On the bit-complexity of sparse polynomial and series multiplication*

Joris van der Hoeven

Laboratoire d'Informatique
UMR 7161 CNRS
École polytechnique
91128 Palaiseau Cedex
France

Email: `joris@texmacs.org`
Web: `http://www.lix.polytechnique.fr/~vdhoeven`

Grégoire Lecerf

Laboratoire de Mathématiques
UMR 8100 CNRS
Université de Versailles
45, avenue des États-Unis
78035 Versailles Cedex
France

Email: `Gregoire.Lecerf@math.uvsq.fr`
Web: `http://www.math.uvsq.fr/~lecerf`

December 1, 2010

In this paper we present various algorithms for multiplying multivariate polynomials and series. All algorithms have been implemented in the C++ libraries of the Mathemagix system. We describe naive and softly optimal variants for various types of coefficients and supports and compare their relative performances. For the first time we are able to observe the benefit of non naive arithmetic for multivariate polynomials and power series, which might lead to speed-ups in several areas of symbolic and numeric computation.

For the sparse representation, we present new softly linear algorithms for the product whenever the destination support is known, together with a detailed bit-complexity analysis for the usual coefficient types. As an application, we are able to count the number of the absolutely irreducible factors of a multivariate polynomial with a cost that is essentially quadratic in the number of the integral points in the convex hull of the support of the given polynomial. We report on examples that were previously out of reach.

Keywords: sparse multiplication, power series, multi-point evaluation, polynomial factorization, algorithm

A.M.S. subject classification: 68W30, 12-04, 30B10, 42-04, 11Y05

∗. This work has been partly supported by the French ANR-09-JCJC-0098-01 MaGiX project, and by the Digiteo 2009-36HD grant of the Région Ile-de-France.

1. INTRODUCTION

It is classical that the product of two integers, or two univariate polynomials over any ring, can be performed in *softly linear time* for the usual dense representations [SS71, Sch77, CK91, Für07]. More precisely, two integers of bit-size at most n can be multiplied in time $\tilde{O}(n) = n(\log n)^{O(1)}$ on a Turing machine, and the product of two univariate polynomials can be done with $\tilde{O}(n)$ arithmetic operations in the coefficient ring. These algorithms are widely implemented in computer algebra systems and turn out to perform well even for problems of medium sizes.

Concerning multivariate polynomials and series less effort has been dedicated towards such fast algorithms and implementations. One of the difficulties is that the polynomials and series behave differently according to their support. In this paper we propose several algorithms that cover a wide range of situations.

1.1. Related works

Representations of multivariate polynomials and series with their respective efficiencies have been discussed since the early ages of computer algebra; for historical references we refer the reader to [Joh74, Sto84, DST87, CGL92]. The representation is an important issue which conditions the performance in an intrinsic way. It is customary to distinguish three main types of representations: dense, sparse, and functional.

A *dense representation* is made of a compact description of the support of the polynomial and the sequence of its coefficients. The main example concerns *block supports* – it suffices to store the coordinates of two opposite vertices. In a dense representation all the coefficients of the considered support are stored, even if they are zero. In fact, if a polynomial has only a few of non-zero terms in its bounding block, we shall prefer to use a *sparse representation* which stores only the sequence of the non-zero terms as pairs of monomials and coefficients. Finally, a *functional representation* stores a function that can produce values of the polynomials at any given points. This can be a pure blackbox (which means that its internal structure is not supposed to be known) or a specific data structure such as *straight-line programs* (see Chapter 4 of [BCS97] for instance).

For dense representations with block supports, it is classical that the algorithms used for the univariate case can be naturally extended: the naive algorithm, Karatsuba's algorithm, and even fast Fourier transforms [CT65, SS71, CK91, Hoe04] can be applied recursively in each variable, with good performance. Another classical approach is the Kronecker substitution which reduces the multivariate product to one variable only. We refer the reader to classical books such as [BP94, GG02]. When the number of the variables is fixed and the partial degrees tend to infinity, these techniques lead to softly linear costs.

For sparse representations, the naive school book algorithm, that performs all the pairwise term products, is implemented in all the computer algebra systems and several other dedicated libraries. It is a matter of constant improvements in terms of data structures, memory management, vectorization and parallelization [Yan98, GL06, MP07, MP09a, MP09b] (see [Fat03] for some comparisons between several implementations available in 2003).

Multivariate dichotomic approaches have not been discussed much in the literature. Let us for instance consider Karatsuba's algorithm (see [GG02, Chapter 8] for details). With one variable z only, the usual way the product is performed begins with splitting P and Q into $P_0(z) + z^k P_1(z)$ and $Q_0(z) + z^l Q_1(z)$, respectively, where k and l are half of the degrees of P and Q. Then we compute recursively $P_0 Q_0$, $P_1 Q_1$, and $(P_0 + P_1)(Q_0 + Q_1)$, and perform suitable linear combinations to recover the result. This approach is efficient in the dense block case because the sizes of the input are correctly divided in each recursive call of the product. In the sparse case this phenomenon hardly occurs, and it is commonly admitted that this approach is useless (see for instance [Fat03, Section 3] or [MS04] for a cost analysis). Nevertheless block versions have been suggested to be useful with several variables in [Hoe02, Section 6.3.3], and refined in [Hoe06, Section 6], but never tested in practice. Further improvements are under progress in [HL10].

In the sparse case, the product can be decomposed into two subproblems: (1) determine the support of R from those of P and Q, (2) compute the coefficients of R. These are independent in terms of complexity and applications. The computation of the support is the most expensive part, that can be seen as a special case of an even more difficult problem called *sparse interpolation*. This is a cornerstone in higher level routines such as the greatest common divisor: to compute the g.c.d. of two polynomials in the sparse representation it is interesting to specialize all the variables but one at several points, compute as many univariate g.c.d.s, and interpolate the result without a precise idea of the support (see for instance [KT90]). We are not dealing with this problem in this paper, but most of the literature in this topic hides fast algorithms for the sparse product of polynomials as soon as the destination support is known, as explained in the next paragraphs.

For coefficient fields of characteristic zero, it is proved in [CKL89] that the product of two polynomials in sparse representation can be computed in softly linear time in terms of operations over the ground field, once the destination support is known. This algorithm uses fast evaluation and interpolation at suitable points built from prime numbers. Unfortunately, the method hides an expensive growth of intermediate integers involved in the linear transformations, which prevents the algorithm from being softly linear in terms of bit-complexity. Indeed this algorithm was essentially a subroutine of the sparse interpolation algorithm of [BT88], with the suitable set of evaluation points borrowed from [GK87]. For more references on sparse interpolation in characteristic zero we refer the reader to [KL88, KLW90, KLL00, GS09].

For coefficients in a finite field, Grigoriev, Karpinski and Singer designed a specific sparse interpolation algorithm in [GKS90], which was then improved in [Wer94, GKS94]. These algorithms are based on special point-sets for evaluation and interpolation, built from a primitive element of the multiplicative subgroup of the ground field. As in [CKL89] a fast product might have been be deduced from this work, but to the best of our knowledge this has never been done until now.

1.2. Our contributions

The main contributions of this paper are practical algorithms for faster computations with multivariate polynomials and series. In Sections 2 and 3 we describe naive algorithms for the dense and sparse representations of polynomials, we recall the Kronecker substitution technique, and discuss bit-complexities with regards to practical performances. We show that our implementations are competitive with the best other software, and we discuss thresholds between sparse and dense representations. Section 4 is devoted to naive algorithms for power series.

In Section 5, we turn to the sparse case. Assuming the destination support to be known, we will focus on the computation of the coefficients. Our approach is similar to [CKL89], but relies on a different kind of evaluation points, which first appeared in [GKS90]. The fast product from [CKL89], which only applies in characteristic zero, suffers from the swell of the intermediate integers. In contrast, our method is primarily designed for finite fields. For integer coefficients we either rely on large primes or the multi-modular methods to deduce new bit-complexity bounds. Section 5 is devoted to the bit-complexity for the most frequently encountered coefficient rings.

Of course, our assumption that the support of the product is known is very strong: in many cases, it can be feared that the computation of this support is actually the hardest part of the multiplication problem. Nevertheless, the computation of the support is negligible in many cases:

1. The coefficients of the polynomials are very large: the support can be computed with the naive algorithms, whereas the coefficients are deduced with the fast ones. A variant is when we need to compute the products of many pairs of polynomials with the same supports.

2. The destination support can be deduced by a simple geometric argument. A major example concerns dense formal power series, truncated by total degree. In Section 6 we adapt the method of [LS03] to our new evaluation-interpolation scheme. The series product of [LS03] applies in characteristic zero only and behaves badly in terms of bit-complexity. Our approach again starts with coefficient fields of positive characteristic and leads to much better bit-costs and useful implementations.

3. When searching for factors of a multivariate polynomial P, the destination support is precisely the support of P. In Section 7 we present a new algorithm for counting the number of absolutely irreducible factors of P. We will prove a new complexity bound in terms of the size of the integral hull of the support of P, and report on examples that were previously out of reach. In a future work, we hope to extend our method into a full algorithm for factoring P.

Our fast product can be expected to admit several other applications, such as polynomial system solving, following [CKL89], but we have not tried.

Most of the algorithms presented in this paper have been implemented in the `C++` library `multimix` of the free computer algebra system Mathemagix [H+02] (revision 4741, available from `http://gforge.inria.fr/projects/mmx/`). Up to our knowledge, this is the first implementation of a sparse multivariate multiplication algorithm with a softly linear asymptotic time complexity.

2. Multiplication of block polynomials

In this section we recall several classical algorithms for computations with dense multivariate polynomials, using the so called "block representation". We will also analyze the additional bit-complexity due to operations on the exponents.

The algorithms of this section do not depend on the type of the coefficients. We let $\mathbb{A}$ be an effective ring, which means that all ring operations can be performed by algorithm. We will denote by $\mathsf{M}(n)$ the cost for multiplying two univariate polynomials of degree n, in terms of the number of arithmetic operations in $\mathbb{A}$. Similarly, we denote by $\mathsf{I}(n)$ the time needed to multiply two integers of bit-size at most n. One can take $\mathsf{M}(n)=O(n\log n\log\log n)$ [CK91] and $\mathsf{I}(n)=O(n\log n\, 2^{\log^* n})$ [Für07], where $\log^*$ represents the iterated logarithm of n. Throughout the paper, we will assume that $\mathsf{M}(n)/n$ and $\mathsf{I}(n)/n$ are increasing. We also assume that $\mathsf{M}(O(n))\subseteq O(\mathsf{M}(n))$ and $\mathsf{I}(O(n))\subseteq O(\mathsf{I}(n))$.

2.1. Dense polynomials using the block representation

Any polynomial P in $\mathbb{A}[z_1, \ldots, z_n]$ is made of a sum of terms, with each term composed of a coefficient and an exponent seen as a vector in $\mathbb{N}^n$. For an exponent $e=(e_1,\ldots,e_n)\in\mathbb{N}^n$, the monomial $z_1^{e_1}\cdots z_n^{e_n}$ will be written z^e. For any $e\in\mathbb{N}^n$, we let P_e denote the coefficient of z^e in P. The *support* of P is defined by $\operatorname{supp} P = \{e\in\mathbb{N}^n : P_e\neq 0\}$.

A *block* is a subset of $\mathbb{N}^n$ of the form $\prod_{j=1}^n \{0,1,\ldots,d_j-1\}$, with $d_1,\ldots,d_n\in\mathbb{N}$. Given a polynomial $P\in\mathbb{A}[z_1,\ldots,z_n]$, its *block support* is the smallest block of the form

$$\operatorname{dsupp}(P)=\prod_{j=1}^{n}\{0,1,\ldots,d_{P,j}-1\}$$

with $\operatorname{supp}(P)\subseteq\operatorname{dsupp}(P)$. In other words, assuming $d_P\neq 0$, we have $d_{P,j}=\deg_{z_j} P+1$ for $j=1,\ldots,n$. We will denote by $d_P=d_{P,1}\cdots d_{P,n}$ the cardinality of $\operatorname{dsupp}(P)$. In the dense *block representation* of P, we store the $d_{P,i}$ and all the coefficients corresponding to the monomials of $\operatorname{dsupp}(P)$.

We order the exponents in the *reverse lexicographic order*, so that

$$x_1^{e_1}\cdots x_n^{e_n}<x_1^{f_1}\cdots x_n^{f_n} \iff \exists j,(e_n=f_n\wedge\cdots\wedge e_{j+1}=f_{j+1}\wedge e_j<f_j).$$

In this way, the i-th exponent $e=(e_1,\ldots,e_n)=\operatorname{exponent}(i,P)$ of P is defined by

$$i=e_1+e_2\,d_{P,1}+e_3\,d_{P,1}\,d_{P,2}+\cdots+e_n\,d_{P,1}\,d_{P,2}\cdots d_{P,n-1}.$$

Conversely, we will write $i=\operatorname{index}(e,P)$ and call i the *index* of e in P. Notice that the index has values from 0 to d_P-1. The coefficient of the exponent e of index i will be written $\operatorname{coefficient}(e,P)$ or $\operatorname{coefficient}(i,P)$, according to the context.

In the cost analysis of the algorithms below, we shall take into account the number of operations in $\mathbb{A}$ but also the bit-cost involved by the arithmetic operations with the exponents.

2.2. Naive product

Let P and Q be the two polynomials that we want to multiply. Their product $R = PQ$ can be computed naively by performing the pairwise products of the terms of P and Q as follows:

Algorithm 1. Naive product for block polynomials.

```
Set $R := 0$
For $i$ from $0$ to $d_P - 1$ do:
    For $j$ from $0$ to $d_Q - 1$ do:
        i. $l := \mathrm{index}(\mathrm{exponent}(i, P) + \mathrm{exponent}(j, Q), R)$;
        ii. Add $\mathrm{coefficient}(i, P)\,\mathrm{coefficient}(j, Q)$ to $\mathrm{coefficient}(l, R)$;
```

Proposition 2. *Assuming the block representation, the product R of P and Q can be computed using $O(d_P d_Q)$ operations in $\mathbb{A}$ plus $O(n\,\mathsf{l}(\log d_R) + d_P d_Q \log d_R)$ bit-operations.*

Proof. Before entering Algorithm 1 we compute all the $d_{R,i}$ and discard all the variables z_i such that $d_{R,i} = 1$. This takes no more that $O(n\,\mathsf{l}(\log d_R))$ bit-operations. Then we compute $d_{R,1} d_{R,2}$, $d_{R,1} d_{R,2} d_{R,3}$, ..., $d_{R,1} d_{R,2} \cdots d_{R,n-1}$, as well as $d_{R,1}(d_{Q,2} - 1)$, $d_{R,1}\, d_{R,2}\, (d_{Q,3} - 1)$, ..., $d_{R,1}\, d_{R,2} \cdots d_{R,n-2}\, (d_{Q,n-1} - 1)$, and $d_{R,1}\, (d_{P,2} - 1)$, $d_{R,1} d_{R,2} (d_{P,3} - 1)$, ..., $d_{R,1} d_{R,2} \cdots d_{R,n-2} (d_{P,n-1} - 1)$ in $O(n\,\mathsf{l}(\log d_R))$ bit-operations.

For computing efficiently the index l at step (i) we make use of the enumeration strategy presented in Lemma 4 below. In fact, for each i, we compute index(exponent(i, P), R) incrementally in the outer loop. Then for each j we also obtain $l :=$ index(exponent(i, P) exponent(j, Q), R) incrementally during the inner loop. The conclusion again follows from Lemma 4. □

Notice that for small coefficients (in the field with two elements for instance), the bit-cost caused by the manipulation of the exponents is not negligible. This naive algorithm must thus be programmed carefully to be efficient with many variables in small degree.

For running efficiently over all the monomials of the source and the destination supports we use the following subroutine:

Algorithm 3. Next index.

```
Input: $e \in \mathrm{supp}(P)$, $f \in \mathrm{supp}(Q)$, and the index $k$ of $e + f$ in $R = PQ$.
Output: $f' = \mathrm{exponent}(\mathrm{index}(f, Q) + 1, Q)$, and $\mathrm{index}(e + f', R)$.
    1. Let $f' := f$; Let $i := 1$;
    2. For $i$ from $1$ to $n$ do:
        a. if $d_{Q,i} = 1$ then continue;
        b. if $f'_i \neq d_{Q,i} - 1$ then return $(f'_1, \ldots, f'_{i-1}, f'_i + 1, f'_{i+1}, \ldots, f'_n)$ and
           $k + d_{R,1} \cdots d_{R,i-1}$;
        c. $f'_i := 0$; $k := k - d_{R,1} \cdots d_{R,i-1}\,(d_{Q,i} - 1)$;
```

3. Return an error.

The algorithm raises an error if, and only if, f is the last exponent of Q. The proof of correctness is straightforward, hence is left to the reader. In fact, we are interested in the total cost spent in the successive calls of this routine to enumerate the indices of all the exponents of $e+\operatorname{dsupp}(Q)$ in R, for any fixed exponent e of P:

LEMMA 4. *Assume that $d_{R,i}\geqslant 2$ for all i, and let e be an exponent of P. If $d_{R,1}\, d_{R,2}$, $d_{R,1}\, d_{R,2}\, d_{R,3}, \ldots, d_{R,1}\, d_{R,2} \cdots d_{R,n-1}$ and $d_{R,1}\,(d_{Q,2}-1)$, $d_{R,1}\, d_{R,2}\,(d_{Q,3}-1)$, …, $d_{R,1}\, d_{R,2} \cdots d_{R,n-2}\,(d_{Q,n-1}-1)$ are given, and if $\operatorname{index}(e, R)$ is known, then the indices in R of the exponents of $e+\operatorname{dsupp}(Q)$ can be enumerated in increasing order with $O(d_Q \log d_R)$ bit-operations.*

Proof. Let m be the number of the $d_{Q,i}$ equal to 1. Each step of the loop of Algorithm 3 takes $O(1)$ if $d_{Q,i}=1$ or $O(\log d_R)$ bit-operations otherwise. Let $d_{Q,i_1}, \ldots, d_{Q,i_{n-m}}$ be the subsequence of $(d_{Q,i})_i$ which are not 1.

When running over all the successive exponents of $\operatorname{dsupp}(Q)$, this loop takes $O(m+\log d_R)$ bit-operations for at most d_Q exponents, and $O(m+2\log d_R)$ bit-operations for at most $d_Q/d_{Q,i_1}$ exponents, and $O(m+3\log d_R)$ bit-operations for at most $d_Q/(d_{Q,i_1}\, d_{Q,i_2})$ exponents, etc. Since the $d_{Q,i_j}\geqslant 2$ for all j, this amounts to $O((m+\log d_R)\, d_Q)$ operations. Since the $d_{R,i}\geqslant 2$ for all i, the conclusion follows from $m=O(\log d_R)$. □

2.3. Kronecker substitution

Let us briefly recall the Kronecker substitution. For computing $R=PQ$, the Kronecker substitution we need is defined as follows:

$$\begin{array}{rcl} K_{d_R}\colon \mathbb{A}[z_1,\ldots,z_n] & \longrightarrow & \mathbb{A}[x] \\ P & \longmapsto & P(x, x^{d_{R,1}}, x^{d_{R,1} d_{R,2}}, \ldots, x^{d_{R,1}\cdots d_{R,n-1}}). \end{array}$$

It suffices to compute $K_{d_R}(P)$ and $K_{d_R}(Q)$, perform their product, and recover R by

$$R=K_{d_R}^{-1}(K_{d_R}(P)\, K_{d_R}(Q)).$$

PROPOSITION 5. *Assuming the block representation, the product $R=PQ$ can be computed using $\mathsf{M}(d_R)$ operations in $\mathbb{A}$ plus $O(n\,\mathsf{I}(\log d_R)+(d_P+d_Q)\log d_R)$ bit-operations.*

Proof. As for the naive approach we start with computing all the $d_{R,i}$ and we discard all the variables z_i such that $d_{R,i}=1$. Then we compute $d_{R,1}\, d_{R,2}$, $d_{R,1}\, d_{R,2}\, d_{R,3}$, …, $d_{R,1}\, d_{R,2}\cdots d_{R,n-1}$ and $d_{R,1}\,(d_{Q,2}-1)$, $d_{R,1}\, d_{R,2}\,(d_{Q,3}-1)$, …, $d_{R,1}\, d_{R,2}\cdots d_{R,n-2}\,(d_{Q,n-1}-1)$ and $d_{R,1}\,(d_{P,2}-1)$, $d_{R,1}\, d_{R,2}\,(d_{P,3}-1)$, …, $d_{R,1}\, d_{R,2}\cdots d_{R,n-2}\,(d_{P,n-1}-1)$. This takes $O(n\,\mathsf{I}(\log d_R))$ bit-operations. Thanks to Lemma 4, this allows to deduce $K_{d_R}(P)$ and $K_{d_R}(Q)$ with $O((d_P+d_Q)\log d_R)$ bit-operations. Thanks to the reverse lexicographic ordering, the inverse $K_{d_R}^{-1}$ is for free. □

Remark 6. A similar complexity can be obtained using evaluation-interpolation methods, such as the fast Fourier transform [CT65] or Schönhage-Strassen's variant [SS71, CK91]. For instance, assuming that $\mathbb{A}$ has sufficiently many 2^p-th roots of unity, we have $\mathsf{M}(n) = O(n \log n)$ using FFT-multiplication. In the multivariate case, the multiplication of P and Q essentially reduces to $3\, d_R / d_{R,j}$ fast Fourier transforms of size $d_{R,j}$ with respect to each of the variables z_j. This amounts to a total number of $O(\log d_{R,1} + \cdots + \log d_{R,n})\, d_R = O(d_R \log d_R)$ operations in $\mathbb{A}$.

Over the integers, namely when $\mathbb{A} = \mathbb{Z}$, one can further apply the Kronecker substitution to reduce to the multiplication of two large integers. For any integer a we write $l_a = \lceil \log_2 (|a| + 1) \rceil$ for its *bit-size*, and denote by $h_P = \max_e l_{P_e}$ the maximal bit-length of the coefficients of P (and similarly for Q and R). Since

$$\max_e |R_e| \leqslant \min(d_P, d_Q) \max_e |P_e| \max_e |Q_e|,$$

we have

$$h_R \leqslant h := h_P + h_Q + l_{\min(d_P, d_Q)}.$$

The coefficients of R thus have bit-length at most h. We will be able to recover them (with their signs) from an approximation modulo 2^{h+1}. The substitution works as follows:

$$\begin{array}{rcl} K_{d_R,h}\colon \mathbb{Z}[z_1, \ldots, z_n] & \longrightarrow & \mathbb{Z} \\ P & \longmapsto & K_{d_R}(P)(2^{h+1}). \end{array}$$

One thus computes $K_{d_R,h}(P)$ and $K_{d_R,h}(Q)$, does the integer product, and recovers

$$R = K_{d_R,h}^{-1}(K_{d_R,h}(P)\, K_{d_R,h}(Q)).$$

COROLLARY 7. *With the above dense representation, the product R of P times Q in $\mathbb{Z}[z_1, \ldots, z_n]$ takes $O(\mathsf{I}(h\, d_R) + n\, \mathsf{I}(\log d_R) + (d_P + d_Q) \log d_R)$ bit-operations.*

Proof. The evaluation at 2^{h+1} takes linear time thanks to the binary representation of the integers being used. The result thus follows from the previous proposition. □

Remark 8. In a similar way, we may use Kronecker substitution for the multiplication of polynomials with modular coefficients in $\mathbb{A} = \mathbb{Z}/p\mathbb{Z}$, $p \in \{2, 3, \ldots\}$. Indeed, we first map $P, Q \in \mathbb{A}[z_1, \ldots z_n]$ to polynomials in $\{0, \ldots, p-1\}[z_1, \ldots, z_n] \subseteq \mathbb{Z}[z_1, \ldots, z_n]$, multiply them as integer polynomials, and finally reduce modulo p.

2.4. Timings

In this paper we will often illustrate the performances of our implementation for $\mathbb{A} = \mathbb{Z}/p\mathbb{Z}$, with $p = 268435459 < 2^{29}$. Timings are measured in seconds or milliseconds, using one core of an INTEL XEON X5450 at 3.0GHz running LINUX and GMP 5.0.0 [Gra91]. The symbol ∞ in the timing tables means that the time needed is very high, and not relevant.

In Tables 1 and 2 we multiply dense polynomials P and Q with $d_{P,i} = d_{Q,i} = d_i$, for $i = 1, 2, \ldots$ We observe that the Kronecker substitution is a very good strategy: it involves less operations on the exponents, and fully benefits from the performances of GMP.

d_1, d_2	10	20	40	80	160
naive	0.23	3.4	54	855	13 616
Kronecker	0.05	0.28	1.7	8.7	42
d_P, d_Q	100	400	1 600	64 000	25 600
d_R	361	1521	6 241	25 281	101 761

Table 1. Block polynomial product with 2 variables (in milliseconds)

d_1, d_2, d_3	10	20	40	80	160
naive	23	1 390	88 643	∞	∞
Kronecker	1.9	25	303	3 208	33 983
d_P, d_Q	1 000	8 000	64 000	512 000	4 096 000
d_R	6 859	59 319	493 039	4 019 679	32 461 759

Table 2. Block polynomial product with 3 variables (in milliseconds)

3. Naive multiplication of sparse polynomials

In this section, we will study the naive multiplication of multivariate polynomials using a sparse representation. Our naive implementation involves an additional dichotomy for increasing its cache efficiency.

3.1. Naive dichotomic multiplication

In this paper, the *sparse representation* of a polynomial $P \in \mathbb{A}[z_1, \ldots, z_n]$ consists of a sequence of exponent-coefficient pairs $(e, P_e) \in \mathbb{N}^n \times \mathbb{A}$. This sequence is sorted according to the reverse lexicographic order on the exponents, already used for the block representation.

Natural numbers in the exponents are represented by their sequences of binary digits. The total size of an exponent $e \in \mathbb{N}^n$ is written $l_e = n + \sum_{j=1}^{n} l_{e_j}$. We let $l_P = \max_{e \in \operatorname{supp} P} l_e$ for the maximum size of the exponents of P, and $s_P = |\operatorname{supp} P|$ for the number of non-zero terms of P.

Comparing or adding two exponents e and f takes $O(l_e + l_f)$ bit-operations. Therefore reading the coefficient of a given exponent e in P costs $O((l_e + l_P) \log s_P)$ bit-operations by a dichotomic search. Adding P and Q can be done with $O(s_P + s_Q)$ additions and copies in $\mathbb{A}$ plus $O((l_P + l_Q) \max(s_P, s_Q))$ bit-operations. Now consider the following algorithm for the computation of $R = PQ$:

ALGORITHM 9. Naive product for sparse polynomials.

```
1. If $s_P = 0$ then return $0$.
2. If $s_P = 1$ then return $(e + f, P_e Q_f)_{f \in \operatorname{supp} Q}$, where $e$ is the only exponent of $P$.
3. Split $P$ into $P_1$ and $P_2$ with respective sizes $h = \lceil s_P / 2 \rceil$ and $s_P - h$.
4. Compute $R_1 = P_1 Q$ and $R_2 = P_2 Q$ recursively.
5. Return $R_1 + R_2$.
```

Proposition 10. *Assuming the sparse representation of polynomials, the product* $R = P\,Q$ *can be computed using* $O(s_P\, s_Q \log \min(s_P, s_Q))$ *operations in* $\mathbb{A}$*, plus* $O((l_P + l_Q) s_P s_Q \log\min(s_P, s_Q))$ *bit-operations.*

Proof. We can assume that $s_P \leqslant s_Q$ from the outset. The number of operations in $\mathbb{A}$ is clear because the depth of the recurrence is $O(\log s_P)$. Addition of exponents only appears in the leaves of the recurrence. The total cost of step 2 amounts to $O((l_P + l_Q) s_P s_Q)$. The maximum bit-size of the exponents of the polynomials in R_1 and R_2 in step 4 never exceeds $O(l_P + l_Q)$, which yields the claimed bound. □

Remark 11. The logarithmic factor $\log\min(s_P, s_Q)$ tends to be quite pessimistic in practice. Especially in the case when $s_R \ll s_P s_Q$, the observed complexity is rather $O(s_P\, s_Q)$. Moreover, the constant corresponding to this complexity is quite small, due to the cache efficiency of the dichotomic approach.

Remark 12. For very large input polynomials it is useful to implemented an additional dichotomy on Q in order to ensure that Q fits in the cache, most of the time.

3.2. Timings

In the next tables we provide timings for $\mathbb{A} = \mathbb{Z}/p\mathbb{Z}$, with $p = 268435459$. For Table 3 we pick up $s_P = s_Q$ random monomials in the block $\{0, \ldots, 10000\}^n$, with random coefficients. Here the exponents are "packed" into a single machine word if possible. This is a classical useful trick for when the coefficients are small that was already used in [Joh74]. For $n = 2$ and $n = 3$ the exponents are packed into a 64 bits word. But for $n = 6$, the packing requires 90 bits, and thus we operate directly on the vector representation.

s_P, s_Q	10	20	40	80	160	320	640	1280
$n = 2$	0.012	0.046	0.176	0.691	2.72	13.8	59.2	263
$n = 3$	0.013	0.051	0.191	0.736	3.23	15.7	81.3	372
$n = 6$	0.022	0.105	0.472	2.14	12.4	85.4	366	1500

Table 3. Sparse polynomial product (packed exponents, in milliseconds)

In the following table we give the timings of same computations with Maple 13:

s_P, s_Q	10	20	40	80	160	320	640	1280
$n = 2$	0.172	0.321	0.927	3.22	14.9	62.5	312	1208
$n = 3$	0.205	0.389	1.13	4.15	19.2	82.2	345	1258
$n = 6$	0.261	0.502	1.58	6.07	27.4	127	511	1636

Table 4. Sparse polynomial product with Maple 13 (in milliseconds)

These timings confirm that our implementation of the naive algorithms is already competitive. In the next table we consider polynomials with increasing size in a fixed bounding hypercube $\{0, \ldots, d - 1\}^n$. The row "density" indicates the ratio of non-zero terms with exponent in $\{0, \ldots, d - 1\}^n$.

density	$n=2$, $d=80$	$n=3$, $d=40$
1%	0.428	60.9
2%	1.67	289.4
3%	4.17	675
4%	6.95	1 244
5%	12.5	2 037
6%	20.5	2 980
Kronecker	8.7	303

Table 5. Sparse polynomial product (in milliseconds) and comparison with Kronecker multiplication from Tables 1 and 2.

In Table 5, we see that the Kronecker substitution is faster from 5 % of density in the first case and 3 % in the second case. Roughly speaking, in order to determine which algorithm is faster one needs to compare $s_P s_Q$ to $\mathsf{M}(d_R)$. The efficiency of the Kronecker substitution relies on the underlying univariate arithmetic.

Let us also consider the following example with $\mathbb{A}=\mathbb{Z}$, borrowed from [Fat03]:

$$\begin{aligned} P &= (1+z_1+z_2+z_3+z_4)^{20} \\ Q &= (1+z_1+z_2+z_3+z_4)^{20}+1. \end{aligned}$$

This example has been very recently used in [MP09b] as a benchmark to compare the new implementation that will be available in MAPLE 14 to other software: in their Table 1, we see that MAPLE 14 takes 2.26 s, which is faster than TRIP, PARI-GP, MAGMA, and SINGULAR. We could not reproduce their computation, because MAPLE 14 is not available yet, but the platform we use is essentially the same, so that we could compare to our timings.

Firstly the Kronecker substitution takes 3.5 s, which is already faster than all other software they tested. The drawback of the Kronecker substitution is the memory consumption, but the direct call of the naive product takes 377 s: the coefficients of the product have at most 83 bits, and the overhead involved by GMP is important. Yet with Chinese remaindering this time drops to 16 s (see for instance Section 5 of [GG02] for a general description of this classical technique). In this situation we can chose the moduli for the Chinese remaindering such that we can add several products of the preimage in one 64 bit-word before reduction. This classical trick leads to 8 s when using primes of 27 bits. Finally the gap can be further lowered: by taking only two numbers that are coprime such as 2^{64} and $2^{32}-1$, for which the remainders are very cheap, the multiplication time is reduced to 4 s.

4. NAIVE MULTIPLICATION OF POWER SERIES

We shall consider multivariate series truncated in total degree. Such a series to order δ is represented by the sequence of its homogeneous components up to degree $\delta-1$. For each component we use a sparse representation. Let F and G be two series to order δ. The homogeneous component of degree i in F is written F_i.

4.1. Naive product

The naive algorithm for computing their product $H=FG$ works as follows:

ALGORITHM 13. Naive product for series.

```
1. Initialize $H_0 := \cdots := H_{\delta-1} := 0$.
2. For $i$ from $0$ to $\delta - 1$ do
       For $j$ from $0$ to $\delta - 1 - i$ do
           $H_{i+j} := H_{i+j} + F_i G_j$.
```

The number of non-zero terms in F is denoted by $s_F = s_{F_1} + \cdots + s_{F_{\delta-1}}$. The maximum size of the exponents of s_F is represented by $l_F = \max_{i \in \{0,\ldots,\delta-1\}} l_{F_i}$.

PROPOSITION 14. *With the above sparse representation, the product $H = FG$ can be done with $O(s_F s_G \log\min(s_F, s_G))$ operations in $\mathbb{A}$, plus $O((l_F + l_G) s_F s_G \log\min(s_F, s_G))$ bit-operations.*

Proof. By Proposition 10, the total number of operations in $\mathbb{A}$ is in

$$O\left(\sum_{d=0}^{\delta-1} \sum_{i+j=d} s_{F_i} s_{G_j} \log\min(s_{F_i}, s_{G_j})\right) = O(s_F s_G \log\min(s_F, s_G)),$$

and the total bit-cost follows in the same way. □

Remark 15. Proposition 14 is pessimistic in many cases: this cost bound is merely the one of the corresponding polynomial product discarding truncation. In the next subsection we are to take care of the truncation in terms of the dense size.

4.2. Analysis for dense series

Let $h_{i,n} = \binom{n-1+i}{n-1}$ represent the number of the monomials of degree i with n variables, and let $s_{\delta,n} = h_{0,n} + \cdots + h_{\delta-1,n} = \binom{n+\delta-1}{n}$ be the number of the monomials of degree at most $\delta - 1$. We shall consider the product in the densest situation, when $s_{F_i} = s_{G_i} = h_{i,n}$ for all i.

PROPOSITION 16. *Assuming the sparse representation of polynomials, the product $H = F\, G$ up till order δ takes $O(s_{\delta,2n} \log s_{\delta,n})$ operations in $\mathbb{A}$, plus $O(n\, l_\delta\, s_{\delta,2n} \log s_{\delta,n})$ bit-operations.*

Proof. The result follows as in proposition 14 thanks to the following identity:

$$\sum_{d=0}^{\delta-1} \sum_{i+j=d} h_{i,n} h_{j,n} = s_{\delta,2n}. \tag{1}$$

This identity is already given in [Hoe06, Section 6] for a similar purpose. We briefly recall its proof for completeness. Let $c_{d,n} = \sum_{i+j=d} h_{i,n} h_{j,n}$, let $C_n(t) = \sum_{d \geqslant 0} c_{d,n} t^n$ for the generating series, and let also $H_n(t) = \sum_{i \geqslant 0} h_{i,n} t^i$. On one hand, we have $C_n(t) = H_n(t)^2$. On the other hand, $H_n(t) = (1-t)^{-n}$. It follows that $C_n(t) = H_{2n}(t)$, and that $c_{d,n} = h_{d,2n}$, whence (1). Finally the bit-cost follows in the same way with l_F and l_G being bounded by $O(n\, l_\delta)$. □

If $\delta = 2$, then $s_{2,n} = n+1$ and $s_{2,2n} = 2n+1$, so the naive sparse product is softly optimal when n grows to infinity. If $\delta = n$ and n is large, then

$$\begin{aligned}\log s_{\delta,n} &= \log \binom{2n-1}{n} \sim (\log 4)\, n \\ \log s_{\delta,2n} &= \log \binom{3n-1}{n} \sim (\log \tfrac{27}{4})\, n.\end{aligned}$$

In particular, we observe that the naive algorithm has a subquadratic complexity in this case. In general, for fixed n and for when δ tends to infinity, the cost is quadratic since the ratio

$$\frac{s_{\delta,2n}}{s_{\delta,n}^2} = \frac{(2n+\delta-1)\cdots\delta}{(n+\delta-1)^2\cdots\delta^2} \frac{n!^2}{(2n)!}$$

tends to $\sqrt{\pi n}/4^n$.

4.3. Timings

Remark that we do not propose a specific dense representation for the series. This could be possible as we did for polynomials and gain in memory space (but not so much with respect to the "packed exponent" technique). However one can not expect more because there is no straightforward generalization of the fast algorithms for series to several variables such as the Kronecker substitution.

In Table 6 we report on timings for $\mathbb{A} = \mathbb{Z}/p\mathbb{Z}$, with $p = 268435459$. Comparing with Tables 1 and 2, we observe that, for small n, Kronecker substitution quickly becomes the most efficient strategy. However, it computes much more and its memory consumption is much higher. For small n, one could also benefit from the truncated Fourier transform, as explained in Section 6 of [Hoe06]. In higher dimensions, say $n = 6$ and order $\delta = 20$, the Kronecker substitution is hopeless: the size of the univariate polynomials to be multiplied is $40^6 \approx 4.1 \cdot 10^9$.

δ	10	20	40	80	160
$n=2$	0.05	0.4	4.8	65	926
$s_{\delta,2}$	55	210	820	3 240	12 880
$n=3$	0.28	7.0	272	13 775	∞
$s_{\delta,n}$	220	1 540	11 480	88 560	∞
$n=6$	13	9 772	∞	∞	∞
$s_{\delta,6}$	5 005	177 100	∞	∞	∞

Table 6. Dense series product (with packed exponents, in milliseconds)

5. Fast multiplication of sparse polynomials

Let $\mathbb{A}$ be an effective algebra over an effective field $\mathbb{K}$, i.e. all algebra and field operations can be performed by algorithm. Let P and Q still be two multivariate polynomials in $\mathbb{A}[z_1, \ldots, z_n]$ we want to multiply, and let $R = P\,Q$ be their product.

We assume we are given a subset $X \subseteq \mathbb{N}^n$ of size s_X that contains the support of R. We let $d_{X,1}, \ldots, d_{X,n} \in \mathbb{N}$ be the minimal numbers with $X \subseteq \prod_{j=1}^n \{0, \ldots, d_{X,j} - 1\}$. With no loss of generality we can assume that $d_{X,j} \geqslant 2$ for all j.

To analyze the algorithms of this section we shall use the quantities $e_P = \sum_{i \in \operatorname{supp} P} l_i$, $e_Q = \sum_{i \in \operatorname{supp} Q} l_i$ and $e_X = \sum_{i \in X} l_i$. We also introduce $\sigma = s_P + s_Q + s_X$ and $\epsilon = e_P + e_Q + e_X$, and $d_X = d_{X,1} \cdots d_{X,n}$.

Since the support of the product is now given, we will neglect the bit-cost due to computations on the exponents.

5.1. Evaluation, interpolation and transposition

Given t pairwise distinct points $\omega_0, \ldots, \omega_{t-1} \in \mathbb{K}^*$ and $s \in \mathbb{N}$, let E: $\mathbb{A}^s \to \mathbb{A}^t$ be the linear map which sends $(a_0, \ldots, a_{s-1})$ to $(A(\omega_0), \ldots, A(\omega_{t-1}))$, with $A(u) = a_{s-1} u^{s-1} + \cdots + a_0$. In the canonical basis, this map corresponds to left multiplication by the generalized Vandermonde matrix

$$V_{s,\omega_0,\ldots,\omega_{t-1}} = \begin{pmatrix} 1 & \omega_0 & \cdots & \omega_0^{s-1} \\ 1 & \omega_1 & \cdots & \omega_1^{s-1} \\ \vdots & \vdots & & \vdots \\ 1 & \omega_{t-1} & \cdots & \omega_{t-1}^{s-1} \end{pmatrix}.$$

The computation of E and its inverse E^{-1} (if $t = s$) correspond to the problems of multi-point evaluation and interpolation of a polynomial. Using binary splitting, it is classical [BM72, Str73, BM74] that both problems can be solved in time $O(\lceil t/s \rceil \mathsf{M}(s) \log s)$; see also [GG02, Chapter 10] for a complete description. Notice that the algorithms only require vectorial operations in $\mathbb{A}$ (additions, subtractions and multiplications with elements in $\mathbb{K}$).

The algorithms of this section rely on the efficient computations of the transpositions $E^\top, (E^{-1})^\top : (\mathbb{A}^t)^* \to (\mathbb{A}^s)^*$ of E and E^{-1}. The map $E^\top$ corresponds to left multiplication by

$$V_{s,\omega_0,\ldots,\omega_{t-1}}^\top = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \omega_0 & \omega_1 & \cdots & \omega_{t-1} \\ \vdots & \vdots & & \vdots \\ \omega_0^{s-1} & \omega_1^{s-1} & \cdots & \omega_{t-1}^{s-1} \end{pmatrix}.$$

By the transposition principle [Bor56, Ber], the operations $E^\top$ and $(E^{-1})^\top$ can again be computed in time $O(\lceil t/s \rceil \mathsf{M}(s) \log s)$.

There is an efficient direct approach for the computation of $E^\top$ [BLS03]. Given a vector $a \in (\mathbb{A}^t)^*$ with entries $a_0, \ldots, a_{t-1}$, the entries $b_0, \ldots, b_{s-1}$ of $E^\top(a)$ are identical to the first s coefficients of the power series

$$\sum_{i<t} \frac{a_i}{1 - \omega_i u}.$$

The numerator and denominator of this rational function can be computed using the classical divide and conquer technique, as described in [GG02, Algorithm 10.9]. If $t \leqslant s$, then this requires $O(\mathsf{M}(s) \log s)$ vectorial operations in $\mathbb{A}$ [GG02, Theorem 10.10]. The truncated division of the numerator and denominator at order s requires $O(\mathsf{M}(s))$ vectorial operations in $\mathbb{A}$. If $t > s$, then we cut the sum into $\lceil t/s \rceil$ parts of size $\leqslant s$, and obtain the complexity bound $O(\lceil t/s \rceil \mathsf{M}(s) \log s)$.

Inversely, assume that we wish to recover $a_0, \ldots, a_{s-1}$ from $b_0, \ldots, b_{s-1}$, when $t = s$. For simplicity, we assume that the ω_i are non-zero (this will be the case in the sequel). Setting $B(u) = b_{s-1} u^{s-1} + \cdots + b_0$, $D(u) = (1 - \omega_0 u) \cdots (1 - \omega_{s-1} u)$ and $S = B D$, we notice that $S(\omega_i^{-1}) = -a_i (u D')(\omega_i^{-1})$ for all i. Hence, the computation of the a_i reduces to two multi-point evaluations of S and $-u D'$ at $\omega_0^{-1}, \ldots, \omega_{s-1}^{-1}$ and s divisions. This amounts to a total of $O(\mathsf{M}(s) \log s)$ vectorial operations in $\mathbb{A}$ and $O(s)$ divisions in $\mathbb{K}$.

5.2. General multiplication algorithm

The Kronecker substitution K_{d_X}, introduced in Section 2.3, sends any monomial $z_1^{i_1} \cdots z_n^{i_n}$ to $x^{\mathrm{index}(i,X)}$, where $\mathrm{index}(i,X) = i_1 + i_2 d_{X,1} + \cdots + i_n d_{X,1} \cdots d_{X,n-1}$. It defines an isomorphism between polynomials with supports in X and univariate polynomials of degrees at most $d_X - 1$, so that $\mathrm{K}_{d_X}(R) = \mathrm{K}_{d_X}(P)\, \mathrm{K}_{d_X}(Q)$.

Assume now that we are given an element $\omega \in \mathbb{K}$ of multiplicative order at least d_X and consider the following evaluation map

$$\begin{array}{rcl} \mathrm{E}\colon \mathbb{A}[z] & \longrightarrow & \mathbb{A}^{s_X} \\ A & \longmapsto & (\mathrm{K}_{d_X}(A)(1), \mathrm{K}_{d_X}(A)(\omega), \ldots, \mathrm{K}_{d_X}(A)(\omega^{s_X - 1})). \end{array}$$

We propose to compute R though the equality $\mathrm{E}(R) = \mathrm{E}(P)\, \mathrm{E}(Q)$.

Given $Y = \{i_1, \ldots, i_t\} \subseteq \prod_{j=1}^n \{0, \ldots, d_{X,j} - 1\}$, let $V_{s_X,Y,\omega}$ be the matrix of E restricted to the space of polynomials with support included in Y. Setting $k_j = \mathrm{index}(i_j, X)$, we have

$$V_{s_X,Y,\omega} := V_{s_X,\omega^{k_1},\ldots,\omega^{k_t}}^{\top} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \omega^{k_1} & \omega^{k_2} & \cdots & \omega^{k_t} \\ \vdots & \vdots & & \vdots \\ \omega^{(s_X-1)k_1} & \omega^{(s_X-1)k_2} & \cdots & \omega^{(s_X-1)k_t} \end{pmatrix}.$$

Taking $Y = \mathrm{supp}\, P$, resp. $Y = \mathrm{supp}\, Q$, this allows us to compute $\mathrm{E}(P)$ and $\mathrm{E}(Q)$ using our algorithm for transposed multi-point evaluation from the preceding subsection. We obtain $\mathrm{E}(R)$ using one Hadamard product $\mathrm{E}(P)\ \mathrm{E}(Q)$. Taking $Y = X$, the points $\omega^{k_1}, \ldots, \omega^{k_t}$ are pairwise distinct, since the k_j are smaller than the order of ω. Hence $V_{s_X,X,\omega}$ is invertible and we recover R from $\mathrm{E}(R)$ using transposed multi-point interpolation.

Proposition 17. *Given two polynomials P and Q in $\mathbb{A}[z_1, \ldots, z_n]$ and an element $\omega \in \mathbb{K}$ of order at least d_X, then the product PQ can be computed using:*

- *$O(\epsilon)$ products in $\mathbb{K}$ that only depend on* $\mathrm{supp}\, P$, $\mathrm{supp}\, Q$ *and X;*
- *$O(s_X)$ inversions in $\mathbb{K}$, $O(s_X)$ products in $\mathbb{A}$, and $O\big(\frac{\sigma}{s_X}\ \mathsf{M}(s_X)\ \log\ s_X\big)$ vectorial operations in $\mathbb{A}$.*

Proof. By classical binary powering, the computation of the sequence $\omega^{d_{X,1}}, \ldots, \omega^{d_{X,1} \cdots d_{X,n-1}}$ takes $O(e_X)$ operations in $\mathbb{K}$ because each $d_{X,j} - 1$ does appear in the entries of X. Then the computation of all the $\omega^{\mathrm{index}(i,P)}$ for $i \in \mathrm{supp}\, P$ (resp. $\mathrm{supp}\, Q$ and X) requires $O(e_P)$ (resp. $O(e_Q)$ and $O(e_X)$) products in $\mathbb{K}$.

Using the complexity results from Section 5.1, we may compute $\mathrm{E}(P)$ and $\mathrm{E}(Q)$ using $O((\lceil s_P/s_X\rceil + \lceil s_Q/s_X\rceil)\,\mathsf{M}(s_X)\log s_X)$ vectorial operations in $\mathbb{A}$. We deduce $\mathrm{E}(R)$ using $O(s_X)$ more multiplications in $\mathbb{A}$. Again using the results from Section 5.1, we retrieve the coefficients R_i after $O(\mathsf{M}(s_X)\,\log\,s_X)$ further vectorial operations in $\mathbb{A}$ and $O(s_X)$ divisions in $\mathbb{K}$. Adding up the various contributions, we obtain the theorem. □

For when the supports of P and Q and also X are fixed all the necessary powers of ω can be shared and seen as a pretreatment, so that each product can be done in softly linear time. This situation occurs in the algorithm for counting the number of absolutely irreducible factors of a given polynomial, that we study in Section 7.

Similar to FFT multiplication, our algorithm falls into the general category of multiplication algorithms by evaluation and interpolation. This makes it possible to work in the so-called "transformed model" for several other operations besides multiplication.

In the rest of this section we describe how to implement the present algorithm for the usual coefficient rings and fields. We analyze the bit-cost in each case.

5.3. Finite fields

If $\mathbb{A}=\mathbb{K}$ is the finite field $\mathbb{F}_{p^k}$ with p^k elements, then its multiplicative group is cyclic of order p^k-1. Whenever $p^k-1\geqslant d_X$, Proposition 17 applies for any primitive element ω of this group. We assume that $\mathbb{F}_{p^k}$ is given as the quotient $\mathbb{F}_p[u]/G(u)$ for some monic and irreducible polynomial G of degree k.

Corollary 18. *Assume $p^k-1\geqslant d_X$, and assume given a primitive element ω of $\mathbb{F}_{p^k}^*$. Given $P, Q\in\mathbb{F}_{p^k}[z_1,\ldots,z_n]$, the product PQ can be computed using*

- $O(\epsilon\,\mathsf{M}(k))$ *ring operations in $\mathbb{F}_p$ that only depend on* $\operatorname{supp}P$, $\operatorname{supp}Q$ *and X;*
- $O\Big(\frac{\sigma}{s_X}\,\mathsf{M}(s_X\,k)\,\log\,s_X + s_X\,\mathsf{M}(k)\,\log\,k\Big)$ *ring operations in $\mathbb{F}_p$ and $O(s_X)$ inversions in $\mathbb{F}_p$.*

Proof. A multiplication in $\mathbb{F}_{p^k}$ amounts to $O(\mathsf{M}(k))$ ring operations in $\mathbb{F}_p$. An inversion in $\mathbb{F}_{p^k}$ requires an extended g.c.d. computation in $\mathbb{F}_p[u]$ and gives rise to $O(\mathsf{M}(k)\log k)$ ring operations in $\mathbb{F}_p$ and one inversion: this can be achieved with the fast Euclidean algorithm [GG02, Chapter 11], with using pseudo-divisions instead of divisions. Using the Kronecker substitution, one product in $\mathbb{F}_{p^k}[u]$ in size n takes $O(\mathsf{M}(n\,k))$ operations in $\mathbb{F}_p$. The conclusion thus follows from Proposition 17. □

Applying the general purpose algorithm from [CK91], two polynomials of degree n over $\mathbb{F}_p$ can be multiplied in time $O(\mathsf{I}(\log p)\,n\log n\log\log n)$. Alternatively, we may lift the multiplicands to polynomials in $\mathbb{Z}[u]$, use Kronecker multiplication and reduce modulo p. As long as $\log\,n=O(\log\,p)$, this yields the better complexity bound $O(\mathsf{I}(n\log p))$. Corollary 18 therefore further implies:

Corollary 19. *Assume $p^k-1\geqslant d_X$ and $\log(s_X k)=O(\log p)$, and assume given a primitive element ω of $\mathbb{F}_{p^k}^*$. Given two polynomials P and Q in $\mathbb{F}_{p^k}[z_1,\ldots,z_n]$, the product PQ can be computed using*

- $O(\epsilon\,\mathsf{I}(k\log p))$ *bit-operations that only depend on* $\operatorname{supp}P$, $\operatorname{supp}Q$ *and X;*

- $O\!\left(\frac{\sigma}{s_X}\,\mathsf{I}(s_X\,k\,\log\,p)\,\log\,s_X+s_X\,\mathsf{I}(k\,\log\,p)\,\log\,k+s_X\,\mathsf{I}(\log\,p)\,\log\log\,p\right)$ *bit-operations.*

If $p^k-1<d_X$ then it is always possible to build an algebraic extension of suitable degree l in order to apply the corollary. Such constructions are classical, see for instance [GG02, Chapter 14]. We need to have $p^{kl}-1\geqslant d_X$, so l should be taken of the order $\log_{p^k} d_X$, which also corresponds to the additional overhead induced by this method. If $\log_{p^k} d_X$ exceeds $O(\log\sigma)$ and $O(\log(k\log p))$, then we notice that the softly linear cost is lost. This situation may occur for instance for polynomials over $\mathbb{F}_2$.

In practice, the determination of the primitive element ω is a precomputation that can be done fast with randomized algorithms. Theoretically speaking, assuming the generalized Riemann hypothesis, and given the prime factorization of p^k-1, a primitive element in $\mathbb{F}_{p^k}^*$ can be constructed in polynomial time [BS91, Section 1, *Applications*].

5.4. Integer coefficients

Let $h_P=\max_i l_{|P_e|}$ denote the maximal bit-size of the coefficients of P and similarly for Q and R. Since

$$\max_e |R_e|\leqslant \min(s_P,s_Q)\max_e|P_e|\max_e|Q_e|,$$

we have

$$h_R\leqslant h:=h_P+h_Q+l_{\min(s_P,s_Q)}.$$

5.4.1. Big prime algorithm

One approach for the multiplication $R=PQ$ of polynomials with integer coefficients is to reduce the problem modulo a suitable prime number p. This prime number should be sufficiently large such that R can be read off from R mod p and such that $\mathbb{F}_p$ admits elements of order at least d_X. It suffices to take $p>\max(2^{h+1},d_X)$, so Corollary 19 now implies:

COROLLARY 20. *Given $P,Q\in\mathbb{Z}[z_1,\ldots,z_n]$, a prime number $p>\max(2^{h+1},d_X)$ and a primitive element ω of $\mathbb{F}_p^*$, we can compute PQ with*

- $O(\epsilon\,\mathsf{I}(\log p))$ *bit-operations that only depend on* $\operatorname{supp}P$, $\operatorname{supp}Q$, *and X;*
- $O\!\left(\frac{\sigma}{s_X}\,\mathsf{I}(s_X\log p)\log s_X+s_X\,\mathsf{I}(\log p)\log\log p\right)$ *bit-operations.*

Let p_n denote the n-th prime number. The prime number theorem implies that $p_n\asymp n\log n$. Cramér's conjecture [Cra36, Sha64] states that

$$\limsup_{n\to\infty}\frac{p_{n+1}-p_n}{(\log p_n)^2}=1.$$

This conjecture is supported by numerical evidence, which is sufficient for our practical purposes. Setting $N=\max(2^{h+1},d_X)$, the conjecture implies that the smallest prime number p with $p>N$ satisfies $p=N+O(\log^2 N)$. Using a polynomial time primality test [AKS04], it follows that this number can be computed by brute force in time $(\log N)^{O(1)}$. In practice, in order to satisfy the complexity bound it suffices to tabulate prime numbers of sizes 2, 4, 8, 16, etc.

5.4.2. Chinese remaindering

In our algorithm and Corollary 20, we regard the computation of a prime number $p > N = \max(2^{h+1}, d_X)$ as a precomputation. This is reasonable if N is not too large. Now the quantity $\log d_X$ usually remains reasonably small. Hence, our assumption that N is not too large only gets violated if $h_P + h_Q$ becomes large. In that case, we will rather use Chinese remaindering. We first compute $r = O(\lceil h/\log d_X \rceil)$ prime numbers $p_1 < \cdots < p_r$ with

$$\begin{aligned} p_1 &> d_X, \\ p_1 \cdots p_r &> 2^{h+1}. \end{aligned}$$

Such a sequence is said to be a *reduced sequence of prime moduli* with order d_X and capacity 2^{h+1}, if, in addition, we have that $\log p = O(h + \log d_X)$, where $p = p_1 \cdots p_r$.

In fact Bertrand's postulate [HW79, Chapter 12, Theorem 1.3] ensures us that there exists p_1 between $d_X + 1$ and $2\,d_X$, then one can take p_2 between $p_1 + 1$ and $2\,p_1$, etc, so that $\log p_r = O(\log d_X + r)$. We stop this construction with $p_1 \cdots p_{r-1} \leqslant 2^{h+1}$ and $p_1 \cdots p_r > 2^{h+1}$, hence with $\log\ p = O(h + \log\ d_X)$. This proves that such reduced sequences actually exist. Of course Bertrand's postulate is pessimistic, and in practice all the p_k are very close to d_X.

Each $\mathbb{F}_{p_k}$ contains a primitive root of unity ω_k of order at least d_X. We next proceed as before, but with $p = p_1 \cdots p_r$ and $\omega \in \mathbb{Z}/p\,\mathbb{Z}$ such that $\omega \bmod p_k = \omega_k$ for each k. Indeed, even though $\mathbb{Z}/p\,\mathbb{Z}$ is not a field, the fact that each $V_{s_X,X,\omega} \bmod p_k = V_{s_X,X,\omega_k}$ is invertible implies that $V_{s_X,X,\omega}$ is invertible, which is sufficient for our purposes.

Corollary 21. *Given $P, Q \in \mathbb{Z}[z_1, \ldots, z_n]$, a reduced sequence $p_1 < \cdots < p_r$ of prime moduli with order d_X and capacity 2^{h+1} and an element $\omega \in \mathbb{Z}/p_1 \cdots p_r\,\mathbb{Z}$ of order at least d_X, we can compute PQ with*

- $O(\epsilon\,\mathsf{I}(\log p))$ *bit-operations that only depend on* $\operatorname{supp} P$, $\operatorname{supp} Q$, *and* X;
- $O\Big(\frac{\sigma}{s_X}\,\mathsf{I}(s_X \log p) \log s_X + s_X\,\mathsf{I}(\log p) \log\log p\Big)$ *bit-operations.*

Proof. This follows from Proposition 17, following the proofs of Corollaries 18 and 19, *mutatis mutandis.* □

Whenever $\log d_X = O(h)$ we have that $\log p = O(h)$. Therefore, for fixed supports of P and Q, and fixed X, this method allows us to compute several products in softly linear time. Remark that for moderate sizes of the coefficients it is even more interesting to compute the products modulo each p_k in parallel, and then use Chinese remaindering to reconstruct the result.

5.5. Floating point coefficients

An important kind of sparse polynomials are power series in several variables, truncated by total degree. Such series are often used in long term integration of dynamical systems [MB96, MB04], in which case their coefficients are floating point numbers rather than integers. Assume therefore that P and Q are polynomials with floating coefficients with a precision of ℓ bits.

Let ξ_P be the maximal exponent of the coefficients of P. For a so called *discrepancy* $\eta_P \in \mathbb{N}$, fixed by the user, we let $\hat{P}$ be the integer polynomial with

$$\hat{P}_i = \lfloor P_i 2^{\ell + \eta_P - \xi_P} \rceil$$

for all i. We have $l_{\hat{P}} \leqslant \ell + \eta_P$ and

$$|P - \hat{P}\, 2^{\xi_P - \ell - \eta_P}| \leqslant 2^{\xi_P - \ell - \eta_P - 1}$$

for the sup-norm on the coefficients. If all coefficients of P have a similar order of magnitude, in the sense that the minimal exponent of the coefficients is at least $\xi_P - \eta_P$, then we actually have $P = \hat{P}\, 2^{\xi_P - \ell - \eta_P}$. Applying a similar decomposition to Q, we may compute the product

$$PQ = \hat{P}\hat{Q}\, 2^{\xi_P + \xi_Q - 2\ell - \eta_P - \eta_Q}$$

using the preceding algorithms and convert the resulting coefficients back into floating point format.

Usually, the coefficients f_i of a univariate power series $f(z)$ are approximately in a geometric progression $\log f_i \sim \alpha\, i$. In that case, the coefficients of the power series $f(\lambda\, z)$ with $\lambda = \mathrm{e}^{-\alpha}$ are approximately of the same order of magnitude. In the multivariate case, the coefficients still have a geometric increase on diagonals $\log f_{\lfloor k_1 i \rfloor, \ldots, \lfloor k_n i \rfloor} \sim \alpha_{k_1, \ldots, k_n}\, i$, but the parameter $\alpha_{k_1, \ldots, k_n}$ depends on the diagonal. After a suitable change of variables $z_i \mapsto \lambda_i\, z_i$, the coefficients in a big zone near the main diagonal become of approximately the same order of magnitude. However, the discrepancy usually needs to be chosen proportional to the total truncation degree in order to ensure sufficient accuracy elsewhere.

5.6. Rational coefficients

Let us now consider the case when $\mathbb{K} = \mathbb{Q}$. Let q_P and q_Q denote the least common multiples of the denominators of the coefficients of P resp. Q. One obvious way to compute PQ is to set $\hat{P} := P q_P$, $\hat{Q} := Q\, q_Q$, and compute $\hat{P}\hat{Q}$ using one of the methods from Section 5.4. This approach works well in many cases (e.g. when P and Q are truncations of exponential generating series). Unfortunately, this approach is deemed to be very slow if the size of q_P or q_Q is much larger than the size of any of the coefficients of PQ.

An alternative, more heuristic approach is the following. Let $p_1 < p_2 < \cdots$ be an increasing sequence of prime numbers with $p_1 > d$ and such that each p_i is relatively prime to the denominators of each of the coefficients of P and Q. For each i, we may then multiply $P \bmod p_i$ and $Q \bmod p_i$ using the algorithm from Section 4. For $i = 1, 2, 4, 8, \ldots$, we may recover $P\,Q \bmod p_1 \cdots p_i$ using Chinese remaindering and attempt to reconstruct $P\,Q$ from $P\,Q \bmod p_1 \cdots p_i$ using rational number reconstruction [GG02, Chapter 5]. If this yields the same result for a given i and $2\,i$, then the reconstructed $P\,Q$ is likely to be correct at those stages. This strategy is well-suited to probabilistic algorithms, for polynomial factorization, polynomial system solving, etc.

Of course, if we have an *a priori* bound on the bit sizes of the coefficients of R, then we may directly take a sufficient number of primes $p_1 < \cdots < p_r$ such that R can be reconstructed from its reduction modulo $p_1 \cdots p_r$.

5.7. Timings

We illustrate the performances of the algorithms of this section for a prime finite field, which is the central case to optimize. We take $\mathbb{A} = \mathbb{Z}/p\mathbb{Z}$, with $p = 23058430\backslash 09213693967 < 2^{62}$. If the size of the product is of the order of $s_P s_Q$ then the naive algorithm is already softly optimal. If the polynomials are close to being dense then the Kronecker substitution is the most efficient in practice. Here we consider a case which lies in between these two extremes.

More precisely, we pick polynomials with terms of total degree at most δ and at least $\delta - 4$ with random coefficients in $\mathbb{F}_p$. The subset X can be easily taken as the set of the monomials of degree at most $2\,\delta$ and at least $2\,\delta - 8$. In Table 7 we compare the fast algorithm of Section 5.3 to the naive one of Section 3 and the direct use of the Kronecker substitution.

δ	20	40	80	160	320	640	1280	2560	5120
naive	0.34	1.2	5.0	20	81	326	1 317	5 457	22 855
Kronecker	0.60	3.5	17	85	456	2 408	10 664	48 765	∞
fast	31	60	123	267	597	1 352	3 064	6 918	15 615
s_X	266	546	1 106	2 226	4 466	8 946	17 906	35 826	71 666

Table 7. Polynomial product with 2 variables of two strips from $\delta - 3$ to δ (in milliseconds)

With 2 (and also with 3 variables) the theoretical asymptotic behaviours are already well reflected. But the fast algorithm only gains for very large sizes. When sharing the same supports in several product the benefit of the fast product can be observed earlier. We shall illustrate this situation in Section 7, for a similar family of examples.

6. Fast multiplication of power series

In this section, we show how to build a multiplication algorithm for formal power series on top of the fast polynomial product from the previous section. We will only consider power series which are truncated with respect to the total degree.

6.1. Total degree

Given $i \in \mathbb{N}^n$, we let $|i| = i_1 + \cdots + i_n$. The *total degree* of a polynomial $P \in \mathbb{A}[z]$ is defined by $\deg P = \max \{|i| : P_i \neq 0\}$. Given a subset $I \subseteq \mathbb{N}^n$, we define the *restriction* P_I of P to I by

$$P_I = \sum_{i \in I} P_i\, z^i.$$

For $\delta \in \mathbb{N}$, we define the initial segments I_δ of $\mathbb{N}^n$ by $I_\delta = \{i \in \mathbb{N}^n : |i| < \delta\}$. Then

$$\mathbb{A}[z]_{I_\delta} = \mathbb{A}[[z]]_{I_\delta} = \{P \in \mathbb{A}[z] : \operatorname{supp} P \subseteq I_\delta\} = \{P_{I_\delta} : P \in \mathbb{A}[z]\}$$

is the set of polynomials of total degree $< \delta$. Given P, $Q \in \mathbb{A}[z]_{I_\delta}$, the aim of this section is to describe efficient algorithms for the computation of $R = (PQ)_{I_\delta}$. We will follow and extend the strategy described in [LS03].

Remark 22. The results of this section could be generalized in the same way as in [Hoe02] to so called weighted total degrees $|i| = \lambda_1 i_1 + \cdots + \lambda_n i_n$ with $\lambda_1, \ldots, \lambda_n > 0$, but for the sake of simplicity, we will stick to ordinary total degrees.

6.2. Projective coordinates

Given a polynomial $P \in \mathbb{A}[z]$, we define its *projective transform* $\mathrm{T}(P) \in \mathbb{A}[z]$ by

$$\mathrm{T}(P)(z_1, \ldots, z_n) = P(z_1 z_n, \ldots, z_{n-1} z_n, z_n).$$

If $\operatorname{supp} P \subseteq I_\delta$, then $\operatorname{supp} \mathrm{T}(P) \subseteq J_\delta$, where

$$J_\delta \;=\; \{(i_1 + i_\delta, \ldots, i_{\delta-1} + i_\delta, i_\delta) : i \in I_\delta\}.$$

Inversely, for any $P \in \mathbb{A}[z]_{J_\delta}$, there exists a unique $\mathrm{T}^{-1}(P) \in \mathbb{A}[z]_{I_\delta}$ with $P = \mathrm{T}(\mathrm{T}^{-1}(P))$. The transformation T is an injective morphism of $\mathbb{A}$-algebras. Consequently, given $P, Q \in \mathbb{A}[z]_{I_\delta}$, we will compute the truncated product $(PQ)_{I_\delta}$ using

$$(PQ)_{I_\delta} = \mathrm{T}^{-1}((\mathrm{T}(P)\,\mathrm{T}(Q))_{J_\delta}).$$

Given a polynomial $P \in \mathbb{A}[z]$ and $j \in \mathbb{N}$, let

$$P_j = \sum_{i_1, \ldots, i_{n-1}} P_{i_1, \ldots, i_{n-1}, j}\, z_1^{i_1} \cdots z_{n-1}^{i_{n-1}} z_n^j \in \mathbb{Z}[z_1, \ldots, z_{n-1}].$$

If $\operatorname{supp} P \subseteq J_\delta$, then $\operatorname{supp} P_j \subseteq X$, with

$$X \;=\; \{i \in \mathbb{N}^{n-1} : i_1 + \cdots + i_{n-1} < \delta\}.$$

6.3. Multiplication by evaluation and interpolation

Let ω be an element of $\mathbb{K}$ of order at least δ^{n-1}. Taking X as above, the construction in Section 5.2 yields a $\mathbb{K}$-linear and invertible evaluation mapping

$$\mathrm{E} : \mathbb{A}[z]_X \longrightarrow \mathbb{A}^X,$$

such that for all $P, Q \in \mathbb{A}[z]_X$ with $PQ \in \mathbb{A}[z]_X$, we have

$$\mathrm{E}(PQ) = \mathrm{E}(P)\,\mathrm{E}(Q). \tag{2}$$

This map extends to $\mathbb{A}[z]_X[z_n]$ using

$$\mathrm{E}(P_0 + \cdots + P_k z_n^k) \;=\; \mathrm{E}(P_0) + \cdots + \mathrm{E}(P_k)\, z_n^k \in \mathbb{A}^X[z_n].$$

Given $P, Q \in \mathbb{A}[z]_{J_\delta}$ and $j < \delta$, the relation (2) yields

$$\mathrm{E}((PQ)_j) = \mathrm{E}(P_j)\,\mathrm{E}(Q_0) + \cdots + \mathrm{E}(P_0)\,\mathrm{E}(Q_j).$$

In particular, if $R = (PQ)_{J_\delta}$, then

$$\mathrm{E}(R) = \mathrm{E}(P)\,\mathrm{E}(Q) \bmod z_n^\delta.$$

Since E is invertible, this yields an efficient way to compute R.

6.4. Complexity analysis

The number of coefficients of a truncated series in $\mathbb{A}[[z]]_{I_\delta}$ is given by

$$|I_\delta| = s_{\delta,n} = \binom{n+\delta-1}{n}.$$

The size $s_X = |X|$ of X is smaller by a factor between 1 and δ:

$$s_X = \binom{n+\delta-2}{n-1} = \frac{n}{n+\delta-1} |I_\delta|.$$

Proposition 23. *Given $P, Q \in \mathbb{A}[[z]]_{I_\delta}$ and an element $\omega \in \mathbb{K}$ of order at least δ^{n-1}, we can compute $(PQ)_{I_\delta}$ using $O(s_X \delta)$ inversions in $\mathbb{K}$, $O(s_X \mathsf{M}(\delta))$ ring operations in $\mathbb{A}$, and $O(\delta \mathsf{M}(s_X) \log s_X)$ vectorial operations in $\mathbb{A}$.*

Proof. We apply the algorithm described in the previous subsection. The transforms T and T^{-1} require a negligible amount of time. The computation of the evaluation points ω^{k_i} only involves $O(s_X)$ products in $\mathbb{K}$, when exploiting the fact that X is an initial segment. The computation of $\mathrm{E}(\mathrm{T}(P))$ and $\mathrm{E}(\mathrm{T}(Q))$ requires $O(\delta \mathsf{M}(s_X) \log s_X)$ vectorial operations in $\mathbb{A}$, as recalled in Section 5.1. The computation of $\mathrm{E}(\mathrm{T}(P))\, \mathrm{E}(\mathrm{T}(Q)) \bmod z_n^\delta$ can be done using $O(s_X \mathsf{M}(\delta))$ ring operations in $\mathbb{A}$. Recovering R again requires $O(\delta \mathsf{M}(s_X) \log s_X)$ vectorial operations in $\mathbb{A}$, as well as $O(s_X \delta)$ inversions in $\mathbb{K}$. □

Remark 24. Since $s_X \delta = \tilde{O}(s_{\delta,n})$ by [LS03, Lemma 3], Proposition 23 ensures that power series can be multiplied in softly linear time, when truncating with respect to the total degree.

6.5. Finite fields

In the finite field case when P, $Q \in \mathbb{F}_{p^k}[[z]]_{I_\delta}$, the techniques from Section 5.3 lead to the following consequence of Proposition 23.

Corollary 25. *Assume $n \geqslant 2$, $p^k - 1 \geqslant \delta^{n-1}$ and $\log (s_X k) = O(\log p)$, and assume given a primitive element ω of $\mathbb{F}_{p^k}^*$. Given $P, Q \in \mathbb{F}_{p^k}[[z]]_{I_\delta}$, we can compute $(PQ)_{I_\delta}$ using*

$$O(\delta \mathsf{I}(s_X k \log p) \log s_X + s_X \delta (\mathsf{I}(k \log p) \log k + \mathsf{I}(\log p) \log \log p))$$

bit-operations.

Proof. The $s_X \delta$ inversions in $\mathbb{F}_{p^k}$ take $O(s_X \delta (\mathsf{I}(k \log p) \log k + \mathsf{I}(\log p) \log \log p))$ bit-operations, when using Kronecker substitution. The $O(\delta\ \mathsf{M}(s_X)\ \log\ s_X)$ ring operations in $\mathbb{F}_{p^k}$ amount to $O(\delta \mathsf{I}(s_X k \log p) \log s_X)$ more bit-operations. Since $n \geqslant 2$ we have $\delta = O(s_X)$, which implies $s_X \mathsf{M}(\delta) = O(\delta \mathsf{M}(s_X))$. The conclusion thus follows from Proposition 23. □

If $p^k - 1 \geqslant \delta^{n-1}$ does not hold, then it is possible to perform the product in an extension of degree $r = 1 + \lfloor \log \delta^{n-1} / \log p^k \rfloor$ of $\mathbb{F}_{p^k}$, so that $(p^k)^r - 1 \geqslant \delta^{n-1}$. Doing so, the cost of the product requires $\tilde{O}(s_X\, \delta)$ operations in $\mathbb{F}_{(p^k)^r}$, which further reduces to $\tilde{O}(s_{\delta,n})$ by [LS03, Lemma 3]. The field extension and the construction of the needed ω can be seen as precomputations for they only depend on n, δ, p and k. Since $r = O(n \log s_{\delta,n})$, we have therefore obtained a softly optimal uniform complexity bound in the finite field case.

Notice that the direct use of Kronecker substitution for multiplying P and Q yields $\tilde{O}(k\,(2\,\delta - 1)^n)$ operations in $\mathbb{F}_p$. In terms of the dense size of P and Q, the latter bound becomes of order $\tilde{O}(2^n\; n!\; k\; s_{\delta,n})$, which is more expensive than the present algorithm.

6.6. Integer coefficients

If $P, Q \in \mathbb{Z}[[z]]_{I_\delta}$, then the assumption $p \geqslant 2^{h+1}$, with $h = h_P + h_Q + l_{s_{\delta,n}}$, guarantees that the coefficients of the result $P Q$ can be reconstructed from their reductions modulo p. If $2^{h+1} < \delta^{n-1}$, and if we are given a prime number $p \in [2^{h+1}, 2^{h+2})$ and a primitive element ω of $\mathbb{F}_{p^r}^*$, where $r = 1 + \lfloor \log \delta^{n-1} / \log p^k \rfloor$, then $(P Q)_{I_\delta}$ can be computed using $\tilde{O}(n\, h\, s_{\delta,n})$ bit-operations, by Corollary 25. Otherwise, in the same way we did for polynomials in Section 5.4, Chinese remaindering leads to:

Corollary 26. *Assume that $n \geqslant 2$, that $2^{h+1} \geqslant \delta^{n-1}$, and that we are given a reduced sequence $p_1 < \cdots < p_r$ of prime moduli with order δ^{n-1} and capacity 2^{h+1}, and an element $\omega \in \mathbb{Z}/p_1 \cdots p_r \mathbb{Z}$ of order at least δ^{n-1}. Given $P, Q \in \mathbb{Z}[[z]]_{I_\delta}$, we can compute $(PQ)_{I_\delta}$ using*

$$O(\delta\, \mathsf{I}(s_X\, h) \log s_X + s_X\, \delta\, \mathsf{I}(h) \log h)$$

bit-operations.

Proof. Let $p = p_1 \cdots p_r$. Since $\log s_X = O(\log p)$, we can use the Kronecker substitution with the algorithm underlying Proposition 23 over $\mathbb{Z}/p\,\mathbb{Z}$ to perform the truncated product of P and Q with $O(\delta\, \mathsf{I}(s_X \log p) \log s_X + s_X\, \delta\, \mathsf{I}(\log p) \log \log p)$ bit-operations. The conclusion thus follows from $\log p = O(h)$. □

Remark that the bound in Corollary 26 is softly optimal, namely $\tilde{O}(h\, s_{\delta,n})$, which is much better than a direct use of Kronecker substitution when n becomes large.

6.7. Timings

We take $\mathbb{A} = \mathbb{Z}/p\mathbb{Z}$, with $p = 268435459$. The following tables demonstrate that the the theoretical *softly linear* asymptotic costs can really be observed.

δ	10	20	40	80	160
$n = 2$	1.5	8.1	53	232	1 151
$n = 3$	13	118	1 121	12 248	173 122

Table 8. Fast series product (in milliseconds)

When comparing Tables 8 and 6, we see that our fast product does not compete with the naive algorithm up to order 160 in the bivariate case. For three variables we observe that it outperforms the naive approach at large orders, but that it is still slower than Kronecker substitution (see Table 2).

δ	10	20	40	60
naive	0.002	0.11	14	184
Kronecker	0.6	1.7	40	∞
fast	0.05	0.8	15	94

Table 9. Series products with 4 variables (in seconds)

For 4 variables we see in Table 9 that the Kronecker product is slower than the naive approach (it also consumes a huge quantity of memory). The naive algorithm is fastest for small orders, but our fast algorithm wins for large orders.

For 5 and more variables the truncation order can not grow very large, and the naive algorithm becomes subquadratic, so that the threshold for the fast algorithm is much higher:

δ	5	10	15	20	25	30
naive	0.0001	0.004	0.09	0.9	5.6	26
Kronecker	0.05	3.7	36	∞	∞	∞
fast	0.06	0.4	2.4	10	32	85

Table 10. Series products with 5 variables (in seconds)

7. Absolute factorization

Let $\mathbb{K}$ be a field, and let $F \in \mathbb{K}[x_1, \ldots, x_n, y] = \mathbb{K}[x, y]$. In this section we are interested to count the number of the *absolutely irreducible factors* of F, i.e. number of irreducible factors of F over the algebraic closure $\bar{\mathbb{K}}$ of $\mathbb{K}$.

7.1. Reduction to linear algebra

Let $d_x = \deg_x F$ denote the total degree in the variables x, and let $d_y = \deg_y F$. The *integral hull* of $\operatorname{supp} F$, which we will denote by S, is the intersection of $\mathbb{Z}^{n+1}$ and the convex hull of $\operatorname{supp} F$ as a subset of $\mathbb{R}^{n+1}$. The method we are to use is not new, but combined with our fast algorithms of Section 5, we obtain a new complexity bound, essentially (for fixed values of n) quadratic in terms of the size of S .

Besides the support of F, we need to introduce

$$\begin{aligned} S_x &= S \cap ((\mathbb{N}^n \setminus \{(0, \ldots, 0)\}) \times \mathbb{N}) \\ S_y &= S \cap (\mathbb{N}^n \times \mathbb{N} \setminus \{0\}) \\ S_{x,y} &= S_x \cap S_y \\ T &= (S_{x,y} + S) \cup (S_x + S_y). \end{aligned}$$

Notice that S_y consists of the elements $e \in S$ with $e_{n+1} > 0$. The set S_x contains the support of

$$\theta_x F = \sum_{(e,f) \in \operatorname{supp} F} (e_1 + \cdots + e_n) F_{e,f} x^e y^f,$$

the set S_y contains the support of

$$\theta_y F = y \frac{\partial F}{\partial y} = \sum_{(e,f) \in \operatorname{supp} F} f F_{e,f} x^e y^f,$$

and $S_{x,y}$ contains $\operatorname{supp} \theta_x \theta_y F$.

The absolute factorization of F mainly reduces to linear algebra by considering the following map:

$$\begin{array}{rcl} D_F \colon \mathbb{K}[x,y]_{S_y} \times \mathbb{K}[x,y]_{S_x} & \longrightarrow & \mathbb{K}[x,y]_T \\ (G,H) & \longmapsto & G\,\theta_x F - H\,\theta_y F - (\theta_x G - \theta_y H) F, \end{array}$$

where $\mathbb{K}[x,y]_{S_y}$ represents the subset of the polynomials with support in S_y (and similarly for S_x, $S_{x,y}$, and T).

Proposition 27. *Assume that $\mathbb{K}$ has characteristic 0 or at least $d_x(2 d_y - 1) + 1$, that F is primitive in y, and that the discriminant of F in y is non-zero. Then the number of the absolutely irreducible factors of F equals the dimension of the kernel of D_F.*

Proof. This result is not original, but for a lack of an exact reference in the literature, we provide the reader with a sketch of the proof adapted from the bivariate case. Let $\varphi_1, \ldots, \varphi_{d_y}$ represent the distinct roots of F in $\overline{\mathbb{K}(x)}$. The assumption on the discriminant of F ensures that all are simple. Now consider the partial fraction decompositions of G/F and H/F:

$$\frac{G}{F} = y \sum_{i=1}^{d_y} \frac{\rho_i}{y - \varphi_i}, \quad \frac{H}{F} = c(x) + \sum_{i=1}^{d_y} \frac{\sigma_i}{y - \varphi_i},$$

where ρ_i and σ_i belong to $\overline{\mathbb{K}(x)}$ and $c(x) \in \mathbb{K}(x)$. The fact that $D_F(G,H) = 0$ is equivalent to

$$\theta_x\left(\frac{G(x,y)}{F(x,y)}\right) = \theta_y\left(\frac{H(x,y)}{F(x,y)}\right),$$

which rewrites into:

$$y \sum_{i=1}^{d_y} \left(\frac{\theta_x(\rho_i)}{y - \varphi_i} + \frac{\rho_i\,\theta_x(\varphi_i)}{(y - \varphi_i)^2}\right) = -\,y \sum_{i=1}^{d_y} \frac{\sigma_i}{(y - \varphi_i)^2}.$$

Therefore $\theta_x(\rho_i)$ must vanish for all i. In characteristic 0, this implies that the ρ_i actually belong to $\bar{\mathbb{K}}$. If the characteristic is least $d_x(2 d_y - 1) + 1$ this still holds by the same arguments as in Lemma 2.4 of [Gao03]. Let $F_1, \ldots, F_r$ denote the absolutely irreducible factors of F, and let $\hat{F}_i = F/F_i$. By applying classical facts on partial fraction decomposition, such as [GG02, Theorem 22.8] or [CL07, Appendix A] for instance, we deduce that G is a linear combination of the $\hat{F}_i\,\theta_y F_i$, hence that (G, H) belongs to the space spanned by the $(\hat{F}_i\,\theta_y F_i, \hat{F}_i\,\theta_x F_i)$ over $\bar{\mathbb{K}}$, for $i \in \{1, \ldots, r\}$.

Since $\operatorname{supp}(\hat{F}_i\,\theta_x F_i) \subseteq S_x$ and $\operatorname{supp}(\hat{F}_i\,\theta_y F_i) \subseteq S_y$, the couples $(\hat{F}_i\,\theta_y F_i, \hat{F}_i\,\theta_x F_i)$ form a basis of the kernel of D_F over $\bar{\mathbb{K}}$, which concludes the proof. □

Proposition 27 was first stated in [Gao03] in the bivariate case for the dense representation of the polynomials, and then in terms of the actual support of F in [GR03] but still for two variables. For several variables and block supports, generalizations have been proposed in [GKM+04, Remark 2.3] but they require computing the partial derivatives in all the variables separately, which yields more linear equations to be solved than with the present approach. Let us recall that the kernel of D_F is nothing else than the first De Rham cohomology group of the complementary of the hypersurface defined by F (this was pointed out in [Lec07], we refer the reader to [Sha09] for the details).

For a complete history of the algorithms designed for the absolute factorization we refer the reader to the introduction of [CL07]. In fact, the straightforward resolution of the linear system defined by $D_F(G,H)=0$ by Gaussian elimination requires $O(s^{\omega-1}t)$ operations in $\mathbb{K}$, where $s=|S_x|+|S_y|$ is the number of the unknowns and $t=|T|\geqslant s$ is the number of equations [Sto00, Proposition 2.11]. Here ω is a real number at most 3 such that the product of two matrices of size $s\times s$ can be done with $O(s^{\omega})$ operations in $\mathbb{K}$. In practice ω is close to 3, so that Gaussian elimination leads to a cost more than quadratic.

In [Gao03], Gao suggested to compute the kernel of D_F using Wiedemann's algorithm: roughly speaking this reduces to compute the image by D_F of at most $2\ |2\ S|$ vectors. With a block support, and for when the dimension is fixed, the Kronecker substitution can be used so that a softly quadratic cost can be achieved in this way. In the next subsection we extend this idea for general supports by using the fast polynomial product of Section 5.

7.2. Algorithm

The algorithm we propose for computing the number of the absolutely irreducible factors of F summarizes as follows:

Algorithm 28. Probable number of absolutely irreducible factors *of F.*

1. *Compute the integral hull S of the support of F. Deduce S_x, S_y, $S_{x,y}$, and T.*
2. *Pre-compute all the intermediate data necessary to the evaluation of D_F by means of the fast polynomial product of Section 5.*
3. *Compute the dimension of the kernel of D_F with the randomized algorithm of [KS91, Section 4], all the necessary random values being taken in a given subset $\mathcal{S}$ of* $\mathbb{K}$.

For simplicity, the following complexity analysis will not take into account the bit-cost involved by operations with the exponents and the supports.

Proposition 29. *Assume that* $\mathbb{K}$ *has characteristic 0 or at least $d_x(2\,d_y-1)+1$, that F is primitive in y, that the discriminant of F in y is non-zero, that we are given an element ω in* $\mathbb{K}$ *of order at least $(2\,d_y+1)\prod_{i=1}^{n}(2\deg_{x_i}F+1)$, and that the given set $\mathcal{S}$ contains at least $5\,|2\,S|-2$ elements.*

Then Algorithm 28 performs the computation of the integral hull S of $|\operatorname{supp}F|$ points of bit-size at most l_F, plus the computation of T, plus $\tilde{O}(|2\ S|^2)$ operations in $\mathbb{K}$. *It returns a correct answer with a probability at least $1-\frac{3}{2}|2\,S|(|2\,S|+1)/|\mathcal{S}|$.*

Proof. Since T is included in $2\,S$, the assumption on the order of ω allows us to apply Proposition 17. In the second step, we thus compute all the necessary powers of ω to evaluate D_F: because the supports are convex, this only amounts to $O(|2\,S|)$ operations in $\mathbb{K}$. Then for any couple $(G, H) \in \mathbb{K}[x, y]_{S_y} \times \mathbb{K}\,[x, y]_{S_x}$, the vector $D_F(G, H)$ can be computed with $\tilde{O}(|2\,S|)$ operations.

Now, by Theorem 3 of [KS91], we can chose $5\,|2\,S| - 2$ elements at random in $\mathcal{S}$ and compute the dimension of the kernel of D_F with $O(|2\,S|)$ evaluations of D_F and $\tilde{O}(|2\,S|^2)$ more operations in $\mathbb{K}$. The probability of success being at least $1 - \frac{3}{2}|2\,S|(|2\,S| + 1)/|\mathcal{S}|$, this concludes the proof. □

Once S is known, the set T can be obtained by means of the naive polynomial product with $\tilde{O}(l_F|S|^2)$ bit-operations by Proposition 10. When the dimension is fixed and S is non-degenerated then $|2\,S|$ grows linearly with $|S|$, whence our algorithm becomes softly quadratic in $|S|$, in terms of the number of operations in $\mathbb{K}$. This new complexity bound is to be compared to a recent algorithm by Weimann that computes the irreducible factorization of a bivariate polynomial within a cost that grows with $|S|^3$ [Wei09a, Wei09b].

In practice, the computation of the integral hull is not negligible when the dimension becomes large. The known algorithms for computing the integral hull of $\operatorname{supp} F$ start by computing the convex hull. The latter problem is classical and can be solved in softly linear time in dimensions 2 and 3 (see for instance [PS85]). In higher dimensions, the size of the convex hull grows exponentially in the worst case and it can be the bottleneck of our algorithm. With the fastest known algorithms, the convex hull can be computed in time $O(s_F^2 + f \log s_F)$ where f is the number of faces of the hull [Sei86] (improvements are to be found in [MS92]). In our implementation, we programmed the naive “gift-wrapping” method, which turned out to be sufficient for the examples below.

In order to enumerate the points with integer coordinates in the convex hull, we implemented a classical subdivision method. This turned out to be sufficient for our purposes. But let us mention that there exist specific and faster algorithms for this task such as in [Bar94b, Bar94a, LZ05] for instance. Discussing these aspects longer would lead us too far from the purposes of the present paper.

7.3. Timings

We chose the following family of examples in two variables, which depends on a parameter α:

$$F_\alpha = \left[x^{\alpha+1} + \sum_{i=0}^{\alpha} a_i x^i y^{\alpha-i}\right] \left[y^{\alpha+1} + \sum_{i=0}^{\alpha} b_i x^i y^{\alpha-i}\right] \left[x^{\lfloor\alpha/2\rfloor-1} y^{\lfloor\alpha/2\rfloor-1} + \sum_{i=0}^{\alpha} c_i x^i y^{\alpha-i}\right].$$

Here $a_i, b_i, c_i \in \mathbb{K} = \mathbb{Z}/p\,\mathbb{Z}$ are taken at random, with $p = 268435459 < 2^{29}$. In Table 11 below, we indicate the size s_F of F, the size of the matrix D_F, and the time spent for computing the integral hull. Then we compare the time taken by the Wiedemann method with the naive and the fast polynomial products. As expected we observe a softly cubic cost with the naive product, and a softly quadratic cost with the fast product. Notice that the supports are rather sparse, so that Kronecker substitution does not compete here.

α	40	80	160	320
integral hull	4	17	66	270
Wiedemann naive	53	411	3 096	24 425
Wiedemann fast	63	293	1 282	5 745
s_{F_α}	447	887	1 767	3 527
size D_{F_α}	1725×926	3445×1846	6885×3686	13765×7366

Table 11. Counting the number of absolutely irreducible factors of F_α (in seconds).

The goal of these examples is less the absolute factorization problem itself than the demonstration of the usefulness of the fast polynomial product on a real application. Let us finally mention that the algorithm with the smallest asymptotic cost, given in [CL07], will not gain on our family F_α, because it starts with performing a random linear change of the variables. To the best of our knowledge, no other software is able to run the present examples faster.

Conclusion

We have presented classical and new algorithms for multiplying polynomials and series in several variables with a special focus on asymptotic complexity. It turns out that the new algorithms lead to substantial speed-ups in specific situations, but are not competitive in a general manner. Of course, the fast algorithms involve many subalgorithms, which make them harder to optimize. With an additional implementation effort, some of the thresholds in our tables might become more favourable for the new algorithms.

In our implementation, all the variants are available independently from one to another, but they can also be combined with given thresholds. This allows the user to finetune the software whenever it is known whether the polynomials are rather dense (which occur if a random change of the variables is done for instance), strictly sparse, etc. In the near future, we hope to extend the present techniques to the division and higher level operations such as the g.c.d. and polynomial factorization.

Bibliography

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[Bar94a] A. I. Barvinok. Computing the Ehrhart polynomial of a convex lattice polytope. *Discrete Comput. Geom.*, 12(1):35–48, 1994.

[Bar94b] A. I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Math. Oper. Res.*, 19(4):769–779, 1994.

[BCS97] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic complexity theory*. Springer-Verlag, 1997.

[Ber] D. Bernstein. The transposition principle. Available from `http://cr.yp.to/transposition.html`.

[BLS03] A. Bostan, G. Lecerf, and É. Schost. Tellegen's principle into practice. In *Proceedings of ISSAC 2003*, pages 37–44. ACM, 2003.

[BM72] A. Borodin and R.T. Moenck. Fast modular transforms via division. In *Thirteenth annual IEEE symposium on switching and automata theory*, pages 90–96, Univ. Maryland, College Park, Md., 1972.

[BM74] A. Borodin and R.T. Moenck. Fast modular transforms. *Journal of Computer and System Sciences*, 8:366–386, 1974.

[Bor56] J. L. Bordewijk. Inter-reciprocity applied to electrical networks. *Applied Scientific Research B: Electrophysics, Acoustics, Optics, Mathematical Methods*, 6:1–74, 1956.

[BP94] D. Bini and V. Pan. *Polynomial and matrix computations (vol. 1): fundamental algorithms*. Birkhauser, 1994.

[BS91] J. Buchmann and V. Shoup. Constructing nonresidues in finite fields and the extended Riemann hypothesis. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 72–79, New York, NY, USA, 1991. ACM.

[BT88] M. Ben-Or and P. Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 301–309, New York, NY, USA, 1988. ACM.

[CGL92] S. Czapor, K. Geddes, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[CK91] D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28:693–701, 1991.

[CKL89] J. Canny, E. Kaltofen, and Y. Lakshman. Solving systems of non-linear polynomial equations faster. In *Proc. ISSAC '89*, pages 121–128, Portland, Oregon, A.C.M., New York, 1989. ACM Press.

[CL07] G. Chèze and G. Lecerf. Lifting and recombination techniques for absolute factorization. *J. Complexity*, 23(3):380–420, 2007.

[Cra36] H. Cramér. On the order of magnitude of the difference between consecutive prime numbers. *Acta Arithmetica*, 2:23–46, 1936.

[CT65] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Computat.*, 19:297–301, 1965.

[DST87] J. H. Davenport, Y. Siret, and É. Tournier. *Calcul formel : systèmes et algorithmes de manipulations algébriques.* Masson, Paris, France, 1987.

[Fat03] R. Fateman. Comparing the speed of programs for sparse polynomial multiplication. *SIGSAM Bull.*, 37(1):4–15, 2003.

[Für07] M. Fürer. Faster integer multiplication. In *Proceedings of the Thirty-Ninth ACM Symposium on Theory of Computing (STOC 2007)*, pages 57–66, San Diego, California, 2007.

[Gao03] S. Gao. Factoring multivariate polynomials via partial differential equations. *Math. Comp.*, 72(242):801–822, 2003.

[GG02] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2-nd edition, 2002.

[GK87] D. Y. Grigoriev and M. Karpinski. The matching problem for bipartite graphs with polynomially bounded permanents is in NC. In *Proceedings of the 28th IEEE Symposium on the Foundations of Computer Science*, pages 166–172, 1987.

[GKM+04] S. Gao, E. Kaltofen, J. May, Z. Yang, and L. Zhi. Approximate factorization of multivariate polynomials via differential equations. In *ISSAC '04: Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 167–174, New York, NY, USA, 2004. ACM.

[GKS90] D. Y. Grigoriev, M. Karpinski, and M. F. Singer. Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields. *SIAM J. Comput.*, 19(6):1059–1063, 1990.

[GKS94] D. Grigoriev, M. Karpinski, and M. F. Singer. Computational complexity of sparse rational interpolation. *SIAM J. Comput.*, 23(1):1–11, 1994.

[GL06] M. Gastineau and J. Laskar. Development of TRIP: Fast sparse multivariate polynomial multiplication using burst tries. In *Proceedings of ICCS 2006*, LNCS 3992, pages 446–453. Springer, 2006.

[GR03] S. Gao and V. M. Rodrigues. Irreducibility of polynomials modulo p via Newton polytopes. *J. Number Theory*, 101(1):32–47, 2003.

[Gra91] T. Granlund et al. GMP, the GNU multiple precision arithmetic library. `http://gmplib.org`, 1991.

[GS09] S. Garg and É. Schost. Interpolation of polynomials given by straight-line programs. *Theoretical Computer Science*, 410(27-29):2659–2662, 2009.

[H+02] J. van der Hoeven et al. Mathemagix, 2002. `http://www.mathemagix.org`.

[HL10] J. van der Hoeven and G. Lecerf. Divide and conquer products for multivariate polynomial and series. Work in progress, 2010.

[Hoe02] J. van der Hoeven. Relax, but don't be too lazy. *J. Symbolic Comput.*, 34:479–542, 2002.

[Hoe04] J. van der Hoeven. The truncated Fourier transform and applications. In J. Gutierrez, editor, *Proc. ISSAC 2004*, pages 290–296, Univ. of Cantabria, Santander, Spain, July 4–7 2004.

[Hoe06] J. van der Hoeven. Newton's method and FFT trading. Technical Report 2006-17, Univ. Paris-Sud, 2006. To appear in JSC.

[HW79] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. Oxford University Press, 1979.

[Joh74] S. C. Johnson. Sparse polynomial arithmetic. *SIGSAM Bull.*, 8(3):63–71, 1974.

[KL88] E. Kaltofen and Y. N. Lakshman. Improved sparse multivariate polynomial interpolation algorithms. In *ISSAC '88: Proceedings of the international symposium on Symbolic and algebraic computation*, pages 467–474. Springer Verlag, 1988.

[KLL00] E. Kaltofen, W. Lee, and A. A. Lobo. Early termination in Ben-Or/Tiwari sparse interpolation and a hybrid of Zippel's algorithm. In *ISSAC '00: Proceedings of the 2000 international symposium on Symbolic and algebraic computation*, pages 192–201, New York, NY, USA, 2000. ACM.

[KLW90] E. Kaltofen, Y. N. Lakshman, and J.-M. Wiley. Modular rational sparse multivariate polynomial interpolation. In *ISSAC '90: Proceedings of the international symposium on Symbolic and algebraic computation*, pages 135–139, New York, NY, USA, 1990. ACM.

[KS91] E. Kaltofen and B. D. Saunders. On Wiedemann's method of solving sparse linear systems. In H. F. Mattson, T. Mora, and T. R. N. Rao, editors, *Proceedings of AAECC-9*, volume 539 of *Lect. Notes Comput. Sci.*, pages 29–38. Springer-Verlag, 1991.

[KT90] E. Kaltofen and B. M. Trager. Computing with polynomials given by black boxes for their evaluations: greatest common divisors, factorization, separation of numerators and denominators. *J. Symbolic Comput.*, 9(3):301–320, 1990.

[Lec07] G. Lecerf. Improved dense multivariate polynomial factorization algorithms. *J. Symbolic Comput.*, 42(4):477–494, 2007.

[LS03] G. Lecerf and É. Schost. Fast multivariate power series multiplication in characteristic zero. *SADIO Electronic Journal on Informatics and Operations Research*, 5(1):1–10, September 2003.

[LZ05] J. B. Lasserre and E. S. Zeron. An alternative algorithm for counting lattice points in a convex polytope. *Math. Oper. Res.*, 30(3):597–614, 2005.

[MB96] K. Makino and M. Berz. Remainder differential algebras and their applications. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational differentiation: techniques, applications and tools*, pages 63–74, SIAM, Philadelphia, 1996.

[MB04] K. Makino and M. Berz. Suppression of the wrapping effect by Taylor model-based validated integrators. Technical Report MSU Report MSUHEP 40910, Michigan State University, 2004.

[MP07] M. Monagan and R. Pearce. Polynomial division using dynamic arrays, heaps, and packed exponent vectors. In *Proc. of CASC 2007*, pages 295–315. Springer, 2007.

[MP09a] M. Monagan and R. Pearce. Parallel sparse polynomial multiplication using heaps. In *ISSAC '09: Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, pages 263–270, New York, NY, USA, 2009. ACM.

[MP09b] M. Monagan and R. Pearce. Sparse polynomial multiplication and division in Maple 14. Available from `http://www.cecm.sfu.ca/CAG/products2009.shtml`, 2009.

[MS92] J. Matoušek and O. Schwarzkopf. Linear optimization queries. In *SCG '92: Proceedings of the eighth annual symposium on Computational geometry*, pages 16–25, New York, NY, USA, 1992. ACM.

[MS04] G. I. Malaschonok and E. S. Satina. Fast multiplication and sparse structures. *Programming and Computer Software*, 30(2):105–109, 2004.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[Sch77] A. Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica*, 7:395–398, 1977.

[Sei86] R. Seidel. Constructing higher-dimensional convex hulls at logarithmic cost per face. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 404–413, New York, NY, USA, 1986. ACM.

[Sha64] D. Shanks. On maximal gaps between successive primes. *Math. Comp.*, 18(88):646–651, 1964.

[Sha09] H. Shaker. Topology and factorization of polynomials. *Math. Scand.*, 104(1):51–59, 2009.

[SS71] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.

[Sto84] D. R. Stoutmeyer. Which polynomial representation is best? In *Proceedings of the 1984 MACSYMA Users' Conference: Schenectady, New York, July 23–25, 1984*, pages 221–243, 1984.

[Sto00] A. Storjohann. *Algorithms for matrix canonical forms*. PhD thesis, ETH, Zürich, Switzerland, 2000.

[Str73] V. Strassen. Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoeffizienten. *Numer. Math.*, 20:238–251, 1973.

[Wei09a] M. Weimann. Algebraic osculation and factorization of sparse polynomials. Available from `http://arxiv.org/abs/0904.0178`, 2009.

[Wei09b] M. Weimann. A lifting and recombination algorithm for rational factorization of sparse polynomials. Available from `http://arxiv.org/abs/0912.0895`, 2009.

[Wer94] K. Werther. The complexity of sparse polynomial interpolation over finite fields. *Appl. Algebra Engrg. Comm. Comput.*, 5(2):91–103, 1994.

[Yan98] T. Yan. The geobucket data structure for polynomials. *J. Symbolic Comput.*, 25(3):285–293, 1998.