Proc. ISSAC 96, Zürich, 1996, pp. 1-9

Arithmetic and factorization of polynomials over $\mathbb{F}_2$

Joachim von zur Gathen and Jürgen Gerhard
Fachbereich 17 Mathematik-Informatik
Universität-GH Paderborn
D-33095 Paderborn, Germany
e-mail: {gathen,jngerhar}@uni-paderborn.de

Extended Abstract

Abstract

We describe algorithms for polynomial multiplication and polynomial factorization over the binary field $\mathbb{F}_2$ and their implementation. They allow polynomials of degree up to 100,000 to be factored in about one day of CPU time.

1 Introduction

The problem of polynomial factorization over the binary field $\mathbb{F}_2$ is, given a monic polynomial $f \in \mathbb{F}_2[x]$, to compute the factorization $f = f_1^{e_1} \cdots f_r^{e_r}$ with monic irreducible pairwise distinct polynomials $f_1, \ldots, f_r \in \mathbb{F}_2[x]$ and positive $e_1, \ldots, e_r \in \mathbb{N}$. The efficiency of the currently known algorithms for this problem relies on fast polynomial arithmetic, in particular, on fast polynomial multiplication.

The multiplication method of Karatsuba & Ofman (1962) has an asymptotic running time of $O(n^{1.59})$ operations in $\mathbb{F}_2$ for polynomials of degree less than n, which is better than the $O(n^2)$ bound for the naïve multiplication algorithm, but still too slow in practice for large n. Schönhage (1977) gives an $O(n \log n \operatorname{loglog} n)$ algorithm based on a ternary FFT with roots of unity of 3-power order. Reischert (1995) implemented several algorithms for polynomial multiplication over $\mathbb{F}_2$, including those by Karatsuba & Ofman, Schönhage, and Cantor (see below). Shoup (1995) has successfully implemented a fast FFT-based algorithm for multiplying polynomials over $\mathbb{F}_p$ for a prime p, using a modular approach, but it seems to be practical only when p is not too small.

Cantor (1989) presented an algorithm for multiplying polynomials of degree less than n over $\mathbb{F}_p$ for a prime p that uses $O(mp^m)$ multiplications and $O(m^2p^m)$ scalar operations (i.e., additions or multiplications by elements of $\mathbb{F}_p$) in the field $\mathbb{F}_{p^m}$, where m is the least power of p with $4n \leq mp^m$. It behaves particularly well in the case $p = 2$. In contrast to the FFT-based algorithms cited above, which evaluate and interpolate at suitably subgroups of the *multiplicative* group of a (finite) field, Cantor's approach uses *additive* subgroups, i.e., $\mathbb{F}_p$-linear subspaces of $\mathbb{F}_{p^m}$. Montgomery (1991) implemented a polynomial factorization algorithm that uses Cantor's algorithm as a subroutine, and was able to factor a sparse polynomial of degree more than 200,000 over $\mathbb{F}_2$ in about 45 hours.

In section 2, we generalize Cantor's method to work for prime powers p and *arbitrary* m. We state explicit "O"-free upper bounds on the time and space cost of our algorithms.

In section 7, we report on experiments in which our implementation of Cantor's original method turned out to be superior to these new variants.

In the last 5 years, dramatic progress in the area of polynomial factorization has been made, both in theory and in practice. The classical algorithms for polynomials over finite fields are due to Berlekamp (1967, 1970), Cantor & Zassenhaus (1981), and Ben-Or (1981). Recently, many variants and asymptotically faster algorithms have been proposed by von zur Gathen & Shoup (1992), Kaltofen (1992), Niederreiter (1994), Gao & von zur Gathen (1994), Kaltofen & Lobo (1994), and Kaltofen & Shoup (1995). Implementations are described in Kaltofen & Lobo (1994), Shoup (1995), and Fleischmann & Roelse (1995).

Section 3 gives an outline of the structure of some modern polynomial factorization algorithms. In sections 4 and 5, we discuss and analyze a new variant of the distinct degree factorization stage in those algorithms, using interval partitions with polynomially growing interval sizes. In section 6, we indicate how the distinct degree factorization stage over $\mathbb{F}_2$ can be further speeded up by the use of an irreducibility test. Finally, an implementation of the polynomial factorization algorithm over $\mathbb{F}_2$ is described in section 7, including examples of running times.

We have mainly concentrated on optimizing our implementation for the distinct degree factorization stage. Of course, more work is required to also optimize for cases where the input is known to be special, say when we factor trinomials or cyclotomic polynomials. In particular, we have not optimized the equal degree factorization stage of our software.

Due to a page limit in these proceedings, we had to omit many details and proofs. They can be found in von zur Gathen & Gerhard (1996).

2 Fast polynomial multiplication over $\mathbb{F}_q$

Let $\mathbb{F}_q$ be a finite field with q elements and m a positive integer. The extension field $\mathbb{F}_{q^m}$ is an m-dimensional vector space over $\mathbb{F}_q$. Suppose that $W \subseteq \mathbb{F}_{q^m}$ is a fixed k-dimensional subspace, where $1 \leq k \leq m$, and that we want to solve the following problems.

Problem 2.1 *(Multipoint evaluation) Given $f \in \mathbb{F}_{q^m}[x]$ of degree less than q^k, compute $f(\alpha)$ for all $\alpha \in W$.*

Problem 2.2 *(Interpolation) Given a map $\gamma : W \longrightarrow \mathbb{F}_{q^m}$, compute the unique polynomial $f \in \mathbb{F}_{q^m}[x]$ of degree less than q^k satisfying $f(\alpha) = \gamma(\alpha)$ for all $\alpha \in W$.*

The algorithms presented below for these two problems admit a natural parallelization, but here we only discuss the sequential versions.

We fix a basis $(\beta_1, \ldots, \beta_k)$ of W over $\mathbb{F}_q$, and for $0 \leq i \leq k$ let W_i be the subspace $W_i = \operatorname{span}\{\beta_1, \ldots, \beta_i\} \subseteq W$ of dimension i. The sets W_i form a strictly ascending chain

$$\{0\} = W_0 \subsetneq W_1 \subsetneq \cdots \subsetneq W_{k-1} \subsetneq W_k = W, \qquad (1)$$

and for $1 \leq i \leq k$ we have the recursive decomposition

$$W_i = \bigcup_{c \in \mathbb{F}_q} (c\beta_i + W_{i-1})$$

of W_i into q pairwise disjoint cosets, which generalizes to

$$\beta + W_i = \bigcup_{c \in \mathbb{F}_q} (\beta + c\beta_i + W_{i-1}) \qquad (2)$$

for arbitrary $\beta \in \mathbb{F}_{q^m}$.

Next, we define the sequence of polynomials $s_i \in \mathbb{F}_{q^m}[x]$ for $0 \leq i \leq k$ by

$$s_i = \prod_{\alpha \in W_i} (x - \alpha).$$

Obviously, s_i is a monic squarefree polynomial of degree q^i, and corresponding to (1), we have

$$x = s_0 \mid s_1 \mid \cdots \mid s_{k-1} \mid s_k.$$

Lemma 2.3 *The following hold for $0 \leq i \leq k$.*

(i) The recursion formulae

$$s_i = \prod_{c \in \mathbb{F}_q} (s_{i-1} - cs_{i-1}(\beta_i)) = s_{i-1}^q - s_{i-1}(\beta_i)^q \cdot s_{i-1}$$

hold if $i \geq 1$.

(ii) s_i is an $\mathbb{F}_q$-linearized polynomial, i.e., $s_i(f + g) = s_i(f) + s_i(g)$ and $s_i(cf) = cs_i(f)$ for all $f, g \in \mathbb{F}_{q^m}[x]$ and $c \in \mathbb{F}_q$.

(iii) $s_i - s_i(\beta) = \prod_{c \in \mathbb{F}_q} (s_{i-1} - s_{i-1}(\beta + c\beta_i)) = \prod_{\alpha \in \beta + W_i} (x - \alpha)$ *for all $\beta \in \mathbb{F}_{q^m}$.*

Note that statement (iii) of the lemma reduces to statement (i) and the definition of the s_i, respectively, if $\beta = 0$.

The decomposition (2) and statement (iii) of the lemma suggest the following algorithm for Problem 2.1.

Algorithm 2.4 *(Multipoint evaluation)*
We assume that the polynomials $s_i \in \mathbb{F}_{q^m}[x]$ for $0 \leq i \leq k$ as defined above and the values $s_i(\beta_j)$ for $0 \leq i < j \leq k$ are precomputed and stored.
Input: $i \in \mathbb{N}$ with $0 \leq i \leq k$, $f \in \mathbb{F}_{q^m}[x]$ of degree less than q^i, and $c_{i+1}, \ldots, c_k \in \mathbb{F}_q$.
Output: The values $f(\alpha)$ for all $\alpha \in \beta + W_i$, where $\beta = \sum_{i<j\leq k} c_j\beta_j \in W$.

1. *If $i = 0$ then return f.*
2. *Compute $s_{i-1}(\beta) = \sum_{i<j\leq k} c_j s_{i-1}(\beta_j)$, and for each $c \in \mathbb{F}_q$ divide f by $s_{i-1} - s_{i-1}(\beta) - cs_{i-1}(\beta_i)$ with remainder, i.e., compute polynomials $g_c, r_c \in \mathbb{F}_{q^m}[x]$ with*
$$f = g_c(s_{i-1} - s_{i-1}(\beta) - cs_{i-1}(\beta_i)), \ \deg r_c < q^{i-1}.$$
3. *For each $c \in \mathbb{F}_q$, recursively call the algorithm with input $i-1$, r_c and $c, c_{i+1}, \ldots, c_k$ to get $\gamma(\alpha) = r_c(\alpha)$ for all $\alpha \in (\beta + c\beta_i) + W_{i-1}$.*
4. *Return $\gamma(\alpha)$ for $\alpha \in \beta + W_i$.*

Note that $s_{i-1}(\beta) + cs_{i-1}(\beta_i) = s_{i-1}(\beta + c\beta_i)$, by the linearity of s_{i-1}.

Notation. We call an addition in $\mathbb{F}_{q^m}$ or a multiplication of an element of $\mathbb{F}_{q^m}$ by an element of $\mathbb{F}_q$ a *scalar operation*, while a *multiplication* is a multiplication of two arbitrary elements of $\mathbb{F}_{q^m}$. When analyzing the space requirements of an algorithm, we only count the *work space*, i.e., the number of elements of $\mathbb{F}_{q^m}$ for which space is used in addition to the space for input, output, and precomputed objects.

Theorem 2.5 *Algorithm 2.4 works correctly and for $i = k$ uses at most $\frac{q-1}{2}k^2q^k + \frac{q+7}{2}kq^k$ scalar operations and $\frac{q-1}{2}k^2q^k + \frac{q-1}{2}kq^k$ multiplications in $\mathbb{F}_{q^m}$. Furthermore, the algorithm uses work space for at most $\frac{q}{q-1}q^k + 3k$ elements of $\mathbb{F}_{q^m}$.*

For the interpolation, we have the following algorithm.

Algorithm 2.6 *(Interpolation)*
We assume that the polynomials $s_i \in \mathbb{F}_{q^m}[x]$ for $0 \leq i \leq k$ as defined above and the values $s_i(\beta_j)$ and $-s_i(\beta_{i+1})^{1-q}$ for $0 \leq i < j \leq k$ are precomputed and stored.
Input: $i \in \mathbb{N}$ with $0 \leq i \leq k$, $c_{i+1}, \ldots, c_k \in \mathbb{F}_q$, and an array of values $\gamma(\alpha) \in \mathbb{F}_{q^m}$, one for each point $\alpha \in \beta + W_i$, where $\beta = \sum_{i<j\leq k} c_j\beta_j \in W$.
Output: The interpolating polynomial $f \in \mathbb{F}_{q^m}[x]$ of degree less than q^i with $f(\alpha) = \gamma(\alpha)$ for all $\alpha \in \beta + W_i$, and the value $s_i(\beta)$.

1. *If $i = 0$ then return $\gamma(\beta)$ and β.*
2. *For each $c \in \mathbb{F}_q$, recursively call the algorithm with input $i-1$, $c, c_{i+1}, \ldots, c_k$, and $\gamma(\alpha)$ for $\alpha \in (\beta + c\beta_i) + W_{i-1}$ to get $r_c \in \mathbb{F}_{q^m}[x]$ of degree less than q^{i-1} with $r_c(\alpha) = \gamma(\alpha)$ for all $\alpha \in (\beta + c\beta_i) + W_{i-1}$, and $s_{i-1}(\beta + c\beta_i)$.*
3. *Compute $s_i(\beta) = \sum_{i<j\leq k} c_j s_i(\beta_j)$, and set*
$$f = -s_{i-1}(\beta_i)^{1-q} \sum_{c \in \mathbb{F}_q} r_c \frac{s_i - s_i(\beta)}{s_{i-1} - s_{i-1}(\beta + c\beta_i)}. \qquad (3)$$
4. *Return f and $s_i(\beta)$.*

Theorem 2.7 *Algorithm 2.6 works correctly and for $i = k$ uses no more than $\frac{q+1}{2}k^2q^k + \frac{q+5}{2}kq^k$ multiplications and $\frac{q+1}{2}k^2q^k + \frac{3q+5}{2}kq^k$ scalar operations in $\mathbb{F}_{q^m}$. Furthermore, it uses work space for at most $\frac{q+1}{q-1}q^k + 3k$ elements of $\mathbb{F}_{q^m}$.*

Theorem 2.8 *The cost of the precomputation stage for Algorithm 2.4 and 2.6 is at most $\frac{1}{6}k^3 + (2\lceil \log_2 q \rceil - \frac{3}{2})k^2 - \frac{2}{3}k$ multiplications, k inversions, and $\frac{1}{6}k^3 + \frac{1}{2}k^2 + \frac{1}{3}k$ scalar operations in $\mathbb{F}_{q^m}$. The space occupied by the precomputed objects is $k^2 + 2k$ memory locations for elements of $\mathbb{F}_{q^m}$.*

Cantor (1989) considers the special case where $q = p$ is prime, $k = m = p^d$ is a power of the characteristic, and $W_i = \ker \varphi^i$ for the $\mathbb{F}_p$-endomorphism $\varphi: \alpha \mapsto \alpha^p - \alpha$ of $\mathbb{F}_{p^{p^d}}$. He gives a basis $(\beta_1, \ldots, \beta_{p^d})$ of $W = W_k = \mathbb{F}_{p^{p^d}}$ over $\mathbb{F}_p$ such that the polynomials s_i are in $\mathbb{F}_p[x]$ for $0 \le i \le p^d$. This reduces the number of multiplications in $\mathbb{F}_{p^m}$ to $O(kq^k)$.

Theorem 2.9 *Two polynomials $a, b \in \mathbb{F}_{q^m}[x]$ whose product has degree less than n, where $q \le n \le q^m$, can be multiplied using less than*

$$\frac{3q^2 - q}{2} n \log_q^2 n + \frac{15q^2 + 7q}{2} n \log_q n$$

multiplications and

$$\frac{3q^2 - q}{2} n \log_q^2 n + \frac{19q^2 + 35q}{2} n \log_q n$$

scalar operations in $\mathbb{F}_{q^m}$, and work space for less than $5qn + 6 \log_q n$ elements of $\mathbb{F}_{q^m}$.

Using classical arithmetic in $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$, two polynomials over the field $\mathbb{F}_q$ whose product has degree less than $n \le \frac{mq^m}{2}$ can be multiplied with at most

$$(6q-2)m^4 q^m + O(m^3 q^m) \le (12q^2 - 4q) n \log_q^3 n + O(n \log_q^2 n)$$

operations in $\mathbb{F}_q$, provided that $q < n$. The bound can be improved to $O(n \log_q^2 n (\log_q \log_q n)^3)$ with a recursive approach.

Several simplifications are possible for $q = 2$, some of which are in Cantor (1989). They lead to the following result.

Theorem 2.10 *Two polynomials $a, b \in \mathbb{F}_{2^m}[x]$ whose product has degree less than $n \le 2^m$ can be multiplied using less than*

$$\frac{3}{2} n \log_2^2 n + \frac{15}{2} n \log_2 n + 8n$$

multiplications and

$$\frac{3}{2} n \log_2^2 n + \frac{29}{2} n \log_2 n + 4n + 9$$

scalar operations in $\mathbb{F}_{2^m}$. In doing so, work space for less than $2 \log_2 n + 2$ elements of $\mathbb{F}_{2^m}$ is used.

3 Polynomial factorization

Many of the modern polynomial factorization algorithms over finite fields (Cantor & Zassenhaus (1981), Ben-Or (1981), von zur Gathen & Shoup (1992), Kaltofen & Shoup (1995), but not those of Berlekamp (1967, 1970), Gao & von zur Gathen (1994), Kaltofen & Lobo (1994), Niederreiter (1994), and Kaltofen & Shoup (1995), Algorithm B) proceed in three stages:

1. **Squarefree factorization (SFF).** Given a monic polynomial $f \in \mathbb{F}_q[x]$ of degree n, compute the unique monic squarefree and pairwise coprime polynomials $g_1, \ldots, g_n \in \mathbb{F}_q[x]$ such that
$$f = \prod_{1 \le i \le n} (g_i)^i.$$
2. **Distinct degree factorization (DDF).** Given a monic squarefree polynomial $f \in \mathbb{F}_q[x]$ of degree n, compute its unique decomposition
$$f = \prod_{1 \le d \le n} h_d \tag{4}$$
into monic polynomials $h_1, \ldots, h_n \in \mathbb{F}_q[x]$ such that each h_d has only irreducible factors of degree d. Such an h_d is called an *equal-degree polynomial of order d*.
3. **Equal degree factorization (EDF).** Given integers $d, r \in \mathbb{N}$ with $r \ge 2$ and a squarefree equal-degree polynomial $f \in \mathbb{F}_q[x]$ of order d and degree $n = rd$, compute its r irreducible factors.

In this and the following sections, $\mathsf{M}(n)$ denotes the multiplication time for polynomials over $\mathbb{F}_q$, i.e., two polynomials of degree less than n can be multiplied with $O(\mathsf{M}(n))$ operations in $\mathbb{F}_q$. Using the approach in section 2, we may e.g. take $\mathsf{M}(n) = n(\log n)^2(\operatorname{loglog} n)^3$. We also use here that a division with remainder and a gcd for polynomials of degree at most n can be computed using $O(\mathsf{M}(n))$ and $O(\mathsf{M}(n) \log n)$ operations in $\mathbb{F}_q$, respectively (see, e.g., Aho *et al.* 1974, Chapter 8).

Using the deterministic algorithm of Yun (1976), stage 1 can be performed at essentially the cost of one gcd, i.e., $O(\mathsf{M}(n) \log n)$ operations in $\mathbb{F}_q$. The asymptotically fastest of the currently known algorithms for stage 2 when the field size q is fixed is due to Kaltofen & Shoup (1995) and uses $O(n^{1.815} \log q)$ operations in $\mathbb{F}_q$. Finally, stage 3 can be performed with $O(n^{1.7} + \mathsf{M}(n) \log r \log q)$ operations in $\mathbb{F}_q$, using a probabilistic algorithm of von zur Gathen & Shoup (1992).

If one factors random polynomials, the dominating cost of the overall algorithm is the cost of the DDF. The reason is that probably an EDF has to be performed only on equal-degree polynomials of small degree (see Flajolet *et al.* (1996) for a detailed analysis). This is confirmed by tests of our factorization routine on random inputs, as reported in section 7.

4 Distinct degree factorization

In the following, we briefly discuss the basic idea of all DDF algorithms over the finite field $\mathbb{F}_q$. We use the following cost measures for our algorithms:

- $\mathsf{P}(n)$, the cost for computing one *modular product* of two polynomials of degree less than n modulo a polynomial of degree at most n. Thus $\mathsf{P}(n) \in O(\mathsf{M}(n))$.
- $\mathsf{Q}(n)$, the cost for one *modular qth power* of a polynomial of degree less than n modulo a polynomial of degree at most n. For large q, we may assume that $\mathsf{Q}(n) \le 2\lfloor \log_2 q \rfloor \mathsf{P}(n)$, while for $q = 2$, modular squaring is cheaper than a general modular multiplication.
- $\mathsf{R}(n)$, the cost for one *remainder computation* of a polynomial of degree at most n modulo a polynomial of degree less than n.
- $\mathsf{D}(n)$, the cost for one *exact division*, i.e., a division where the remainder is known to be zero, of two polynomials of degree at most n.
- $\mathsf{G}(n)$, the cost for one *gcd computation* of two polynomials of degree at most n.

All of the above functions count operations in $\mathbb{F}_q$. This high-level cost analysis is convenient to achieve simultaneously two goals: explicit "O"-free estimates, at least for the dominant term, for various algorithms, and also good asymptotic bounds.

Let $f \in \mathbb{F}_q[x]$ be a monic squarefree polynomial of degree n. The following well-known algorithm computes the DDF (4) of f. For polynomials $a, b \in \mathbb{F}_q[x]$ with $b \neq 0$, we denote the remainder of a modulo b of degree less than $\deg b$ by $a \operatorname{rem} b$.

Algorithm 4.1
Input: A monic squarefree polynomial $f \in \mathbb{F}_q[x]$ of degree n.
Output: The polynomials $h_1, \ldots, h_n \in \mathbb{F}_q[x]$ as in (4).

1. *Set $a_0 = x$, $b_0 = f$, and $m = \lfloor \frac{n}{2} \rfloor$.*
2. *Repeat steps 3 to 5 for $i = 1, \ldots, m$.*
3. $\quad a_i = a_{i-1}^q \operatorname{rem} b_{i-1}$.
4. $\quad h_i = \gcd(a_i - x, b_{i-1})$.
5. $\quad b_i = \dfrac{b_{i-1}}{h_i}$.
6. *Set $h_i = 1$ for $m < i \leq n$. If $b_m \neq 1$, set $h_{\deg b_m} = b_m$.*
7. *Return $h_1, \ldots, h_n$.*

The cost of the algorithm is

$$m(\mathsf{Q}(n) + \mathsf{D}(n) + \mathsf{G}(n)) \in O(n \cdot \mathsf{M}(n)(\log q + \log n))$$

operations in $\mathbb{F}_q$. One drawback of the algorithm is that most of the gcds computed will equal 1.

To overcome this problem, some polynomial factorization algorithms (von zur Gathen & Shoup 1992, Kaltofen & Shoup 1995) use a "blocking strategy": the range for the degrees of possible nontrivial factors of f excepting the one of largest degree is partitioned into intervals $\mathbf{I}_1, \ldots, \mathbf{I}_k$, and there is one gcd computation per interval $\mathbf{I}_j$ which extracts the product of all irreducible factors of f with degree in $\mathbf{I}_j$ (*coarse DDF*). If that gcd turns out to be 1, we know that $h_i = 1$ for all $i \in \mathbf{I}_j$, having computed only one gcd instead of $\#\mathbf{I}_j$ many. If the degree of the gcd is less than $2 \min \mathbf{I}_j$, then we have found an irreducible factor. Otherwise, however, a further step has to be performed to compute the h_i for $i \in \mathbf{I}_j$, e.g., by a linear or binary search of the interval (*fine DDF*).

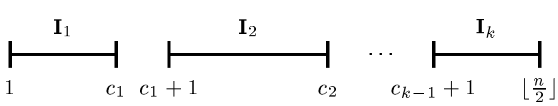

Figure 4.1: An interval partition

We now introduce some notation. Let $m = \lfloor n/2 \rfloor$. An *interval partition* of $\{1, \ldots, m\}$ is a sequence of integers $0 = c_0 < c_1 < c_2 < \cdots < c_{k-1} < c_k = m$, where $0 < k \in \mathbb{N}$ is the *length* of the partition. The sets $\mathbf{I}_j = \{c_{j-1} + 1, \ldots, c_j\}$ for $1 \leq j \leq k$ are the *intervals* of the partition (see Figure 4.1). For $c, d \in \mathbb{N}$ with $c < d$, we define $(c, d]$ to be the set of polynomials

$$\{f \in \mathbb{F}_q[x] : c < \deg p \leq d \text{ for any irreducible factor } p \text{ of } f\}.$$

By an *interval polynomial* for $\mathbf{I}_j$ we mean a polynomial in $(0, c_j]$ which is divisible by each irreducible polynomial in $(c_{j-1}, c_j]$. For example,

$$\prod_{c_{j-1} < i \leq c_j} (x^{q^i} - x) \tag{5}$$

is an interval polynomial for $\mathbf{I}_j$, used in von zur Gathen & Shoup (1992). Another example is the polynomial

$$\prod_{0 \leq i < c_j - c_{j-1}} (x^{q^{c_j}} - x^{q^i})$$

from Kaltofen & Shoup (1995). Note that interval polynomials need not be squarefree. Kaltofen & Shoup (1995) also use this notion, but their interval polynomials differ from ours in that they are already reduced modulo the polynomial to be factored.

With the above notation, a coarse DDF algorithm can be stated as follows.

Algorithm 4.2 *Coarse DDF.*
Input: A monic squarefree polynomial $f \in \mathbb{F}_q[x]$ of degree n.
Output: The polynomials $H_j = \prod_{i \in \mathbf{I}_j} h_i \in \mathbb{F}_q[x]$ for $1 \leq j \leq k$, where $h_1, \ldots, h_n$ are as in (4), plus an irreducible factor of f of degree more than $\frac{n}{2}$, if such a factor exists.

1. *$B_0 = f$.*
2. *Repeat steps 3 to 5 for $j = 1, \ldots, k$.*
3. $\quad$ *Compute the remainder I_j of an interval polynomial for $\mathbf{I}_j$ modulo B_{j-1}.*
4. $\quad H_j = \gcd(I_j, B_{j-1})$.
5. $\quad B_j = \dfrac{B_{j-1}}{H_j}$.
6. *Return $H_1, \ldots, H_k$. If $B_k \neq 1$, then also return B_k.*

For constant interval sizes, the above scheme already appears in von zur Gathen & Shoup (1992), Kaltofen & Shoup (1995), and Shoup (1995). Their algorithms only differ in the computation of the interval polynomials. Note that this is exactly the ordinary DDF algorithm when $c_j = j$ for $1 \leq j \leq m$. Intuitively, the interval partition should be chosen in such a way that the intervals increase in size, since a random polynomial has many small but only few large irreducible factors on average.

Using the remainder modulo B_{j-1} of the polynomial (5) for I_j, the cost for the computation of I_j is $(c_j - c_{j-1})$ modular qth powers and the same number of modular multiplications, and hence the cost of the above algorithm is

$$m(\mathsf{P}(n) + \mathsf{Q}(n)) + k(\mathsf{D}(n) + \mathsf{G}(n)) \in O(\mathsf{M}(n)(n \log q + k \log n))$$

operations in $\mathbb{F}_q$. Thus we have reduced the number of gcd computations from m to k in comparison to Algorithm 4.1. The price we pay for this is m additional modular multiplications—which are cheaper than gcd computations—and the fact that we do not yet have the complete DDF of g. If $H_j \neq 1$ and the degree of H_j is less then $2(c_{j-1} + 1)$, we know that H_j is irreducible and equal to $h_{\deg H_j}$, and also $h_i = 1$ for $c_{j-1} < i \leq c_j$ with $i \neq \deg H_j$. Otherwise, a fine DDF on H_j will be performed, but the hope is that this will not happen very often.

In practice, the algorithm will be stopped as soon as $\deg B_j < 2(c_{j-1}+1)$, since then B_j must be irreducible and equal to $h_{\deg b_j}$ (this is sometimes called *early abort*), but we do not take this into account in the following analysis in order to keep things simple.

Von zur Gathen & Shoup (1992) and Kaltofen & Shoup (1995) use the interval partition $c_j = lj$ with constant interval sizes and $l = \lceil n^\beta \rceil$ for some real constant β with $0 \le \beta \le 1$. Their algorithms rely on fast multipoint evaluation over the ring $\mathbb{F}_q[x]/(f)$ and on fast matrix multiplication for the computation of the interval polynomials. The cost for the gcd computations in Kaltofen & Shoup (1995) is $O((n^\beta + n^{1-\beta})\mathsf{M}(n)\log n)$.

5 Worst case analysis of the coarse / fine DDF algorithm with polynomially growing interval sizes

In this section, we assume that the interval partition is defined by $c_j = \min\{\lceil j^d \rceil, m\}$ for some $d \in \mathbb{R}$ with $d \ge 1$, where $m = \lfloor n/2 \rfloor$. For simplicity, we assume that $d \in \mathbb{N}$ in the sequel. The number of intervals is $k = \lceil m^{1/d} \rceil$. We will analyze the total cost to compute the complete DDF in the worst case when using a linear search fine DDF algorithm, as follows.

Algorithm 5.1 *Linear search fine DDF.*
Input: $j \in \{1, \ldots, k\}$, *the polynomial* $H_j = \prod_{i \in \mathbf{I}_j} h_i \in \mathbb{F}_q[x]$, *where the* h_i *are as in (4), and* $A_j = x^{q^{c_{j-1}}} \operatorname{rem} H_j \in \mathbb{F}_q[x]$.
Output: The polynomials $h_i \in \mathbb{F}_q[x]$ *for* $i \in \mathbf{I}_j$.

1. *Set* $b_{c_{j-1}} = H_j$ *and* $a_{c_{j-1}} = A_j$.
2. *Repeat steps 3 to 5 for* $i = c_{j-1}+1, \ldots, c_j$.
3. $a_i = a_{i-1}^q \operatorname{rem} b_i$.
4. $h_i = \gcd(a_i - x, b_{i-1})$.
5. $b_i = \dfrac{b_{i-1}}{h_i}$.
6. *Return* $h_{c_{j-1}+1}, \ldots, h_{c_j}$.

This is just the part of the ordinary DDF algorithm for the interval $\mathbf{I}_j$, with the only exception that the polynomial $b_{c_{j-1}}$ at the beginning of the iteration is in $(c_{j-1}, c_j]$ and not only in $(c_{j-1}, n]$. It was also used by von zur Gathen & Shoup (1992) and Kaltofen & Shoup (1995).

The cost for this algorithm is at most $dn^{(d-1)/d}(\mathsf{Q}(n) + \mathsf{G}(n) + \mathsf{D}(n))$ or $O(n^{(d-1)/d}\mathsf{M}(n)(\log q + \log n))$ operations in $\mathbb{F}_q$. Since the coarse DDF algorithm already computes the remainder of $x^{q^{c_{j-1}}}$ modulo some multiple of H_j, the cost for the computation of A_j is $\mathsf{R}(n)$ operations in $\mathbb{F}_q$ for one division with remainder by H_j for $1 \le j \le k$. Thus the overall cost for both the coarse and the fine DDF algorithm is at most

$$\frac{n}{2}\mathsf{P}(n) + (\frac{n}{2} + dn^{(d-1)/d})\mathsf{Q}(n) + n^{1/d}\mathsf{R}(n) + (dn^{(d-1)/d} + n^{1/d})(\mathsf{D}(n) + \mathsf{G}(n))$$

operations in $\mathbb{F}_q$ for n large enough. Minimizing the two exponents $(d-1)/d$ and $1/d$ leads to the following result.

Theorem 5.2 *The distinct degree factorization of a squarefree polynomial* $f \in \mathbb{F}_q[x]$ *of degree* n *can be computed using* $O(n^{1/2}\mathsf{M}(n)\log n)$ *operations in* $\mathbb{F}_q$ *for gcd computations, and* $O(n \cdot \mathsf{M}(n)\log q)$ *operations in* $\mathbb{F}_q$ *in total. This can be achieved by means of a coarse DDF algorithm with interval partition defined by* $c_j = j^2$ *and a fine DDF algorithm with linear interval search.*

Note that this saves a factor of $\log n$ in comparison to the asymptotic running time of Algorithm 4.1, which in particular for $q = 2$ is a significant gain.

It follows from the results in von zur Gathen *et al.* (1995) that for polynomially growing interval sizes, the expected total number of factors in the "bad" intervals is constant for random inputs, so that a fine DDF algorithm with binary interval search promises to save some more costly gcd computations.

6 Employing irreducibility tests

In this section, we indicate how to speed up the coarse DDF algorithm by using an irreducibility test. Suppose that we have already found all irreducible factors of $f \in \mathbb{F}_2[x]$ of degree at most $c \in \mathbb{N}$, and that a "large" factor $b \in \mathbb{F}_2[x]$ collecting all irreducible factors of f of degree more than c remains. If b is irreducible, then the coarse DDF algorithm will only find this out after reaching the degree $\deg b/2$, and this may take most of the total time spent for factoring f, as our experiments in section 7 indicate.

In our implementation, we run an irreducibility test on b in parallel to the coarse DDF algorithm on a second machine. If the coarse DDF algorithm finds a factor, the irreducibility test is aborted and restartet for the remaining polynomial. If, however, the irreducibility test says that the polynomial is irreducible, then the coarse DDF algorithm is aborted. The irreducibility test we use is based on Fact 7.3 and Theorem 7.5 in von zur Gathen & Shoup (1992), has an asymptotic running time of $O((n^2 + n^{1/2} \cdot \mathsf{M}(n))\log^2 n/\operatorname{loglog} n)$ operations in $\mathbb{F}_2$ for a squarefree polynomial in $\mathbb{F}_2[x]$ of degree n, and uses space for $O(n^{3/2})$ elements of $\mathbb{F}_2$, where $n = \deg b$. It involves the computation of matrix products of size $n/2 \times n/2$, which we compute with $O(n^{3/2})$ operations in $\mathbb{F}_2$, using classical matrix arithmetic. The same technique was also used in Shoup (1995) for the computation of modular compositions. Asymptotically, the total time for the irreducibility test is almost the same order of magnitude as the $O(n \cdot \mathsf{M}(n))$ bound for the coarse DDF algorithm, but in our implementation it significantly reduces the total running time. We have further speeded up the irreducibility test by making use of intermediate data computed by the coarse DDF algorithm. As a consequence, the irreducibility test is the faster, the higher the degree is that the coarse DDF algorithm has reached when the irreducibility test is launched. The details can be found in von zur Gathen & Gerhard (1996).

7 Implementation and running times

In this section, we describe our implementation of the polynomial factorization algorithm over $\mathbb{F}_2$ on two Sparc Ultra 1 computers, rated at 143 MHz each. The software is written in `C++`. Polynomials over $\mathbb{F}_2$ are represented as arrays of 32-bit unsigned integers, and 32 consecutive coefficients of a polynomial are packed into one machine word. We built a C++ class for polynomials over $\mathbb{F}_2$ offering standard operations like copying, reversing, shifting, and determining the degree of polynomials, the arithmetic operations addition, multiplication, squaring, division with remainder, and the Extended Euclidean Algorithm.

Polynomial multiplication. We have implemented several algorithms for polynomial multiplication over $\mathbb{F}_2$: the school method, Karatsuba & Ofman, Cantor's method over $\mathbb{F}_{2^{16}}$ and over $\mathbb{F}_{2^{32}}$, and our extension of Cantor's method over $\mathbb{F}_{2^{20}}$. We did not implement Schönhage's algorithm. The timings of Reischert (1995) indicate that in his implementation, it beats Cantor's method for degrees above 500,000, and for degrees around 40,000,000, Schönhage's algorithm is faster than Cantor's by a factor of $\approx \frac{3}{2}$.

As basis for the classical multiplication and the method of Karatsuba & Ofman, we have tried several algorithms for the multiplication of polynomials of degree less than 32. The fastest turned out to be by 9 multiplications of 8-bit blocks à la Karatsuba & Ofman, where the 8-bit blocks are multiplied via table-lookup (the corresponding table uses 128k bytes of main memory).

Multiplication in $\mathbb{F}_{2^{16}}$ is implemented by means of an exponentiation and a discrete logarithm table with respect to a primitive element of the multiplicative group. This was also done by Montgomery (1991) and Reischert (1995). The cost for one multiplication in $\mathbb{F}_{2^{16}}$ is then essentially the cost for three table lookups and one addition of 16-bit integers. The size of each of the two tables is 256k bytes of main memory. Multiplication in $\mathbb{F}_{2^{20}}$ is done in the same way, using an exponentiation and a logarithm table of size 4M bytes each. One multiplication in $\mathbb{F}_{2^{32}}$ is reduced to essentially three multiplications in $\mathbb{F}_{2^{16}}$ in a Karatsuba & Ofman like way, using that $\mathbb{F}_{2^{32}}$ is a quadratic extension of $\mathbb{F}_{2^{16}}$.

If we want to multiply two polynomials $a, b \in \mathbb{F}_2[x]$ using multipoint evaluation and interpolation at linear subspaces of one of the three fields $\mathbb{F}_{2^m}$ with $m \in \{16, 20, 32\}$ as above, we write a and b as

$$a = \sum_{0 \le i < r} a_i y^i, \quad b = \sum_{0 \le i < r} b_i y^i,$$

with $a_i, b_i \in \mathbb{F}_2[x]$ of degree less than $\frac{m}{2}$ and $y = x^{m/2}$. Then we regard y as a new indeterminate, substitute a generator γ of $\mathbb{F}_{2^m} = \mathbb{F}_2[\gamma]$ over $\mathbb{F}_2$ for x in the a_i and b_i, and multiply the resulting polynomials over $\mathbb{F}_{2^m}[y]$, as in section 2. Finally, we replace γ by x in the coefficients of the product polynomial, and compute $ab \in \mathbb{F}_2[x]$ by substituting $x^{m/2}$ for y. In this way, we can multiply polynomials in $\mathbb{F}_2[x]$ of degree less than $m2^{m-2}$.

n	classical	K & O	Cantor $m=16$	Cantor $m=32$	this paper $m=20$
16384	0.25	0.04	0.08	0.08	0.32
32768	0.97	0.13	0.16	0.17	0.72
65536	3.87	0.39	0.33	0.36	1.60
131072	15.48	1.22	0.72	0.76	3.53
262144	61.97	3.55	1.59	1.66	7.48
524288	247.65	10.61		3.65	16.12
1048576	1002.73	31.96		8.00	35.46

Table 7.1: Average times in CPU seconds for one multiplication of two polynomials of degree $n-1$.

Table 7.1 shows the average time in CPU seconds to multiply polynomials over $\mathbb{F}_2$ with the various algorithms for 10 pseudorandomly chosen inputs. There are no entries for Cantor's algorithm with $m = 16$ for degrees larger than 262144 because this is the maximal degree for which the method works (see above). As the theory predicts, our algorithm is slower than both variants of Cantor's algorithm. The main reason is that the polynomials s_i have coefficients in $\mathbb{F}_2$ in Cantor's algorithm, while their coefficients are in $\mathbb{F}_{2^{20}}$ in our algorithm. It is interesting that our implementation of Cantor's algorithm with $m = 32$ is nearly as fast as the variant with $m = 16$ (on a different but slower machine, it is even slightly faster). We use $m = 32$ throughout.

Our implementation is about 3.3 times faster than the implementation of Montgomery (1991), which is presumably the consequence of higher processor speed, and about as fast as the implementation of Reischert (1995), whose program ran on a slower Sun Sparc 10/41 machine. The crossover point in our implementation between the classical algorithm and Karatsuba & Ofman is near degree 576, between Karatsuba & Ofman and Cantor with $m = 32$ near degree 35840.

Polynomial division. For division with remainder, we use the classical method for small degrees and Newton inversion (see Aho *et al.* 1974, Chapter 8) for large degrees. In the context of polynomial factorization, we are often in the situation that the divisor polynomial f is fixed throughout many divisions, namely the polynomial to be factored. Then Newton inversion admits the precomputation of

$$(x^{\deg f} \cdot f(x^{-1}))^{-1} \bmod x^{\deg f}, \tag{6}$$

which does not depend on the particular dividend, using $O(\mathsf{M}(\deg f))$ operations in $\mathbb{F}_2$, and the cost for computing one remainder modulo f is essentially the cost for two polynomial multiplications of degree less than $\deg f$. If we use an evaluation / interpolation scheme like Cantor's algorithm for polynomial multiplication, further savings are possible by precomputing the multipoint evaluation of f and of the polynomial (6). This reduces the cost for one remainder computation modulo f to about $\frac{4}{3}$ the cost for one polynomial multiplication of degree less than $\deg f$. A similar trick was used by Shoup (1995).

n	classical	Newton inversion	
		precomp.	remainder comp.
16384	0.24	0.07	0.09
32768	0.95	0.20	0.27
65536	3.80	0.80	0.51
131072	15.24	2.11	1.09
262144	61.06	4.95	2.38
524288	245.00	11.18	5.23
1048576	978.33	25.02	11.31

Table 7.2: Average times in CPU seconds for one division with remainder of a polynomial of degree $2n-3$ by a polynomial of degree $n-1$.

Table 7.2 shows the average time to compute one division with remainder using the classical method and Newton iteration, respectively, for 10 pseudorandomly chosen inputs. The crossover point between the two algorithms when the precomputation time is not counted is near degree 3584.

Polynomial gcds. For the computation of gcds, we use both the classical method and and a faster $O(\mathsf{M}(n) \log n)$ algorithm, also known as "half-gcd" (see Aho *et al.* 1974,

n	classical	"half-gcd"
16384	1.33	1.38
32768	5.79	3.66
65536	24.46	9.94
131072	100.56	26.85
262144	402.57	70.36
524288	1617.76	178.56
1048576	7017.88	439.81

Table 7.3: Average times in CPU seconds for one gcd of two polynomials of degree $n-1$.

Strassen 1983). Table 7.3 shows the average time in CPU seconds for the computation of one gcd using both methods for 10 pseudorandomly chosen inputs. The crossover point between the two algorithms is near degree 28672.

Polynomial factorization. Our polynomial factorization algorithm consists of the three stages described in section 3. For the squarefree factorization, we use the standard algorithm (see, e.g., Geddes *et al.* (1992), Chapter 8), with a special trick for the finite field $\mathbb{F}_2$, using the fact that $\gcd(f, f')$ is a square for any $f \in \mathbb{F}_2[x]$.

To compute the distinct degree factorization, we have implemented the coarse DDF algorithm as described in section 4, with early abort, and the interval partition defined by $c_j = 2j^2$, with intervals $\mathbf{I}_1 = \{1, 2\}$, $\mathbf{I}_2 = \{3, \ldots, 8\}$, $\mathbf{I}_3 = \{9, \ldots, 18\}$, ..., $\mathbf{I}_j = \{2(j-1)^2+1, \ldots, 2j^2\}$. Furthermore, we use a binary search fine DDF algorithm.

Using a similar trick as Montgomery (1991), we reduce the number of modular multiplications in the computation of an interval polynomial for the interval $\{c+1, \ldots, d\}$ in the coarse DDF algorithm from $d-c+1$ to about $\frac{d-c}{4}$. This saves a factor of $\frac{9}{20}$ of the running time.

We use the irreducibility test as described in section 6. The process is spawned as soon as the coarse DDF algorithm reaches degree 1000, and respawned every time the coarse DDF algorithm finds a new factor to check whether the remaining polynomial is irreducible.

The equal degree factorization is done as in Ben-Or (1981), at an expected cost of $O(d \cdot \mathsf{M}(rd) \log r)$ operations in $\mathbb{F}_2$ for a squarefree equal-degree polynomial of order d with r irreducible factors.

Factorization experiments. Table 7.4 shows examples of running times on the main machine for the factorization algorithm. The elapsed wall clock time differs from the CPU time on the main machine only by at most three per cent in all experiments, and we have omitted it. The third column contains the amount of disk space in megabytes that the algorithm used for storing intermediate results. For the examples with degree 262143, the amount of disk space used was limited by the size of the hard disk. The fourth column shows the degree at which the coarse DDF algorithm ended or was aborted when the irreducibility test certified the remaining factor to be irreducible (in the latter case, the degree is written in italic), and the last column contains the *factorization pattern*, i.e., the degree sequence of the irreducible factors of the input polynomial. For example, $1^2, 2, 3, 3$ means that the polynomial has one linear factor occurring twice, an irreducible quadratic and two different irreducible cubic factors, each of the latter occuring only once. In our experiments, we never had to perform a fine DDF or an EDF for total degrees above 8000. Since the algorithm is distributed over two machines, both the CPU time on the main machine and the elapsed wall clock time depend on the work load of both machines.

We denote by $N_i(f)$ the degree of the ith largest irreducible factor of $f \in \mathbb{F}_2[x]$. The actual running time of the algorithm for an individual polynomial f depends on several factors:

- $N_1(f)$, the degree of the largest irreducible factor. With "early abort", the coarse DDF algorithm stops near the degree $\max\{N_2(f), \lfloor N_1(f)/2 \rfloor\}$ (because of blocking, the actual abort degree maybe somewhat higher), so that the algorithm runs faster if the degrees of all irreducible factors are relatively small. However, for random polynomials this will rarely occur, since it is widely believed that there is some constant $\gamma \in \mathbb{R}$ with $0.5 < \gamma < 1$ such that $N_1(f)$ is $(\gamma + o(1))n$ on average for a random polynomial f of degree n, since a similar statement is true for integers (see Knuth & Pardo 1976).

- $N_2(f)$, the degree of the second largest irreducible factor of f. This is because the irreducibility test for the largest prime factor is launched when the coarse DDF algorithm reaches the degree $N_2(f)$.

Let $f \in \mathbb{F}_2[x]$ the polynomial to be factored, $n = \deg f$, and t the average time for the multiplication of two polynomials of degree about n in CPU seconds. Then the time used for one division with remainder of a polynomial of degree less than $2n$ by a factor of f is approximately $2t$ when using Karatsuba & Ofman's multiplication algorithm, or even about $\frac{4}{3}t$ when we use Cantor's method for multiplication. If $d \leq n/2$ is the degree where the coarse DDF algorithm stops and we neglect the cost for precomputations in the division algorithm, gcd computations, fine DDF, and EDF, the algorithm essentially performs d modular squarings and $d/4$ modular multiplications for the computation of the interval polynomials. This leads to an estimate for the total running time of $\approx \frac{11}{4}dt$ CPU seconds with Karatsuba & Ofman multiplication and $\approx \frac{23}{12}dt$ with Cantor's multiplication, which is in good accordance with the times in Tables 7.1 and 7.4. The worst case for our DDF algorithm is when f has two irreducible factors of distinct degrees, both about $n/2$. The irreducibility test is of no help in this case, and we get an estimated running time of $\frac{11}{8}nt$ and $\frac{23}{24}nt$ CPU seconds, respectively.

Factoring trinomials can be done still faster, since division with remainder by a trinomial costs essentially the same as a polynomial addition. This leads to an estimated running time of $\frac{1}{4}dt$, where d and t are as above. We have implemented a variant of our factorization algorithm for trinomials of the form $x^n + x + 1$ which exploits the sparseness, and factored the trinomial $x^{216091} + x + 1$ from Montgomery (1991) in less than 7 hours of CPU time.

References

A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading MA, 1974.

M. Ben-Or, Probabilistic algorithms in finite fields. In *Proc. 22nd IEEE Symp. Foundations Computer Science*, 1981, 394–398.

E. R. Berlekamp, Factoring polynomials over finite fields. *Bell System Tech. J.* **46** (1967), 1853–1859.

n	time	disk space	abort degree	factorization pattern
16383	$6'$	5	*2806*	1^5, 3, 4^2, 9, 361, 1667, 1827, 12503
16383	$8'$	7	*3698*	1^5, 3, 6, 7, 16, 26, 39, 80, 94, 110, 556, 2825, 12616
16383	$10'$	10	*4922*	1^4, 224, 266, 587, 1201, 4099, 10002
16383	$15'$	13	6728	1^3, 6, 24, 249, 283, 930, 6563, 8325
32767	$26'$	15	*3888*	1^2, 2^2, 8, 46, 306, 330, 32071
32767	$42'$	29	7442	1^2, 1^3, 2, 13, 23, 73, 140, 153, 393, 2145, 2177, 3308, 3695, 7245, 13395
32767	$56'$	42	10658	12, 30, 31, 34, 96, 1232, 1876, 3590, 3616, 10414, 11836
32767	$59'$	39	9839	1^2, 16, 22, 90, 102, 359, 791, 798, 1824, 9085, 19678
65535	$1^h18'$	44	*5618*	1^2, 1^4, 2, 33, 143, 319, 551, 2772, 61709
65535	$1^h41'$	59	*7498*	1^3, 1, 10, 31, 590, 824, 1037, 1898, 3831, 57310
65535	$2^h03'$	73	*9320*	1, 1, 3^2, 42, 71, 205, 607, 852, 2197, 3066, 3165, 7891, 47431
65535	$2^h14'$	79	*10082*	1^2, 5, 18, 29, 56, 80, 94, 259, 643, 1476, 3294, 8328, 51251
131071	$4^h26'$	138	*9094*	1^2, 1^2, 2, 14, 20, 23, 331, 1187, 3696, 125794
131071	$10^h10'$	331	*21184*	1, 1^3, 3, 449, 483, 1274, 18136, 110722
131071	$12^h18'$	428	27378	1, 1^4, 14, 67, 203, 631, 3546, 3580, 3877, 3924, 10400, 23894, 26057, 27069, 27804
131071	$13^h45'$	467	*29892*	1^2, 1^3, 2, 5, 8, 68, 111, 359, 1048, 1607, 12758, 15699, 28780, 70621
262143	$48^h25'$	1024	*48166*	2, 56, 110, 174, 1096, 1876, 13616, 29823, 44413, 170977

Table 7.4: CPU times for factoring some pseudorandomly chosen polynomials of degree n using two Sparc Ultra 1 computers rated at 143 MHz.

E. R. Berlekamp, Factoring polynomials over large finite fields. *Math. Comp.* **24** (1970), 713–735.

D. G. Cantor, On arithmetical algorithms over finite fields. *Journal of Combinatorial Theory, Series A* **50** (1989), 285–300.

D. G. Cantor and H. Zassenhaus, A new algorithm for factoring polynomials over finite fields. *Math. Comp.* **36** (1981), 587–592.

P. Flajolet, X. Gourdon, and D. Panario, Random polynomials and polynomial factorization. Proc. ICALP '96, to appear, 1996.

P. Fleischmann and P. Roelse, Comparative implementations of Berlekamp's and Niederreiter's polynomial factorization algorithms. Preprint, 1995.

S. Gao and J. von zur Gathen, Berlekamp's and Niederreiter's polynomial factorization algorithms. In *Finite Fields: Theory, Applications and Algorithms*, ed. G. L. Mullen and P. J.-S. Shiue, vol. 168 of *Contemporary Mathematics*. Amer. Math. Soc., 1994, 101–115.

J. von zur Gathen and J. Gerhard, Arithmetic and factorization of polynomials over $\mathbb{F}_2$. Technical report, University of Paderborn, to appear, 1996.

J. von zur Gathen and V. Shoup, Computing Frobenius maps and factoring polynomials. *Comput complexity* **2** (1992), 187–224.

J. von zur Gathen, X. Gourdon, and D. Panario, Average-case analysis of some polynomial factorization algorithms. Unpublished, 1995.

K. O. Geddes, S. R. Czapor, and G. Labahn, *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

E. Kaltofen, Polynomial factorization 1987–1991. In *Proc. Latin'92, Lecture Notes in Computer Science* **583**, São Paulo, Brazil, 1992, 294–313.

E. Kaltofen and A. Lobo, Factoring high-degree polynomials by the black box Berlekamp algorithm. In *Proc. ISSAC '94*, ed. J. von zur Gathen and M. Giesbrecht. ACM Press, 1994, 90–98.

E. Kaltofen and V. Shoup, Subquadratic-time factoring of polynomials over finite fields. In *Proc. 27th Annual ACM Symp. Theory of Computing*. ACM Press, 1995, 398–406.

A. Karatsuba and Y. Ofman, Умножение многозначных чисел на автоматах. *Dokl. Akad. Nauk USSR* **145** (1962), 293–294. Multiplication of multidigit numbers on automata, Soviet Physics–Doklady **7** (1963), 595-596.

D. E. Knuth and L. Trabb Pardo, Analysis of a simple factorization algorithm. *Theoretical Computer Science* **3** (1976), 321–348.

P. L. Montgomery, Factorization of $X^{216091} + X + 1 \bmod 2$ — a problem of Herb Doughty. Preprint, 1991.

H. Niederreiter, New deterministic factorization algorithms for polynomials over finite fields. *Contemporary Mathematics* **168** (1994), 251–268.

D. Reischert, Schnelle Multiplikation von Polynomen über GF(2) und Anwendungen. Diplomarbeit, University of Bonn, Germany, 1995.

A. Schönhage, Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Inf.* **7** (1977), 395–398.

V. Shoup, A new polynomial factorization algorithm and its implementation. To appear in J. Symb. Comp., 1995.

V. Strassen, The computational complexity of continued fractions. *SIAM J. Comput.* **12** (1983), 1–27.

D. Y. Y. Yun, On square-free decomposition algorithms. In *Proc. ACM Symp. Symbolic and Algebraic Computation*, ed. R. D. Jenks, 1976, 26–35.