))$, which is $O(t\log t)$ as t goes to infinity. Therefore, the dependence on t in Algorithm 2 is super-linear when FFT multiplication is used, as opposed to linear for Algorithm 1. On the other hand, if only Karatsuba multiplication is used, then large values of t will have a considerable impact on the performances of Algorithm 2 since the dependency in t is exponential.

6 Computational results

Although our analysis predicts that the batch smoothness test and the single tests will have similar performances as $d \to \infty$, it is expected to be somewhat more complicated in practice. First, our asymptotic analysis suppressed logarithmic terms and constants and hence does not adequately differentiate between the actual costs of multiplication and remainder computation. Thus, the actual runtime functions are more complicated, as well as our estimate of an optimal batch size. Secondly, implementations of polynomial arithmetic generally use more algorithms than the two assumed in our analysis. As a minimum, Karatsuba multiplication is used for some range of polynomial degrees between “schoolbook” and FFT algorithms (eg. NTL [13] switches to Karatsuba for degree greater than 16)) and some switch between other algorithms as well (eg. the GF2X library [1]). In this section, we give numerical data comparing the performance of the single-polynomial and batch smoothness tests.

6.1 Arithmetic operations

The crossover points d_0 between "schoolbook" and Karatsuba multiplication, and d_1 between Karatsuba and FFT multiplication, occur in the analysis of the batch smoothness test described in Section 5. It is interesting to know if these only have a theoretical significance or if they occur in the practical experiments that we ran. To illustrate that this is the case, we show the evolution of the run time of multiplication and division as d grows in $\mathbb{F}_{31}[x]$ and $\mathbb{F}_2[x]$. Table 1 compares the quadratic time multiplication and division (respectively denotes as Plain mul and Plain rem) to the quasi-linear time method based on FFT for polynomials in $\mathbb{F}_{31}[x]$ of degree between 100 and 90000. The implementation used is the one of the NTL library [13]. In Table 2, we compare Toom-Cook multiplication and the FFT-based multiplication for polynomials of degree between 150000 and 100000000 with the gf2x library [1]. In both cases, the timings in CPU msec for 100 operations were obtained on an Intel Xeon 1.87 GHz with 256 GB of memory and are presented in the Appendix. The crossover point for polynomials in $\mathbb{F}_{31}[x]$ is around $d = 400$ and for $d = 150000$ in $\mathbb{F}_2[x]$. Note that strictly speaking, these timings do not give the value of d_1. Indeed, we could not isolate Karatsuba multiplication in the corresponding libraries. Also, we could not run the FFT-based algorithm using $\mathbb{F}_2[x]$ for $d < 150000$ because the thresholds were hard coded. However, we can certainly hope from the timings that d_1 (and thus d_0) are within practical reach.

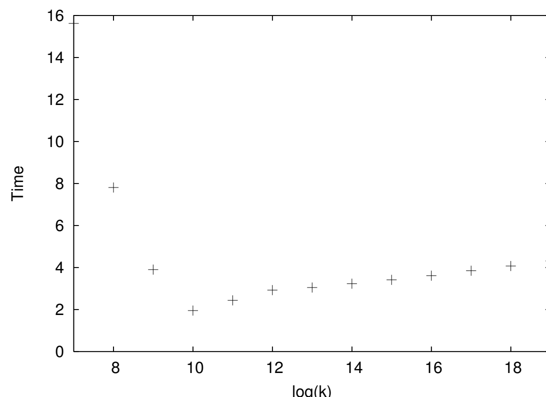
6.2 Optimal size of batch

The analysis of the batch smoothness test in Section 5 showed the existence of an optimal size of batch k of the form $k_{opt} = O\left(\frac{\deg(P)}{d}\right)$. To illustrate the impact of k on the run time, we fixed $d = 100$ and tested the t-smoothness of 100 polynomials in $\mathbb{F}_2[x]$ for $t = 25$. We ran our experiment on an Intel Xeon 1.87 GHz with 256 GB of memory. The corresponding timings are presented in the Appendix. Figure 3 shows the graph of the amortized time in CPU msec for $\log(k) = 5, \cdots, 19$. Note that we only take powers of 2 to optimize the use of the tree structure. We clearly see that there is an optimum value. On the same architecture, we ran other experiments to show the dependency of the optimal value of k on d and $\deg(P)$. Table 3 shows the optimal value of k for the test of smoothness of polynomials in $\mathbb{F}_2[x]$ of fixed degree $d = 100$ when t varies between 5 and 25. Likewise, in Table 4, we fix $t = 25$ and let the degree of the polynomials in $\mathbb{F}_2[x]$ vary between 100 and 1000. In each case, the metric to choose the optimal k is the amortized CPU time for 100 tests. Despite a few outliers, the general trend predicted by the theory seems to be respected. Table 3 shows that the optimal value of k gets larger as t (and thus $\deg(P)$) gets larger. Meanwhile, Table 4 shows that when d gets larger, the optimal k gets smaller, which is consistent with the term in $\frac{1}{d}$ predicted by the analysis. Since the analysis of Section 5 is asymptotic, and since only moderate values of d and t are within practical range, it is delicate to confirm the theory with our available data. In addition, we have a very low granularity for k (20 different values). However, the results presented in Table 3 and 4 are quite promising. Indeed, we can see for example that in Table 3 for $\deg(P) = 67100116$, $k = 131072$ while for $\deg(P) = 8384230$, $k = 16384$. As $\frac{67100116}{8384230} \approx 8 = \frac{131072}{16384}$, this is consistent with a linear dependence in $\deg(P)$. In Table 4, we have $k = 131072$ for $g = 100$ while $k = 16384$ for $g = 1000$. We have $\frac{131072}{16384} = 8$ while the dependency in $\frac{1}{d}$ predicted by the analysis suggests a ratio of 10.

6.3 Dependence on t

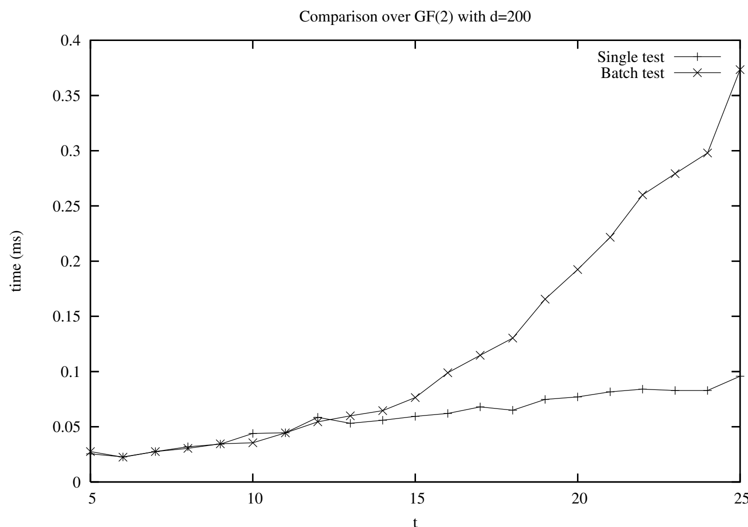
The smoothness test algorithms presented in Section 3 and Section 4 both depend on the bound t on the degree of the polynomials in the factor base. The larger t, the more expensive a smoothness test is. The expected time of Algorithm 1 has a term in $td^\theta \log(q)$ where $1 < \theta \leq 2$. As discussed in the previous section, the dependence on t in the cost of Algorithm 2 is expected to be roughly $t \log t$ when FFT multiplication is used.

Figure 3: Optimal value of k in $\mathbb{F}_2[x]$ for $t = 25$ and $d = 100$



To illustrate this, we ran experiments in $\mathbb{F}_2[x]$ and $\mathbb{F}_3[x]$ to show the impact of FFT multiplication, and in $\mathbb{F}_4[x]$ where we only have "schoolbook" and Karatsuba multiplication. We fixed the degree d of the polynomials to be tested to $d = 200$ in $\mathbb{F}_2[x]$ and $\mathbb{F}_3[x]$, and $d = 10$ in $\mathbb{F}_4[x]$. We measured the time to test the t-smoothness of 100 polynomials for t between 5 and 25 in $\mathbb{F}_2[x]$, between 5 and 15 in $\mathbb{F}_3[x]$ and between 5 and 10 in $\mathbb{F}_4[x]$. In each case, we compare the amortized cost of testing the smoothness of one polynomial using Algorithm 1 (single FFT for $\mathbb{F}_2[x]$ and $\mathbb{F}_3[x]$, single Karatsuba for $\mathbb{F}_4[x]$) and Algorithm 2 (batch test). The timings, which are displayed in Table 5 Table 6, and Table 7 available in the Appendix were obtained on a machine with 64 Intel Xeon X7560 2.27 GHz cores and 256 GB of shared RAM.

Figure 4: Dependency on t in $\mathbb{F}_2[x]$ with $d = 200$



All the timings show that the cost increases with the size of t. The analysis predicts a linear dependency in t for the single tests. We see in Table 5 that this is consistent with the timings of Algorithm 1 with FFT. For example, in $\mathbb{F}_2[x]$ for $t = 25$, the average time is 0.0957 msec while it is 0.0255 for $t = 5$. We have $\frac{0.0957}{0.0255} \approx 3.75$ while the theory predicts a ratio of 5. Likewise, in $\mathbb{F}_3[x]$, the time for $t = 15$ is 5.263 msec while it is 1.597 for $t = 5$, which is a ratio of $\frac{5.263}{1.597} \approx 3.29$ while the theory predicts a ratio of 3. The expected super-linear dependency in t of Algorithm 2

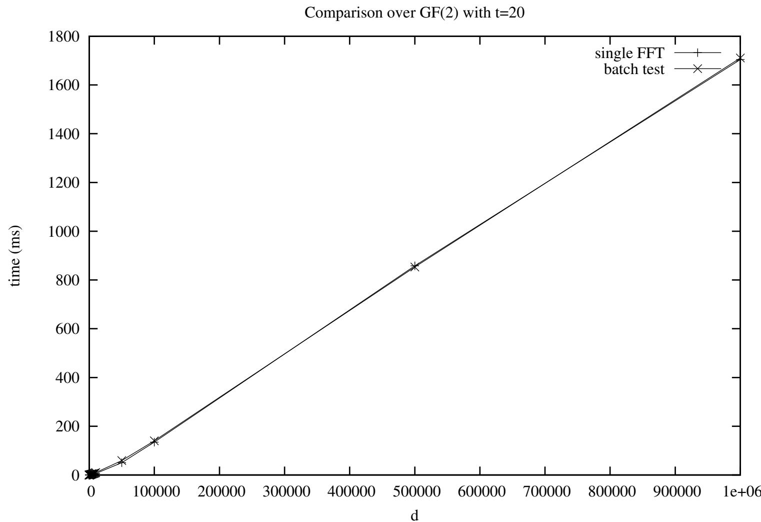
with FFT multiplication does not appear quite as clearly, but the growth does appear to be worse than linear.

Table 7 shows us the dependency in t for Algorithm 1 and Algorithm 2 in $\mathbb{F}_4[x]$. The time for Algorithm 1 with $t = 10$ is 2.437 while it is 1.396 for $t = 5$. The ratio is $\frac{2.437}{1.396} \approx 1.74$ while the theory predicts a ratio of 2. For Algorithm 2, the time with $t = 10$ is 3.867 while it is 1.428 for $t = 5$. The ratio is $\frac{3.867}{1.428} \approx 2.70$. In this case, NTL uses Kronecker substitution to perform the multiplication in $\mathbb{F}_2[x]$ using asymptotically fast arithmetic, so the expected ratio should be slightly above 2.

6.4 Dependency in d

When fast multiplication is assumed, the asymptotic complexity of both Algorithm 1 and Algorithm 2 is quasi linear in d, and when using Karatsuba multiplication, the theory predicts that it is in $d^{\log_2(3)}$. To illustrate this, we ran experiments for fixed values of t and increasing d in $\mathbb{F}_2[x]$, $\mathbb{F}_3[x]$ and $\mathbb{F}_4[x]$. We compared the performances of the single test with FFT multiplication (single FFT) and Algorithm 2 (batch test). In $\mathbb{F}_4[x]$, only Karatsuba multiplication is available for both Algorithm 1 (denoted single Karatsuba) and for Algorithm 2. The timings, which are displayed in Table 8, Table 9, Table 10, and Table 11 available in the Appendix were obtained on a machine with 64 Intel Xeon X7560 2.27 GHz cores and 256 GB of shared RAM.

Figure 5: Dependency on d for large d in $\mathbb{F}_2[x]$ for $t = 20$



We observe that Algorithm 1 and Algorithm 2 perform very similarly when using either the FFT multiplication. In addition, according to the theory, their run time seems to grow linearly with the degree once d is sufficiently large. This is shown in particular in Figure 5, in the case of $\mathbb{F}_2[x]$ where FFT arithmetic is available and we consider d up to 10^6. Our experiments for $\mathbb{F}_4$ presented in Figure 7 also show a roughly linear growth. Note that these experiments were designed to investigate the performance of the algorithm for values of d close to d_0, the Karatsuba threshold. For larger values of d, NTL switches to Kronecker substitution, using the quasi-linear FFT implementation of GF2X.

The linearity in d is less clear for $d \leq 1000$ as shown in Figure 6 in $\mathbb{F}_2[x]$ for $t = 10$. However, it is interesting to note that the variations of the cost with Algorithm 1 and Algorithm 2 are very similar. This similarity is also well illustrated by Figure 7 which shows the dependency on d in the run time of Algorithm 1 and Algorithm 2 for $d \leq 100$ in $\mathbb{F}_4[x]$ for $t = 6$. Once $d > 25$, the cost functions seem to differ by a constant, as predicted by the theory.

Figure 6: Dependency on d for small d in $\mathbb{F}_2[x]$ for $t = 10$

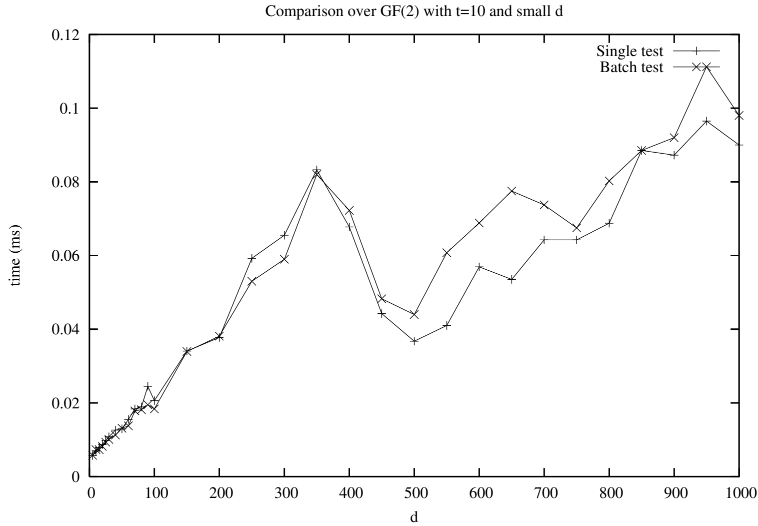
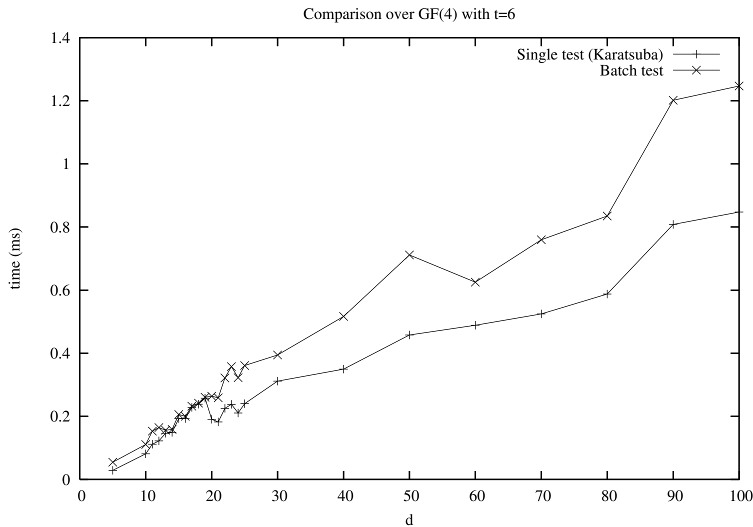


Figure 7: Dependency on d for small d in $\mathbb{F}_4[x]$ for $t = 6$



For small values of d near the thresholds d_0 and d_1, we also see both algorithms exhibiting roughly the same performance. Recall that one motivation for considering the batch algorithm in this context is that it can take advantage of asymptotically faster arithmetic in cases when the single polynomial algorithm is forced to use schoolbook arithmetic. Unfortunately we did not observe a dramatic improvement even in this scenario. For example, in Figure 7 ($\mathbb{F}_4[x]$) we notice that the two algorithms have roughly the same performance when d is close to 16, NTL's threshold for swithching from schoolbook arithmetic to Karatsuba in this case.

In all our timings showing the dependency in d for fixed t, Algorithm 1 performs better (by a constant factor) than Algorithm 2 at the notable exception of Table 10 which shows the run time in $\mathbb{F}_3[x]$ for fixed $t = 5$. There, the batch smoothness test seems to provide a mild speed-up by a constant factor. This may be explained by the conjunction of a small value of t (and thus

of $\deg(P)$) and by lower thresholds for the value of d_1 where the fast multiplication becomes competitive (which occurs in the theoretical prediction).

6.5 Examples of practical relevance

Our first example is the curve C155 from [14], a genus 31 hyperelliptic curve defined over $\mathbb{F}_{2^5}$. This curve is the result of the Weil descent on an elliptic curve over $\mathbb{F}_{2^{155}}$ as shown in [8]. For this example, a smoothness bound of 4 is used and the polynomials to be factored have degree 36. The optimal size of batch is $k = 32768$. The average times for testing the smoothness of degree-36 polynomials are

- 0.0005884 CPU sec with single test and fast multiplication,
- 0.0012091 CPU sec with the batch test.

The timings were obtained on a machine with 64 Intel Xeon X7560 2.27 GHz cores and 256 GB of shared RAM. The implementation of polynomial arithmetic in $\mathbb{F}_{2^5}[X]$ we used only has school-book and Karatsuba remainder algorithms available (no FFT) — we expect somewhat better performance of the batch method if FFT were added to the implementation.

Our second example is taken from the discrete logarithm computation in $\mathbb{F}_{2^{1039}}$ described in [6]. Here, we assume a smoothness bound of 25 and that the polynomials in $\mathbb{F}_2[x]$ in the cofactorization step have degree 99. The factor base has 2807196 primes and the degree of their product is 67100116. The optimal batch size is $k = 131072$. The average times for testing the smoothness of degree-99 polynomials are

- 0.00007016 CPU sec with single test and fast multiplication,
- 0.00019327 CPU sec with the batch test.

The timings were obtained on a machine with 64 Intel Xeon X7560 2.27 GHz cores and 256 GB of shared RAM. This computation makes use of the GF2X library directly, which does include optimized polynomial arithmetic for large degree operands.

7 Conclusion

Our theoretical analysis and numerical experiments show that the batch smoothness test does in general not out-perform the simpler, more memory-friendly single polynomial test. The theoretical analysis shows that if FFT multiplication is available, both methods have the same asymptotic quasi-linear complexity with respect to the degree d of the polynomials to be tested. As a function of the smoothness bound, the batch method has worse asymptotic complexity, namely super-linear as opposed to linear. In most practical cases, for sufficiently large d the behavior of the two methods only differs by a constant, thus backing up the theory. The single smoothness test is more efficient in almost all cases. The two factors that can make the batch smoothness test faster than single tests are a low smoothness bound and a low threshold on the degree for which FFT multiplication becomes fast, as we can see in Table 10.

References

[1] *gf2x*, a C/C++ software package containing routines for fast arithmetic in GF(2)[x] (multiplication, squaring, GCD) and searching for irreducible/primitive trinomials. Available at http://gf2x.gforge.inria.fr/.

[2] D. Bernstein, *How to find smooth parts of integers*, submited to *Mathematics of Computation.*

[3] J.-F. Biasse and M. Jacobson, *Practical improvements to class group and regulator computation of real quadratic fields*, Algorithmic Number Theory - ANTS-IX (Nancy, France) (G. Hanrot, F. Morain, and E. Thomé, eds.), Lecture Notes in Computer Science, vol. 6197, Springer-Verlag, 2010, pp. 50–65.

[4] G. Bisson and A. Sutherland, *Computing the endomorphism ring of an ordinary elliptic curve over a finite field*, Journal of Number Theory **113** (2011), 815–831.

[5] D. Coppersmith, *Fast evaluation of logarithms in fields of characteristic two*, IEEE Transactions on Information Theory **30** (1984), no. 4, 587–594.

[6] J. Detrey, P. Gaudry, and M. Videau, *Relation collection for the function field sieve*, ARITH 21 - 21st IEEE International Symposium on Computer Arithmetic (Austin, Texas, United States) (A. Nannarelli, P.-M. Seidel, and P. Tang, eds.), IEEE, 2013, pp. 201–210.

[7] A. Enge and P. Gaudry, *A general framework for subexponential discrete logarithm algorithms*, Acta Arithmetica **102** (2002), 83–103.

[8] M. Jacobson, A. Menezes, and A. Stein, *Solving elliptic curve discrete logarithm problems using weil descent*, Journal of the Ramanujan Mathematical Society **16** (2001), 231–260.

[9] H. Lenstra, *Factoring integers with elliptic curves*, The Annals of Mathematics **126** (1987), no. 3, 649–673.

[10] R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*, Cambridge University Press, New York, NY, USA, 1986.

[11] A. Odlyzko, *Discrete logarithms in finite fields and their cryptographic significance*, Proc. Of the EUROCRYPT 84 Workshop on Advances in Cryptology: Theory and Application of Cryptographic Techniques (New York, NY, USA), Springer-Verlag New York, Inc., 1985, pp. 224–314.

[12] A. Schnhage and V. Strassen, *Schnelle multiplikation groer zahlen*, Computing **7** (1971), no. 3-4, 281–292 (German).

[13] V. Shoup, *NTL: A Library for doing Number Theory*, Software, 2010, http://www.shoup.net/ntl.

[14] M. Velichka, M. Jacobson, and A. Stein, *Computing discrete logarithms in the jacobian of high-genus hyperelliptic curves over even characteristic finite fields*, 2013, To appear in Mathematics of Computation.

A Timings

In this appendix, we present the timings that were used to illustrate the theoretical predictions on the run time of Algorithm 1 and Algorithm 2, as well as for the comparison of their performance. Table 1, Table 2, Table 3 and Table 4 were obtained on an Intel Xeon 1.87 GHz with 256 GB of memory while the rest of the timings was obtained on a machine with 64 Intel Xeon X7560 2.27 GHz cores and 256 GB of shared RAM.

Table 1: Arithmetic operations in $\mathbb{F}_{31}$ with respect to the degree d

d	Plain mul	FFT mul	Plain rem	FFT rem
100	10	30	30	60
200	40	50	120	160
300	70	100	280	330
400	110	100	490	340
500	160	110	760	360
600	230	200	1100	710
700	270	200	1500	720
800	340	210	1940	730
900	400	220	2480	750
1000	480	220	3060	760
2000	1470	470	12200	1620
3000	2760	920	26690	3320
4000	4340	980	47570	3450
5000	6610	1860	74160	6880
6000	8210	1920	109350	7260
7000	10340	2000	149180	7370
8000	13130	2050	195620	7480
9000	16820	3920	246860	14870
10000	19900	3900	304000	15200
20000	59800	8400	1176800	30800
30000	100200	8700	2606200	27400
40000	178900	17400	4713200	56200
50000	244600	18400	7702100	66200
60000	302700	18500	10073400	57600
70000	431900	34800	14829300	134100
80000	532000	35700	19094700	137900
90000	600400	37100	24063600	141400

Table 2: Multiplication in $\mathbb{F}_2[x]$ with respect to the degree d

d	Toom-Cook mul	FFT mul
150000	470	430
250000	950	690
500000	2430	1490
1000000	6650	3250
5000000	63840	23200
10000000	183330	55030
50000000	1595270	314470
100000000	4110380	731010

Table 3: Influence of t on the optimal value of k for $d = 100$

t	$\deg(P)$	k
5	52	1024
6	106	1024
7	232	1024
8	472	1024
9	976	1024
10	1966	1024
11	4012	2048
12	8032	1024
13	16222	2048
14	32476	4096
15	65206	1024
16	130486	2048
17	261556	2048
18	523132	4096
19	1047418	8192
20	2094958	16384
21	4191976	32768
22	8384230	16384
23	16772836	131072
24	33545716	262144
25	67100116	131072

Table 4: Influence of d on the optimal value of k for $t = 25$

d	k
100	131072
200	65536
300	131072
400	32768
500	65536
600	65536
700	16384
800	16384
900	32768
1000	16384

Table 5: Influence of t on the run time in $\mathbb{F}_2[x]$ for $d = 200$

t	single FFT	batch test
5	0.0255000	0.0275000
6	0.0227500	0.0225000
7	0.0275000	0.0275000
8	0.0318750	0.0303750
9	0.0342500	0.0345000
10	0.0440000	0.0355000
11	0.0446250	0.0445000
12	0.0585625	0.0544375
13	0.0531563	0.0599375
14	0.0558750	0.0645625
15	0.0593750	0.0764688
16	0.0621172	0.0988672
17	0.0679414	0.1147110
18	0.0649724	0.1303620
19	0.0747079	0.1656130
20	0.0770960	0.1923600
21	0.0816949	0.2216070
22	0.0840844	0.2599670
23	0.0828506	0.2791900
24	0.0828362	0.2979370
25	0.0957900	0.3734650

Table 6: Influence of t on the run time in $\mathbb{F}_3[x]$ for $d = 200$

t	single FFT	batch test
5	1.59750	1.51150
6	1.99750	2.02450
7	2.41375	2.43750
8	2.52275	2.85962
9	3.19660	3.82236
10	3.44301	4.47033
11	3.82687	5.39385
12	4.00995	6.07482
13	4.56101	7.25818
14	4.84185	8.53217
15	5.26337	11.35560

Table 7: Influence of t on the run time in $\mathbb{F}_4[x]$ for $d = 10$

t	single Karatsuba	batch test
5	1.39675	1.42875
6	1.63213	1.80631
7	1.92971	2.21272
8	2.11232	2.68086
9	2.40723	3.20971
10	2.43713	3.86726

Table 8: Influence of d on the run time in $\mathbb{F}_2[x]$ for $t = 10$

d	single FFT	batch test
5	0.00567969	0.00577148
10	0.00699061	0.00732203
15	0.00784766	0.00731641
20	0.00868750	0.00817188
25	0.00984375	0.00934375
30	0.01081250	0.01006250
40	0.01262500	0.01125000
50	0.01297560	0.01312110
60	0.01550000	0.01375000
70	0.01837500	0.01781250
80	0.01890620	0.01815620
90	0.02450000	0.01950000
100	0.02062500	0.01837500
150	0.03406250	0.03393750
200	0.03775000	0.03812500
250	0.05925000	0.05300000
300	0.06550000	0.05900000
350	0.08325000	0.08200000
400	0.06775000	0.07225000
450	0.04425000	0.04825000
500	0.03675000	0.04400000
550	0.04100000	0.06075000
600	0.05690620	0.06884370
650	0.05350000	0.07750000
700	0.06425000	0.07375000
750	0.06425000	0.06750000
800	0.06875000	0.08025000
850	0.08850000	0.08850000
900	0.08725000	0.09200000
950	0.09650000	0.11125000
1000	0.09000000	0.09800000

Table 9: Influence of large d on the run time in $\mathbb{F}_2[x]$ for $t = 20$

d	single FFT	batch test
1000	0.16896600	0.80926700
1500	0.34201700	1.32210000
2000	0.43851000	1.71919000
2500	0.65992600	2.07696000
3000	0.99763800	2.73519000
4000	1.33407000	3.30458000
5000	1.91657000	4.23340000
6000	2.71314000	5.71678000
7000	3.26151000	6.24501000
8000	3.85816000	7.04251000
9000	4.83971000	8.08675000
10000	6.14594000	9.65036000
50000	50.58590000	58.94140000
100000	134.76200000	140.34500000

Table 10: Influence of d on the run time in $\mathbb{F}_3[x]$ for $t = 5$

d	single FFT	batch test
5	0.00948437	0.0147812
10	0.02300000	0.0267500
15	0.03675000	0.0427500
20	0.05100000	0.0572500
25	0.07250000	0.0807500
30	0.10275000	0.1112500
40	0.14700000	0.1472500
50	0.31050000	0.2767500
60	0.37475000	0.3435000
70	0.60350000	0.5162500
80	0.59375000	0.5267500
90	0.70975000	0.6420000
100	0.64875000	0.6045000
150	1.24425000	1.1432500
200	1.34800000	1.2955000
250	2.03675000	1.9017500
300	3.50200000	3.1397500
350	3.88800000	3.6395000
400	3.51150000	3.2830000
450	3.76100000	3.4872500
500	4.28825000	4.0065000
550	7.23575000	6.8602500
600	7.38875000	7.0150000
650	7.47050000	6.8910000

Table 11: Influence of d on the run time in $\mathbb{F}_4[x]$ for $t = 6$

d	single Karatsuba	batch test
5	0.0286934	0.0544766
10	0.0810391	0.1108360
11	0.1116380	0.1528570
12	0.1221730	0.1645870
13	0.1462990	0.1558240
14	0.1493440	0.1583590
15	0.1928120	0.2057660
16	0.1927390	0.2000050
17	0.2282190	0.2318750
18	0.2385310	0.2418750
19	0.2575210	0.2603760
20	0.1904380	0.2635780
21	0.1823950	0.2587520
22	0.2251990	0.3215900
23	0.2377760	0.3575710
24	0.2104650	0.3225120
25	0.2399790	0.3608470
30	0.3114220	0.3942500
40	0.3496950	0.5163750
50	0.4579820	0.7115210
60	0.4888120	0.6253440
70	0.5246250	0.7597500
80	0.5874060	0.8348750
90	0.8080250	1.2016500
100	0.8473290	1.2471700
200	1.6273100	1.8025000
300	2.5498800	2.8096300
400	3.1251300	3.3780000
500	4.2640000	4.4917500
600	5.0027500	5.3097500
700	5.5476900	6.0990300
800	5.8550000	6.0532500
900	7.3746500	7.5213100
1000	9.2463700	9.3040600
5000	43.8528000	43.7536000
10000	85.3161000	86.2299000
50000	467.7270000	489.9680000
100000	986.6330000	1005.4700000