Journal of Symbolic Computation 50 (2013) 227–254

Contents lists available at SciVerse ScienceDirect

Journal of Symbolic Computation

www.elsevier.com/locate/jsc

On the bit-complexity of sparse polynomial and series multiplication ☆,☆☆

Joris van der Hoeven, Grégoire Lecerf

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

ARTICLE INFO

Article history:
Received 8 April 2012
Accepted 10 June 2012
Available online 16 August 2012

Keywords:
Sparse multiplication
Power series
Multi-point evaluation
Polynomial factorization
Algorithm

ABSTRACT

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, under the assumption that a tight superset of the support of the product is known, we are able to observe the benefit of asymptotically fast arithmetic for sparse 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.
© 2012 Published by Elsevier B.V.

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 (Schönhage and Strassen, 1971;

☆ 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.
☆☆ This paper was written using GNU TeXmacs (see http://www.texmacs.org) but Elsevier does not support this word processing software. Any resulting loss in typesetting quality should therefore be imputed to Elsevier.

E-mail addresses: vdhoeven@lix.polytechnique.fr (J. van der Hoeven), lecerf@lix.polytechnique.fr (G. Lecerf).
URLs: http://www.lix.polytechnique.fr/~vdhoeven (J. van der Hoeven), http://lecerf.perso.math.cnrs.fr (G. Lecerf).

0747-7171/$ – see front matter © 2012 Published by Elsevier B.V.
http://dx.doi.org/10.1016/j.jsc.2012.06.004

Schönhage, 1977; Cantor and Kaltofen, 1991; Fürer, 2007). 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.

Less effort has been dedicated into the design and implementation of fast algorithms for multiplying multivariate polynomials and series. 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.

We also recall that the complexities of various more complex operations on univariate polynomials can usually be expressed in terms of the complexity of multiplication; see Aho et al. (1974), Bini and Pan (1994), von zur Gathen and Gerhard (2003) for more details on division, *g.c.d.*, *l.c.m.*, Taylor shift, multi-point evaluation, interpolation, etc. It can be expected that better multiplication algorithms for multivariate polynomials will lead to the development of better algorithms for many other operations on multivariate polynomials.

1.1. *Classical multivariate polynomial arithmetic*

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 Johnson (1974), Fateman (1974), Moenck (1976), Stoutemyer (1984), Alagar and Probst (1987), Davenport et al. (1987), Ponder (1991), Czapor et al. (1992). 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* (see definition in Section 2.1), in which case the description of the support simply contains the degree of the polynomial in each variable. 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 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 as considered by Kaltofen and Trager, 1990) or a specific data structure such as *straight-line programs* (see Chapter 4 of Bürgisser et al., 1997 for instance).

For dense representations with block supports, it is classical that standard univariate multiplication algorithms admit natural extensions: the naive algorithm, Karatsuba's algorithm, and fast Fourier transforms (Cooley and Tukey, 1965; Schönhage and Strassen, 1971; Cantor and Kaltofen, 1991; van der Hoeven, 2004) can be applied recursively in each variable, with good performance. Another classical approach is Kronecker substitution which allows to reduce the multivariate product to a univariate one. We refer the reader to classical books such as Bini and Pan (1994), von zur Gathen and Gerhard (2003). 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 multiplication, 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 (Yan, 1998; Gastineau and Laskar, 2006; Monagan and Pearce, 2007, 2009, 2010) (see Fateman, 2003 for some comparisons between several implementations available in 2003).

Multivariate *dichotomic* (also known as *divide and conquer*, or *binary splitting*) approaches have not been discussed much in the literature. Let us for instance consider Karatsuba's algorithm (see von zur Gathen and Gerhard, 2003, Chapter 8 for details). With one variable z only, the usual version of this algorithm first splits P and Q into $P_0(z) + z^k P_1(z)$ and $Q_0(z) + z^l Q_1(z)$, respectively, where $k = \lceil \deg P/2 \rceil$ and $l = \lceil \deg Q/2 \rceil$. Then Karatsuba's algorithm recursively computes

P_0Q_0, P_1Q_1, and $(P_0+P_1)(Q_0+Q_1)$, and performs suitable linear combinations to recover the result. This approach is efficient in the dense block case because each of the input polynomials is cut into two parts of roughly equal sizes in each recursive call of the product (Fateman, 1974). In the sparse case this phenomenon hardly occurs, and it is commonly admitted that this approach is useless (see for instance Fateman, 2003, Section 3 or Malaschonok and Satina, 2004; Zanoni, 2009 for a cost analysis). Alternative *blockwise versions* for the multivariate case have been suggested in van der Hoeven (2002, Section 6.3.3), and refined in van der Hoeven (2010, Section 6), but never tested in practice. Recent theoretical advances can be found in van der Hoeven and Lecerf (2012).

1.2. Related work on fast sparse multiplication

In practice, one often encounters polynomials which are neither really dense nor really sparse. For instance, if a "dense" multivariate polynomial in n variables is truncated by total degree

$$P=\sum_{i_1+\cdots+i_n<d} P_{i_1,\ldots,i_n} z_1^{i_1}\cdots z_n^{i_n},$$

then only $\mathcal{O}(d^n/n!)$ of the d^n coefficients $P_{i_1,\ldots,i_n}$ with $i_1,\ldots,i_n<d$ are non-zero. This makes the use of, say, multivariate Fourier transforms inefficient for large or medium dimensions n. In such intermediate situations (between dense and sparse), it is important to analyze the complexity of multiplication not only as a function of the sizes of the input polynomials, but *also* in terms of the size of the actual product.

In this setting, the computation of the product can actually be decomposed into two subproblems, which are mostly independent in terms of complexity and applications:

M1 Determine the support of R from those of P and Q.
M2 Compute the coefficients of R.

The computation of the support is often the most expensive part; actually, we may see **M1** 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* as necessary to interpolate the result without a precise idea of the support (see for instance Zippel, 1979; Kaltofen and Trager, 1990).

There exists an extensive literature on the problem of sparse interpolation: Prony (1795), Blahut (1979), Grigoriev and Karpinski (1987), Massey and Schaub (1988), Ben-Or and Tiwari (1988), Kaltofen and Lakshman (1988), Kaltofen et al. (1990, 2000), Garg and Schost (2009). In particular, there exist efficient solutions to **M1** if a bound on the size of the support of R is known. In absence of such a bound, there are also efficient probabilistic algorithms for the computation of the support. In this paper, we will not deal with the problem of support computations and focus on question **M2**. We also notice that most of the existing literature on this topic is devoted to the first problem **M1**, but usually evades the question of fast *practical* sparse multiplication algorithms once that the destination support *is* known.

For coefficient fields of characteristic zero, it is proved in Canny et al. (1989) 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 Ben-Or and Tiwari (1988), with the suitable set of evaluation points borrowed from Grigoriev and Karpinski (1987).

For coefficients in a finite field, Grigoriev, Karpinski and Singer designed a specific sparse interpolation algorithm in Grigoriev et al. (1990), which was then improved in Werther (1994), Grigoriev et al. (1994). 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. Another approach has also been proposed by Huang and Rao (1999). As in Canny et al. (1989) a fast product *might* have been deduced from these works, but to the best of our knowledge this has never been done until now.

1.3. Overview of the article

In Sections 2 and 3 we describe naive and well-known algorithms for the dense and sparse representations of polynomials, we recall the technique of Kronecker substitution, and discuss bit-complexities in view of practical performance. We report on our implementations, and discuss thresholds between sparse and dense representations. Section 4 is devoted to naive algorithms for power series. The algorithms in Sections 2, 3 and 4 are classical. We mainly recall them for convenience of the reader and in order to make a fair comparison between the new algorithms from the subsequent sections and the more classical ones.

In Section 5, we turn to the sparse case. Assuming the destination support to be known, we focus on the computation of the coefficients of the product. Our approach is similar to Canny et al. (1989), but relies on a different kind of evaluation points, which first appeared in Grigoriev et al. (1990). The fast product from Canny et al. (1989), which only applies in characteristic zero, suffers from intermediate integer swell. In contrast, our method is primarily designed for finite fields. For integer coefficients we either rely on large primes or on multi-modular methods to deduce new bit-complexity bounds. Section 5 is also 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 Lecerf and Schost (2003) to our new evaluation–interpolation scheme. The series product of Lecerf and Schost (2003) 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) 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 Canny et al. (1989), but we have not tried. Some recent applications to polynomial reduction can be found in van der Hoeven (2012). Most of the algorithms presented in this paper have been implemented in the C++ library `multimix` of the free computer algebra system MATHEMAGIX (van der Hoeven et al., 2002) (revision 6478, available from `http://gforge.inria.fr/projects/mmx/`).

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 with unity, 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 $\mathsf{M}(n) = O(n \log n \log\log n)$ (Cantor and Kaltofen, 1991) and $\mathsf{I}(n) = O(n \log n 2^{\log^* n})$ (Fürer, 2007, 2009), 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 $\mathbb{N}^n = \{0, 1, 2, 3, \ldots\}^n$. 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 $\operatorname{supp} P = \{e \in \mathbb{N}^n \colon\ P_e \neq 0\}$.

A *block* is a Cartesian product

$$\prod_{j=1}^{n} \{0, 1, \ldots, d_j - 1\} \subseteq \mathbb{N}^n$$

with $d_1, \ldots, d_n \in \mathbb{N}$. Given a polynomial $P \in \mathbb{A}[z_1, \ldots, z_n]$, its *block support*, written $\operatorname{dsupp} P$, is the smallest block of the form

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

which contains $\operatorname{supp} P$. We will denote by $d_P = d_{P,1} \cdots d_{P,n}$ the cardinality of $\operatorname{dsupp} P$. In other words, assuming $d_P \neq 0$, we have $d_{P,j} = \deg_{z_j} P + 1$ for $j = 1, \ldots, n$. 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

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

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.

Example 1. If $\mathbb{A} = \mathbb{Z}$, $n = 2$, and $P = 1 + z_1^2 + 3z_1 z_2 + 5z_1^3 z_2$, then we have that $\operatorname{supp} P = \{(0,0), (2,0), (1,1), (3,1)\}$, $d_{P,1} = 4$, $d_{P,2} = 2$, and $\operatorname{dsupp} P = \{0, 1, 2, 3\} \times \{0, 1\}$. The index of $(3, 1)$ in P is $3 + 1 \times 4 = 7$.

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-block-product).*
INPUT: P and Q in $\mathbb{A}[z_1, \ldots, z_n]$.
OUTPUT: PQ.

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

Proposition 1. *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 $O(n\mathsf{I}(\log d_R) + d_P d_Q \log d_R)$ bit-operations.*

Proof. Before entering naive-block-product 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 $O(n\mathsf{I}(\log d_R))$ bit-operations. Then we compute $d_{R,1}d_{R,2}, d_{R,1}d_{R,2}d_{R,3}, \ldots, d_{R,1}d_{R,2}\cdots d_{R,n-1}$, as well as $d_{R,1}(d_{Q,2} - 1), d_{R,1}d_{R,2}(d_{Q,3} - 1), \ldots, d_{R,1}d_{R,2}\cdots d_{R,n-2}(d_{Q,n-1} - 1)$, and $d_{R,1}(d_{P,2} - 1), d_{R,1}d_{R,2}(d_{P,3} - 1), \ldots, d_{R,1}d_{R,2}\cdots d_{R,n-2}(d_{P,n-1} - 1)$ in $O(n\mathsf{I}(\log d_R))$ bit-operations.

For computing efficiently the index l we make use of the enumeration strategy presented in Lemma 1 below. In fact, for each i, we compute $\mathrm{index}(\mathrm{exponent}(i, P), R)$ incrementally in the outer loop. Then for each j we also obtain

$$l := \mathrm{index}\big(\mathrm{exponent}(i, P) + \mathrm{exponent}(j, Q), R\big)$$

incrementally during the inner loop. The conclusion again follows from Lemma 1. □

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 2 *(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 $f' := f$ and $i := 1$.
(2) For $i$ from 1 to $n$ do
    (a) If $d_{Q,i} = 1$ then continue with the next value of $i$.
    (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 $e + \mathrm{dsupp}\, Q$ in R, for any fixed exponent e of P:

Lemma 1. *Assume that $d_{R,i} \geqslant 2$ for all i, and let e be an exponent of P. If the quantities $d_{R,1}d_{R,2}, d_{R,1}d_{R,2}d_{R,3}, \ldots, d_{R,1}d_{R,2}\cdots d_{R,n-1}$ and $d_{R,1}(d_{Q,2} - 1), d_{R,1}d_{R,2}(d_{Q,3} - 1), \ldots, d_{R,1}d_{R,2}\cdots d_{R,n-2}(d_{Q,n-1} - 1)$ are given, and if $\mathrm{index}(e, R)$ is known, then the indices in R of the exponents of $e + \mathrm{dsupp}\, Q$ 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 next-index 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

With the notations from Section 2.1, let us briefly recall Kronecker substitution. For the computation of $R = PQ$, we will need the following version of Kronecker substitution:

$$\begin{aligned} \mathrm{K}_{d_R} &: \mathbb{A}[z_1, \ldots, z_n] \longrightarrow \mathbb{A}[x], \\ P &\longmapsto P\big(x, x^{d_{R,1}}, x^{d_{R,1} d_{R,2}}, \ldots, x^{d_{R,1} \cdots d_{R,n-1}}\big). \end{aligned}$$

It suffices to compute $\mathrm{K}_{d_R}(P)$ and $\mathrm{K}_{d_R}(Q)$, multiply the results, and recover R by

$$R = \mathrm{K}_{d_R}^{-1}\big(\mathrm{K}_{d_R}(P) \mathrm{K}_{d_R}(Q)\big).$$

Proposition 2. *Assuming the block representation, the product $R = PQ$ can be computed using $\mathsf{M}(d_R)$ operations in $\mathbb{A}$ plus $O(n\mathsf{l}(\log d_R) + (d_P + d_Q)\log d_R)$ 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 $d_{R,1} d_{R,2}, d_{R,1} d_{R,2} d_{R,3}, \ldots, d_{R,1} d_{R,2} \cdots d_{R,n-1}$ and $d_{R,1}(d_{Q,2} - 1), d_{R,1} d_{R,2}(d_{Q,3} - 1), \ldots, d_{R,1} d_{R,2} \cdots d_{R,n-2}(d_{Q,n-1} - 1)$ and $d_{R,1}(d_{P,2} - 1), d_{R,1} d_{R,2}(d_{P,3} - 1), \ldots, d_{R,1} d_{R,2} \cdots d_{R,n-2}(d_{P,n-1} - 1)$. This takes $O(n\mathsf{l}(\log d_R))$ bit-operations.

Thanks to Lemma 1, we deduce $\mathrm{K}_{d_R}(P)$ and $\mathrm{K}_{d_R}(Q)$ with $O((d_P + d_Q)\log d_R)$ bit-operations. Thanks to the reverse lexicographic ordering, the inverse $\mathrm{K}_{d_R}^{-1}$ is for free. □

Remark 1. A similar complexity can be obtained using evaluation–interpolation methods, such as the fast Fourier transform (Cooley and Tukey, 1965) or Schönhage–Strassen's variant (Schönhage and Strassen, 1971; Cantor and Kaltofen, 1991). 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 $3d_R/d_{R,j}$ 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 (when $\mathbb{A} = \mathbb{Z}$), Kronecker substitution can also be used in order to reduce the multiplication of two polynomials to the multiplication of two large integers. For any integer a we write

$$l_a := \big\lceil \log_2\big(|a| + 1\big) \big\rceil \tag{1}$$

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

$$h_R \leqslant h := h_P + h_Q + l_{\min(d_P, d_Q)}. \tag{2}$$

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:

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

d_1, d_2	10	20	40	80	160	320
Naive	0.254	3.93	61.3	983	15 729	252 183
Kronecker	0.047	0.265	1.58	7.68	36.7	185
d_P, d_Q	100	400	1600	64 000	25 600	102 400
d_R	361	1521	6241	25 281	101 761	408 321

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

d_1, d_2, d_3	10	20	40	80	160	320
Naive	27.2	1599	102 399	∞	∞	∞
Kronecker	1.79	22.6	269	2860	26 237	2 479 804
d_P, d_Q	1000	8000	64 000	512 000	4 096 000	32 768 000
d_R	6859	59 319	493 039	4 019 679	32 461 759	260 917 119

$$\begin{aligned} \mathrm{K}_{d_R,h} : \mathbb{Z}[z_1, \ldots, z_n] &\longrightarrow \mathbb{Z}, \\ P &\longmapsto \mathrm{K}_{d_R}(P)\bigl(2^{h+1}\bigr). \end{aligned}$$

One thus computes $\mathrm{K}_{d_R,h}(P)$ and $\mathrm{K}_{d_R,h}(Q)$, does the integer product, and recovers

$$R = \mathrm{K}_{d_R,h}^{-1}\bigl(\mathrm{K}_{d_R,h}(P)\mathrm{K}_{d_R,h}(Q)\bigr).$$

Corollary 1. *With the above dense representation, the product R of P times Q in $\mathbb{Z}[z_1, \ldots, z_n]$ takes $O(\mathsf{I}(hd_R) + n\mathsf{I}(\log d_R) + (d_P + d_Q)\log d_R)$ bit-operations, where d_R, d_P, and d_Q are the dense sizes of R, P, and Q respectively, and h is defined in* (2)*.*

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

Remark 2. 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 performance of our implementation for $\mathbb{A} = \mathbb{Z}/p\mathbb{Z}$, with $p = 268\,435\,459 < 2^{29}$. Timings are measured in seconds or milliseconds, using one core of an INTEL CORE I7 at 2.66 GHz running MAC OS X 10.6.4 and GMP 5.0.1 (Granlund et al., 1991). The symbol ∞ in the timing tables means that the time needed is very high, and not relevant. The source code of our implementation can be found in the header file `block_polynomial_modular_int.hpp` of the `multimix` package of MATHEMAGIX. The implemented Kronecker substitution is the one of Remark 2. Our naive product uses hardware integers to store the coefficients.

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 Kronecker substitution is a very good strategy: it involves less operations on the exponents, and fully benefits from the performance of GMP.

3. Naive multiplication of sparse polynomials

In this section, we study the naive multiplication of multivariate polynomials using a sparse representation.

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 bit-size* of an exponent $e \in \mathbb{N}^n$ is written l_e and is defined as $n + \sum_{j=1}^n l_{e_j}$, where l_{e_j} represents the bit-size of e_j as defined in (1). 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.

Example 2. Let $\mathbb{A} = \mathbb{Z}$, and $P = 1 + z_1^2 + 3z_1z_2 + 5z_1^3z_2 \in \mathbb{A}[z_1, z_2]$ be as in Example 1. Then we have that $s_P = 4$, and that $l_{(0,0)} = 2$, $l_{(2,0)} = 2+2 = 4$, $l_{(1,1)} = 2+2 = 4$, $l_{(3,1)} = 2+2+1 = 5$, whence $l_P = 5$.

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 binary 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 3 *(naive-sparse-product).*

```
INPUT: $P$ and $Q$ in $\mathbb{A}[z_1, \ldots, z_n]$.
OUTPUT: $PQ$.

(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 3. *Assuming the sparse representation of polynomials, the product $R = PQ$ 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 and use naive-sparse-product. 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 (5) never exceeds $O(l_P + l_Q)$, which yields the claimed bound. □

Remark 3. 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 4. For very large input polynomials it is useful to implement an additional dichotomy on Q in order to ensure that Q fits in the cache, most of the time. The sum of two polynomials can also benefit from the cache oblivious merge algorithm in Blelloch et al. (2010, Section 2).

Remark 5. The algorithm naive-sparse-product was already known, analyzed, and also extended to a parallel computational model in Ponder (1991, Section 5). More efficient variants were designed in Johnson (1974), Monagan and Pearce (2009).

Table 3
Sparse polynomial product in MATHEMAGIX, $n = 3$ (in milliseconds).

s_P, s_Q	80	160	320	640	1280	2560	5120
Default	0.782	3.22	16.2	65.8	277	1192	6021
Packed	0.365	1.56	6.66	34.9	155	674	3172

Table 4
Sparse polynomial product in MAPLE 13, $n = 3$ (in milliseconds).

s_P, s_Q	80	160	320	640	1280	2560	5120
Expand mod p	3.52	16.6	73.2	349	1757	8068	34 772
sdmp packed	0.323	1.21	4.91	22.4	94	419	2086

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

Density	$n = 2$, $d = 80$	$n = 3$, $d = 40$
1%	0.462	46.7
2%	1.60	153
3%	2.75	331
4%	4.35	538
5%	5.81	804
6%	7.83	1108
7%	10.12	1465
Kronecker	7.68	269

3.2. *Timings*

In the next tables we provide timings for $\mathbb{A} = \mathbb{Z}/p\mathbb{Z}$, with $p = 268\,435\,459$, and $n = 3$, and we pick up $s_P = s_Q$ random monomials in the block $\{0, \dots, 10\,000\}^n$, with random coefficients. In MATHEMAGIX, unless specified otherwise, the exponents are represented by vectors with 32-bits unsigned hardware integer entries. The time needed for this product is displayed in the row "default" of Table 3. The corresponding MATHEMAGIX source code can be found in the file `sparse_polynomial_modular_int.hpp` of the `multimix` package. Internally, the polynomials to be multiplied are converted into a "packed" representation for the exponents. Packing the exponents is a classical and useful trick when the coefficients are small, and already used by Johnson (1974). In the present case each exponent is packed into a 64 bits word. The time spent by the product on the actual packed representations is displayed in the row "packed" of Table 3. With the current value of p, the coefficient products are stored into 64 bits hardware integers, so that we accumulate up to 64 coefficient products before doing a division by p.

For the sake of comparison, we report on the performance of MAPLE 13 for the same operations in Table 4. The row "Expand mod p" corresponds to executing the command `Expand (P * Q) mod p`, while the row "sdmp packed" corresponds to measuring the time spent by the `sdmp` (Monagan and Pearce, 2007, 2009, 2010) library (distributed within MAPLE 13) to perform the product in the packed representation.

These timings show that our implementation of the naive algorithms is not as optimized as in `sdmp`, although competitive. Notice that the time needed for packing and unpacking exponents is only important in the extreme case when $s_R \approx s_P s_Q$. In the sequel we will only use the default product, thereby taking into account the time spent in the conversions.

In Table 5 we consider polynomials of increasing size in a fixed block $\{0, \dots, d-1\}^n$. The row "density" indicates the ratio of non-zero terms with exponent in $\{0, \dots, d-1\}^n$.

In Table 5, we see that Kronecker substitution is faster for densities larger than 6% 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 $\mathsf{M}(d_R)$ since the efficiency of Kronecker substitution really relies on the underlying univariate arithmetic.

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 the product $H = FG$ works as follows:

Algorithm 4 *(naive-series-product).*
INPUT: F and G in $\mathbb{A}[[z_1, \ldots, z_n]]$ to precision δ.
OUTPUT: FG.

```
(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 4. *With the above sparse representation, the product $H = FG$ can be computed using $O(s_F s_G \log \min(s_F, s_G))$ arithmetic 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 3, the total number of operations in $\mathbb{A}$ is in

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

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

Proposition 4 is pessimistic in many cases: the complexity bound is merely a bound for the corresponding polynomial product without truncation. In the next subsection we will take truncations into account in our complexity bound.

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, which occurs when $s_{F_i} = s_{G_i} = h_{i,n}$ for $i \in \{1, 2\}$.

Proposition 5. *Assuming the sparse representation of polynomials, the product $H = FG$ 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 4 thanks to the following identity:

$$\sum_{d=0}^{\delta-1} \sum_{i+j=d} h_{i,n} h_{j,n} = s_{\delta,2n}. \tag{3}$$

Table 6
Naive dense series product (in milliseconds).

δ	10	20	40	80	160	320
$n=2$	0.0642	0.587	7.39	70.6	677	7871
$n=3$	0.373	7.05	173	7151	409 926	∞
$n=6$	9.80	3350	∞	∞	∞	∞
$s_{\delta,2}$	55	210	820	3240	12 880	51 360
$s_{\delta,3}$	220	1540	11 480	88 560	695 520	5 512 640
$s_{\delta,6}$	5005	177 100	8 145 060	437 353 560	25 564 880 880	1 562 461 336 800

This identity is already given in van der Hoeven (2010, 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 the 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 (3). Finally the bit-cost follows in the same way with l_F and l_G being bounded by $O(nl_\delta)$. $\square$

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

$$\log s_{\delta,n} = \log \binom{2n-1}{n} \sim (\log 4) n,$$
$$\log s_{\delta,2n} = \log \binom{3n-1}{n} \sim \left(\log \frac{27}{4}\right) n.$$

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 $\sqrt{\pi n}/4^n$. For small n, one could also benefit from the truncated Fourier transform, as explained in van der Hoeven (2010, Section 6).

4.3. Timings

In Table 6 we report on timings for $\mathbb{A} = \mathbb{Z}/p\mathbb{Z}$, with $p = 268\,435\,459$, obtained with our implementation in the file `sparse_jet_modular_int.hpp` in the `multimix` package. 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. In higher dimensions, say $n = 6$ and order $\delta = 20$, Kronecker substitution becomes hopeless: the size of the univariate polynomials to be multiplied is $40^6 \approx 4.1 \cdot 10^9$.

Notice that we did not propose a specific dense representation for our series. We might have done so in order to gain in memory space (but not so much with respect to the "packed exponent" technique). However one cannot expect much more because there is no straightforward generalization to this setting of certain fast algorithms such as Kronecker substitution.

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 again be the two multivariate polynomials in $\mathbb{A}[z_1, \ldots, z_n]$ that we want to multiply, and let $R = PQ$ denote their product. We assume that we are given a set $X \subseteq \mathbb{N}^n = \{0, 1, 2, 3, \ldots\}^n$ of size s_X which contains the support of R. We let $d_{X,1}, \ldots, d_{X,n} \in \mathbb{N}$ be the minimal numbers with $X \subseteq \prod_{j=1}^{n}\{0,\dots,d_{X,j}-1\}$. Without loss of generality we may assume that $d_{X,j} \geqslant 2$ for all j.

For the analysis of the algorithms in this section, we introduce the quantities $e_P = \sum_{e\in \operatorname{supp} P} l_e$, $e_Q = \sum_{e\in \operatorname{supp} Q} l_e$ and $e_X = \sum_{e\in X} l_e$, where l_e has been defined in the beginning of Section 3.1. 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.

Example 3. Let $\mathbb{A} = \mathbb{K} = \mathbb{Q}$, $n = 2$, $P = 1 + z_1^2 + 3z_1z_2 + 5z_1^3z_2$, and $Q = 1 + 2z_1z_2$. Then we have $s_P = 4$, $s_Q = 2$, $e_P = 15$ (from Example 2), and $e_Q = 6$. We can take $X = \{(0,0),(2,0),(1,1),(3,1),(2,2),(4,2)\}$, since $PQ = 1 + z_1^2 + 5z_1z_2 + 7z_1^3z_2 + 6z_1^2z_2^2 + 10z_1^4z_2^2$.

5.1. *Evaluation, interpolation and transposition*

Given t pairwise distinct points $\omega_0,\dots,\omega_{t-1} \in \mathbb{K}^*$ and $s \in \mathbb{N}$, let $E : \mathbb{A}^s \to \mathbb{A}^t$ be the linear map which sends $(a_0,\dots,a_{s-1})$ to $(A(\omega_0),\dots,A(\omega_{t-1}))$, 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

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

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 (Borodin and Moenck, 1972; Strassen, 1973; Borodin and Moenck, 1974) that both problems can be solved in time $O(\lceil t/s \rceil \mathsf{M}(s)\log s)$; see also von zur Gathen and Gerhard (2003, Chapter 10) for a complete description. Notice that the algorithms only require *vectorial operations* in $\mathbb{A}$ (vector additions, subtractions and multiplications by 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

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

By the transposition principle (Bordewijk, 1956; Bernstein, 2006; Bostan et al., 2003), 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 $E^\top$ (Bostan et al., 2003). Given a vector $a \in (\mathbb{A}^t)^*$ with entries $a_0,\dots,a_{t-1}$, the entries $b_0,\dots,b_{s-1}$ of $E^\top(a)$ are identical to the first s coefficients of the power series

$$\sum_{0\leqslant i<t} \frac{a_i}{1-\omega_i u} = \sum_{j\geqslant 0} \left(a_0\omega_0^j + \cdots + a_{t-1}\omega_{t-1}^j\right)u^j.$$

The numerator and denominator of the latter rational function can be computed using the classical divide and conquer technique, as described in von zur Gathen and Gerhard (2003, Algorithm 10.9). If $t \leqslant s$, then this requires $O(\mathsf{M}(s)\log s)$ vectorial operations in $\mathbb{A}$ (von zur Gathen and Gerhard, 2003, 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 at most s, and obtain the complexity bound $O(\lceil t/s \rceil \mathsf{M}(s)\log s)$.

Inversely, assume that we wish to recover $a_0,\dots,a_{s-1}$ from $b_0,\dots,b_{s-1}$, 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 $S = BD$, we notice that $S(\omega_i^{-1}) = -a_i(uD')(\omega_i^{-1})$ for all i. Hence, the computation of the a_i reduces to two multi-point evaluations of S and $-uD'$ at $\omega_0^{-1}, \dots, \omega_{s-1}^{-1}$ and s divisions in $\mathbb{K}$. 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

$$\begin{aligned} \mathrm{E} : \mathbb{A}[z] &\longrightarrow \mathbb{A}^{s_X}, \\ A &\longmapsto \big(\mathrm{K}_{d_X}(A)(1), \mathrm{K}_{d_X}(A)(\omega), \dots, \mathrm{K}_{d_X}(A)\big(\omega^{s_X - 1}\big)\big). \end{aligned}$$

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

Given $Y = \{i_1, \dots, i_t\} \subseteq \prod_{j=1}^n \{0, \dots, d_{X,j} - 1\}$, 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

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

Taking $Y = \operatorname{supp} P$, *respectively* $Y = \operatorname{supp} Q$, this allows us to compute $\mathrm{E}(P)$ and $\mathrm{E}(Q)$ using the algorithm for transposed multi-point evaluation from the preceding subsection. We obtain $\mathrm{E}(R) = \mathrm{E}(P)\mathrm{E}(Q)$ using one Hadamard (*i.e.* elementwise) product of two vectors. Taking $Y = X$, the points $\omega^{k_1}, \dots, \omega^{k_t}$ 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. Using the above notations, our fast algorithm for the computation of sparse products becomes:

Algorithm 5 *(fast-sparse-product).*
INPUT: $P, Q \in \mathbb{A}[z_1, \dots, z_n]$ and a finite set $X \subseteq \mathbb{N}^n$ with $\operatorname{supp}(PQ) \subseteq X$.
OUTPUT: PQ.

```
(1) Compute $\omega^{k_i}$, where $k_i = \mathrm{index}(i, X)$ for all $i \in X$.
(2) Compute $\mathrm{E}(P)$ and $\mathrm{E}(Q)$.
(3) Compute $\mathrm{E}(P)\mathrm{E}(Q)$.
(4) Compute $R = PQ$ as $\mathrm{E}^{-1}(\mathrm{E}(P)\mathrm{E}(Q))$.
```

Recalling that $\epsilon = e_P + e_Q + e_X$ and $\sigma = s_P + s_Q + s_X$, the computational complexity of the fast algorithm is as follows:

Proposition 6. *Given two polynomials P and Q in $\mathbb{A}[z_1, \dots, z_n]$, a subset $X \subseteq \mathbb{N}^n$ that contains the support of PQ, 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* $\operatorname{supp} P$, $\operatorname{supp} Q$ *and X;*
- *$O(s_X)$ inversions in $\mathbb{K}$, $O(s_X)$ products in $\mathbb{A}$, and $O(\frac{\sigma}{s_X} \mathsf{M}(s_X) \log s_X)$ vectorial operations in $\mathbb{A}$.*

Proof. By classical binary powering, the computation of the sequence

$$\omega^{d_{X,1}}, \dots, \omega^{d_{X,1} \cdots d_{X,n-1}}$$

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 $\omega^{\text{index}(i,P)}$ for $i \in \operatorname{supp} P$ (resp. $\operatorname{supp} Q$ 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 can compute $\mathsf{E}(P)$ and $\mathsf{E}(Q)$ 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 $\mathsf{E}(R)$ 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. $\square$

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 6 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 2. *Let $P, Q \in \mathbb{F}_{p^k}[z_1, \ldots, z_n]$, let X be a finite subset $X \subseteq \mathbb{N}^n$ with $X \supseteq \operatorname{supp} PQ$, and let ω be a primitive element of $\mathbb{F}_{p^k}^*$, and assume that $p^k - 1 \geqslant d_X$. Then 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;*
- *$O(\frac{\sigma}{s_X}\mathsf{M}(s_X k)\log s_X + s_X \mathsf{M}(k)\log k)$ 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 (von zur Gathen and Gerhard, 2003, Chapter 11), with using pseudo-divisions instead of divisions. Using Kronecker substitution, one product in $\mathbb{F}_{p^k}[u]$ in size n takes $O(\mathsf{M}(nk))$ operations in $\mathbb{F}_p$. The conclusion thus follows from Proposition 6. $\square$

Applying the general purpose algorithm by Cantor and Kaltofen (1991), 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 2 therefore further implies:

Corollary 3. *Let $P, Q \in \mathbb{F}_{p^k}[z_1, \ldots, z_n]$, let X be a subset $X \subseteq \mathbb{N}^n$ that contains the support of PQ, and let ω be a primitive element of $\mathbb{F}_{p^k}^*$, and assume that $p^k - 1 \geqslant d_X$ and that $\log(s_X k) = O(\log p)$. Then 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;*
- *$O(\frac{\sigma}{s_X}\mathsf{I}(s_X k \log p)\log s_X + s_X \mathsf{I}(k \log p)\log k + s_X \mathsf{I}(\log p)\log\log p)$ 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 latter corollary. Such constructions are classical, see for instance von zur Gathen and Gerhard (2003, 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 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 (Buchmann and Shoup, 1991, Section 1, *Applications*).

5.4. Integer coefficients

Let $\mathbb{A} = \mathbb{Z}$, and 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

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

In this subsection we investigate two classical strategies for reducing polynomial multiplication over the integers to polynomial multiplication over finite fields: (1) picking a sufficiently large prime number and (2) Chinese remaindering coefficients using several small prime numbers.

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 $R \bmod p$ 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 3 now implies:

Corollary 4. *Given $P, Q \in \mathbb{Z}[z_1, \ldots, z_n]$, a subset $X \subseteq \mathbb{N}^n$ that contains the support of PQ, 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;*
- *$O(\frac{\sigma}{s_X} \mathsf{I}(s_X \log p) \log s_X + s_X \mathsf{I}(\log p) \log \log p)$ 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 (Cramér, 1936; Shanks, 1964) states that

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

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 by Agrawal et al. (2004), 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., together with the corresponding primitive elements.

5.4.2. Chinese remaindering

In our fast algorithm and Corollary 4, we regard the computation of a prime number $p > N = \max(2^{h+1}, d_X)$ and the corresponding primitive elements 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

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

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 (Hardy and Wright, 1979, Chapter 12, Theorem 1.3) ensures us that there exists p_1 between $d_X + 1$ and $2d_X$, then one can take p_2 between $p_1 + 1$ and $2p_1$, 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 5. *Given $P, Q \in \mathbb{Z}[z_1, \ldots, z_n]$, a finite subset $X \subseteq \mathbb{N}^n$ with $X \supseteq PQ$, 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;
- $O(\frac{\sigma}{s_X}\mathsf{I}(s_X \log p) \log s_X + s_X \mathsf{I}(\log p) \log\log p)$ *bit-operations.*

Proof. This follows from Proposition 6, following the proofs of Corollaries 2 and 3, *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 (Makino and Berz, 1996, 2005), 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

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

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 (see also van der Hoeven, 2002, Section 6.2). 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

Techniques for reducing the case when $\mathbb{K} = \mathbb{Q}$ to the case of integer coefficients are very classical. For the sake of completeness, we will briefly recall the main remarks which can be made in this context.

Let q_P and q_Q denote the least common multiples of the denominators of the coefficients of P *respectively* Q. One obvious way to compute PQ is to set $\hat{P} := Pq_P$, $\hat{Q} := Qq_Q$, 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 5.3. For $i = 1, 2, 4, 8, \ldots,$ we may recover $PQ \bmod p_1 \cdots p_i$ using Chinese remaindering and attempt to reconstruct PQ from $PQ \bmod p_1 \cdots p_i$ using rational number reconstruction (von zur Gathen and Gerhard, 2003, Chapter 5). If this yields the same result for a given i and $2i$, then the reconstructed PQ 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 performance of the algorithms in this section for a finite prime field, which is the central case to optimize. We take $\mathbb{A} = \mathbb{Z}/p\mathbb{Z}$, with

$$p = 2\,305\,843\,009\,213\,693\,967 < 2^{62}.$$

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 Kronecker substitution is most efficient in practice. Here we consider a case which lies in between these two extremes.

More precisely, we pick bivariate polynomials with terms of total degree at most δ and at least $\delta - 3$ with random coefficients in $\mathbb{Z}/p\mathbb{Z}$. The subset X can be easily taken as the set of the monomials of degree at most 2δ and at least $2\delta - 6$. In Table 7 we compare the fast algorithm of Section 5.3 to the naive one of Section 3 and the direct use of Kronecker substitution. The source code for our fast algorithm can be found in the file `sparse_polynomial_fast.hpp` in the `multimix` package. The source code and the documentation for the arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$ can be found in the file `modular_int.hpp` and the `doc` directory in the `numerix` package.

In the bivariate case (as well as in the trivariate case), the asymptotic complexity is already reflected in the actual timings. But the fast algorithm only gains for very large sizes.

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.

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

δ	20	40	80	160	320	640	1280	2560	5120
s_P	78	158	318	638	1278	2558	5118	10 238	20 478
s_X	266	546	1106	2226	4466	8946	17 906	35 826	71 666
Naive	0.501	2.00	8.08	33.7	133	514	2208	8669	34 422
Kronecker	0.767	3.77	17.4	82.7	433	2203	9350	39 426	1 374 955
Fast	31.9	60.1	120	259	585	1298	2894	6335	14 440

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 $\deg P = \max\{|i|\colon\ P_i \neq 0\}$. 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 $I_\delta = \{i \in \mathbb{N}^n\colon\ |i| < \delta\}$. Then

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

is the set of polynomials of total degree at most $\delta - 1$. 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 by Lecerf and Schost (2003).

Remark 6. The results of this section could be generalized in the same way as in van der Hoeven (2002) 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 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

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

Inversely, for any $P \in \mathbb{A}[z]_{J_\delta}$, there exists a unique polynomial $\mathrm{T}^{-1}(P) \in \mathbb{A}[z]_{I_\delta}$ 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

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

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

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

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

$$X = \big\{i \in \mathbb{N}^{n-1}\colon\ i_1 + \cdots + i_{n-1} < \delta\big\}. \tag{4}$$

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

$$\mathrm{E}(PQ) = \mathrm{E}(P)\mathrm{E}(Q). \tag{5}$$

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

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

Given $P, Q \in \mathbb{A}[z]_{J_\delta}$ and $j < \delta$, the relation (5) yields

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

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. According to the latter notation, the truncated product of P and Q summarizes as follows:

Algorithm 6 *(fast-series-product).*
INPUT: P and Q in $\mathbb{A}[[z_1, \ldots, z_n]]$ to precision δ.
OUTPUT: PQ.

```
(1) Compute $\omega^{k_i}$, where $k_i = \mathrm{index}(i, X)$ for all $i \in X$, as defined in (4).
(2) Compute $\mathrm{E}(\mathrm{T}(P))$ and $\mathrm{E}(\mathrm{T}(Q))$.
(3) Compute $\mathrm{E}(\mathrm{T}(P))\mathrm{E}(\mathrm{T}(Q))$.
(4) Compute $R = PQ$ as $\mathrm{T}^{-1}(\mathrm{E}^{-1}(\mathrm{E}(\mathrm{T}(P))\mathrm{E}(\mathrm{T}(Q))))$.
```

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 7. *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 fast-series-product. 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 7. Since $s_X\delta = \tilde{O}(s_{\delta,n})$ by Lecerf and Schost (2003, Lemma 3), Proposition 7 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 7.

Corollary 6. *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*

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

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 7. □

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 Lecerf and Schost (2003, 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 PQ 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 $(PQ)_{I_\delta}$ can be computed using $\tilde{O}(nhs_{\delta,n})$ bit-operations, by Corollary 6. Otherwise, in the same way we did for polynomials in Section 5.4, Chinese remaindering leads to:

Corollary 7. *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*

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

bit-operations.

Proof. Let $p = p_1 \cdots p_r$. Since $\log s_X = O(\log p)$, we can use Kronecker substitution with the algorithm underlying Proposition 7 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 7 is softly optimal, namely $\tilde{O}(hs_{\delta,n})$, which is much better than a direct use of Kronecker substitution when n becomes large.

Table 8
Fast series product (in milliseconds).

δ	10	20	40	80	160	320
$n=2$	1.39	7.66	48	219	969	4077
$n=3$	12.8	117	1040	9734	91 042	∞

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

δ	10	20	40	50	60
Naive	0.00117	0.0507	5.6	29.3	123
Kronecker	0.0567	1.74	36.3	1313	∞
Fast	0.0522	0.838	15.5	41.2	92

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

δ	5	10	15	20	25	30	35
Naive	0.000123	0.0084	0.167	1.66	10.7	52.4	216
Kronecker	0.0364	3.14	28.7	∞	∞	∞	∞
Fast	0.071	0.405	2.26	9.41	29.9	78.2	181

6.7. Timings

We take $\mathbb{A} = \mathbb{Z}/p\mathbb{Z}$, with $p = 268\,435\,459$, and report on the performance of our implementation made in the file `sparse_jet_fast.hpp` of the `multimix` package. The following tables demonstrate that the theoretical *softly linear* asymptotic costs can really be observed.

When comparing Table 8 to 6, we see that our fast product does not compete with the naive algorithm up to order 320 in the bivariate case. For three variables we observe that it also outperforms the naive approach at large orders, but that it is still slower than Kronecker substitution reported in Tables 1 and 2.

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 variables (see Table 10), we take $\mathbb{A} = \mathbb{Z}/p\mathbb{Z}$, with $p = 2\,305\,843\,009\,213\,693\,967$. The Kronecker product is very expensive, the naive algorithm becomes subquadratic, so that the threshold for the fast algorithm is much higher.

7. Absolute factorization

Let $\mathbb{K}$ be a field, and let $F \in \mathbb{K}[x_1, \dots, x_n, y] = \mathbb{K}[x, y]$. In this section we are interested in counting the number of the *absolutely irreducible factors* of F, *i.e.* the number of the 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 $\mathbb{R}^{n+1}$. If S' is an other subset of $\mathbb{Z}^{n+1}$ then we write $S + S'$ for the *Minkowski sum* $\{s + s' \colon\ s \in S,\ s' \in S'\}$.

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

$$S_x = S \cap \big(\big(\mathbb{N}^n \setminus \{(0, \dots, 0)\}\big) \times \mathbb{N} \big),$$

$$S_y = S \cap \left(\mathbb{N}^n \times \mathbb{N} \setminus \{0\}\right),$$
$$S_{x,y} = S_x \cap S_y,$$
$$T = (S_{x,y} + S) \cup (S_x + S_y).$$

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 $\operatorname{supp} \theta_x \theta_y F$.

Example 4. If $\mathbb{K} = \mathbb{Q}$ and $F = 2 + 3x_1 + 4y^2$, then we have $S = \{(0,0), (1,0), (0,2)\}$, $S_x = \{(1,0)\}$, $S_y = \{(0,2)\}$, $S_{x,y} = \{\}$, $\theta_x F = 3x_1$, $\theta_y F = 8y^2$, $\theta_x \theta_y F = 0$.

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

$$D_F : \mathbb{K}[x, y]_{S_y} \times \mathbb{K}[x, y]_{S_x} \longrightarrow \mathbb{K}[x, y]_T,$$
$$(G, H) \longmapsto G\theta_x F - H\theta_y F - (\theta_x G - \theta_y H)F,$$

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 8. *Assume that $\mathbb{K}$ has characteristic 0 or at least $d_x(2d_y - 1) + 1$, 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 in $\overline{\mathbb{K}(x)}$ of $F(x, y)$ seen as in $\mathbb{K}(x)[y]$. The assumption on the discriminant of F ensures that all are simple. Now consider the partial fraction decompositions of G/F and H/F:

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

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 $d_x(2d_y - 1) + 1$ this still holds by the same arguments as in Gao (2003, Lemma 2.4). 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 von zur Gathen and Gerhard (2003, Theorem 22.8) or Chèze and Lecerf (2007, 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 pairs $(\hat{F}_i \theta_y F_i, \hat{F}_i \theta_x F_i)$ form a basis of the kernel of D_F over $\bar{\mathbb{K}}$, which concludes the proof. □

Proposition 8 was first stated by Gao (2003) in the bivariate case for the dense representation of the polynomials, and then in terms of the actual support of F in Gao and Rodrigues (2003) but still for two variables. For several variables and block supports, generalizations have been proposed in Gao et al. (2004, 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 by Lecerf, 2007, we refer the reader to Shaker, 2009 for the details).

For a complete history of the algorithms designed for the absolute factorization we refer the reader to the introduction of Chèze and Lecerf (2007). 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 (Storjohann, 2000, 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 Gao (2003, Sections 3 and 4), concerning the bivariate case, Gao computes the kernel of D_F using Wiedemann's algorithm: roughly speaking this reduces to compute the image by D_F of at most $2|2S|$ vectors. With a block support, and for when the dimension is fixed, 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 7 *(probabilistic-factor-count).*
INPUT: F in $\mathbb{K}[x_1, \ldots, x_n, y]$.
OUTPUT: The 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) Precompute 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 Kaltofen and Saunders (1991, 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 9. *Assume that $\mathbb{K}$ has characteristic 0 or at least $d_x(2d_y - 1) + 1$, 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 $(2d_y + 1) \prod_{i=1}^n (2 \deg_{x_i} F + 1)$, and that the given set $\mathcal{S}$ contains at least $5|2S| - 2$ elements. Then* probabilistic-factor-count *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 $\tilde{O}(|2S|^2)$ operations in $\mathbb{K}$. It returns a correct answer with a probability at least $1 - \frac{3}{2}|2S|(|2S| + 1)/|\mathcal{S}|$.*

Proof. Since T is included in $2S$, the assumption on the order of ω allows us to apply Proposition 6. 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 $O(|2S|)$ 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 $\tilde{O}(|2S|)$ operations.

Now, by Kaltofen and Saunders (1991, Theorem 3), we can choose $5|2S|-2$ elements at random in $\mathcal{S}$ and compute the dimension of the kernel of D_F with $O(|2S|)$ evaluations of D_F and $\tilde{O}(|2S|^2)$ more operations in $\mathbb{K}$. The probability of success being at least $1-\frac{3}{2}|2S|(|2S|+1)/|\mathcal{S}|$, this concludes the proof. $\square$

Remark 8. Looking closer to Kaltofen and Saunders (1991, Theorem 3), one can see that probabilistic-factor-count always returns an upper bound on the number of the absolutely irreducible factors of F.

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 3. When the dimension is fixed and S is non-degenerated then $|2S|$ 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 $|S|^3$ (Weimann, 2010, 2012).

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 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 Preparata and Shamos, 1985, Chapter 3). 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 (Seidel, 1986) (improvements are to be found in Matoušek and Schwarzkopf, 1992). In our implementation, we programmed the naive "gift-wrapping" method (Preparata and Shamos, 1985, Chapter 3), 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 Barvinok (1994b, 1994a), Lasserre and Zeron (2005) for instance. Discussing these aspects longer would lead us too far from the purposes of the present paper.

7.3. Timings

We choose 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 $a_i, b_i, c_i \in \mathbb{K}$, where $\mathbb{K} = \mathbb{Z}/p\mathbb{Z}$ are taken at random, with $p = 268\,435\,459 < 2^{29}$. In Table 11, we indicate the size s_F of F, the size $|S_\alpha|$ of the convex hull of the support of F_α, the size of the matrix D_F, and the time spent for computing the integral hull. Then we compare the timings for Wiedemann's method with those for 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.

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 by Chèze and Lecerf (2007), 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.

Remark 9. After the submission of this article, further progress has been made on the factorization of bivariate polynomials in terms of the convex hulls of their supports (Berthomieu and Lecerf, 2012).

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

α	40	80	160	320
Integral hull	4	16	62	270
Wiedemann naive	25	176	1313	10 305
Wiedemann fast	66	295	1352	6006
s_{F_α}	447	887	1767	3527
$\lvert S_\alpha \rvert$	465	925	1845	3685
Size of D_{F_α}	1725×926	3445×1846	6885×3686	$13\,765 \times 7366$

8. 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 sub-algorithms, which make them harder to optimize. With an additional implementation effort, we think that some of the thresholds in our tables might become more favorable for the new algorithms.

In our implementation, all the variants are available independently from one another, and they can be combined with given thresholds. This allows the user to fine tune the software for particular applications. For instance, in specific situations, it may be known whether the polynomials are rather dense (which occurs if a random change of the variables is done for instance), strictly sparse, etc. This feature turns out to be useful since the automatic determination of thresholds remains a difficult practical issue in several variables. In the near future, we hope to extend the present techniques to higher level operations such as the *g.c.d.* and polynomial factorization.

Acknowledgements

We would like to thank the anonymous referees for their helpful detailed comments, and especially for their indications to establish Table 4.

References

Agrawal, M., Kayal, N., Saxena, N., 2004. PRIMES is in P. Ann. of Math. 160 (2), 781–793.
Aho, A.V., Hopcroft, J.E., Ullman, J.D., 1974. The Design and Analysis of Computer Algorithms. Addison–Wesley.
Alagar, V.S., Probst, D.K., 1987. A fast, low-space algorithm for multiplying dense multivariate polynomials. ACM Trans. Math. Software 13 (1), 35–57.
Barvinok, A.I., 1994a. Computing the Ehrhart polynomial of a convex lattice polytope. Discrete Comput. Geom. 12 (1), 35–48.
Barvinok, A.I., 1994b. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. Math. Oper. Res. 19 (4), 769–779.
Ben-Or, M., Tiwari, P., 1988. A deterministic algorithm for sparse multivariate polynomial interpolation. In: Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC '88). ACM Press, pp. 301–309.
Bernstein, D., 2006. The transposition principle. Available from http://cr.yp.to/transposition.html.
Berthomieu, J., Lecerf, G., 2012. Reduction of bivariate polynomials from convex-dense to dense, with application to factorizations. Math. Comp. 81 (279).
Bini, D., Pan, V., 1994. Polynomial and Matrix Computations (vol. 1): Fundamental Algorithms. Birkhäuser.
Blahut, R., 1979. Transform techniques for error control codes. IBM J. Res. Develop. 23, 299–315.
Blelloch, G.E., Gibbons, P.B., Simhadri, H.V., 2010. Low depth cache-oblivious algorithms. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10). ACM Press, pp. 189–199.
Bordewijk, J.L., 1956. Inter-reciprocity applied to electrical networks. Appl. Sci. Res. B: Electrophys. Acoust. Optics Math. Methods 6, 1–74.
Borodin, A., Moenck, R.T., 1972. Fast modular transforms via division. In: IEEE Conference Record of 13th Annual Symposium on Switching and Automata Theory. IEEE, pp. 90–96.
Borodin, A., Moenck, R.T., 1974. Fast modular transforms. J. Comput. System Sci. 8, 366–386.
Bostan, A., Lecerf, G., Schost, É., 2003. Tellegen's principle into practice. In: Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation (ISSAC '03). ACM Press, pp. 37–44.
Buchmann, J., Shoup, V., 1991. Constructing nonresidues in finite fields and the extended Riemann hypothesis. In: Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing (STOC '91). ACM Press, pp. 72–79.

Bürgisser, P., Clausen, M., Shokrollahi, M.A., 1997. Algebraic Complexity Theory. Springer-Verlag.
Canny, J., Kaltofen, E., Lakshman, Y., 1989. Solving systems of non-linear polynomial equations faster. In: Proceedings of the 1989 International Symposium on Symbolic and Algebraic Computation (ISSAC '89). ACM Press, pp. 121–128.
Cantor, D.G., Kaltofen, E., 1991. On fast multiplication of polynomials over arbitrary algebras. Acta Inform. 28, 693–701.
Chèze, G., Lecerf, G., 2007. Lifting and recombination techniques for absolute factorization. J. Complexity 23 (3), 380–420.
Cooley, J.W., Tukey, J.W., 1965. An algorithm for the machine calculation of complex Fourier series. Math. Comp. 19, 297–301.
Cramér, H., 1936. On the order of magnitude of the difference between consecutive prime numbers. Acta Arith. 2, 23–46.
Czapor, S., Geddes, K., Labahn, G., 1992. Algorithms for Computer Algebra. Kluwer Academic Publishers.
Davenport, J.H., Siret, Y., Tournier, É., 1987. Calcul formel : systèmes et algorithmes de manipulations algébriques. Masson, Paris, France.
Fateman, R., 1974. Polynomial multiplication, powers and asymptotic analysis: some comments. SIAM J. Comput. 3 (3), 4–15.
Fateman, R., 2003. Comparing the speed of programs for sparse polynomial multiplication. SIGSAM Bull. 37 (1), 4–15.
Fürer, M., 2007. Faster integer multiplication. In: Proceedings of the Thirty-Ninth ACM Symposium on Theory of Computing (STOC 2007). ACM Press, pp. 57–66.
Fürer, M., 2009. Faster integer multiplication. SIAM J. Comput. 39 (3), 979–1005.
Gao, S., 2003. Factoring multivariate polynomials via partial differential equations. Math. Comp. 72 (242), 801–822.
Gao, S., Rodrigues, V.M., 2003. Irreducibility of polynomials modulo p via Newton polytopes. J. Number Theory 101 (1), 32–47.
Gao, S., Kaltofen, E., May, J., Yang, Z., Zhi, L., 2004. Approximate factorization of multivariate polynomials via differential equations. In: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation (ISSAC '04). ACM Press, pp. 167–174.
Garg, S., Schost, É., 2009. Interpolation of polynomials given by straight-line programs. Theoret. Comput. Sci. 410 (27–29), 2659–2662.
Gastineau, M., Laskar, J., 2006. Development of TRIP: Fast sparse multivariate polynomial multiplication using burst tries. In: Computational Science—ICCS 2006. In: Lecture Notes in Comput. Sci., vol. 3992. Springer-Verlag, pp. 446–453.
von zur Gathen, J., Gerhard, J., 2003. Modern Computer Algebra, second ed. Cambridge University Press.
Granlund, T., et al., 1991. GMP, the GNU multiple precision arithmetic library. Available from http://gmplib.org.
Grigoriev, D.Y., Karpinski, M., 1987. 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. IEEE, pp. 166–172.
Grigoriev, D.Y., Karpinski, M., Singer, M.F., 1990. Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields. SIAM J. Comput. 19 (6), 1059–1063.
Grigoriev, D., Karpinski, M., Singer, M.F., 1994. Computational complexity of sparse rational interpolation. SIAM J. Comput. 23 (1), 1–11.
Hardy, G.H., Wright, E.M., 1979. An Introduction to the Theory of Numbers. Oxford University Press.
van der Hoeven, J., 2002. Relax, but don't be too lazy. J. Symbolic Comput. 34 (6), 479–542.
van der Hoeven, J., 2004. The truncated Fourier transform and applications. In: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation (ISSAC '04). ACM Press, pp. 290–296.
van der Hoeven, J., 2010. Newton's method and FFT trading. J. Symbolic Comput. 45 (8).
van der Hoeven, J., 2012. On the complexity of polynomial reduction. Tech. rep., HAL, http://hal.archives-ouvertes.fr/hal-00658704.
van der Hoeven, J., Lecerf, G., 2012. On the complexity of multivariate blockwise polynomial multiplication. In: Proceedings of the 2012 International Symposium on Symbolic and Algebraic Computation (ISSAC '12). ACM Press, pp. 211–218.
van der Hoeven, J., et al., 2002. Mathemagix. http://www.mathemagix.org.
Huang, M.-D.A., Rao, A.J., 1999. Interpolation of sparse multivariate polynomials over large finite fields with applications. J. Algorithms 33 (2), 204–228.
Johnson, S.C., 1974. Sparse polynomial arithmetic. SIGSAM Bull. 8 (3), 63–71.
Kaltofen, E., Lakshman, Y.N., 1988. Improved sparse multivariate polynomial interpolation algorithms. In: Proceedings of the 1988 International Symposium on Symbolic and Algebraic Computation (ISSAC '88). ACM Press, pp. 467–474.
Kaltofen, E., Trager, B.M., 1990. 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.
Kaltofen, E., Saunders, B.D., 1991. On Wiedemann's method of solving sparse linear systems. In: Applied Algebra, Algebraic Algorithms and Error-Correcting Codes. In: Lecture Notes in Comput. Sci., vol. 539. Springer-Verlag, pp. 29–38.
Kaltofen, E., Lakshman, Y.N., Wiley, J.-M., 1990. Modular rational sparse multivariate polynomial interpolation. In: Proceedings of the 1990 International Symposium on Symbolic and Algebraic Computation (ISSAC '90). ACM Press, pp. 135–139.
Kaltofen, E., Lee, W., Lobo, A.A., 2000. Early termination in Ben-Or/Tiwari sparse interpolation and a hybrid of Zippel's algorithm. In: Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation (ISSAC '00). ACM Press, pp. 192–201.
Lasserre, J.B., Zeron, E.S., 2005. An alternative algorithm for counting lattice points in a convex polytope. Math. Oper. Res. 30 (3), 597–614.
Lecerf, G., 2007. Improved dense multivariate polynomial factorization algorithms. J. Symbolic Comput. 42 (4), 477–494.
Lecerf, G., Schost, É., 2003. Fast multivariate power series multiplication in characteristic zero. SADIO Electron. J. Inform. Oper. Res. 5 (1), 1–10.
Makino, K., Berz, M., 1996. Remainder differential algebras and their applications. In: Computational Differentiation: Techniques, Applications and Tools. SIAM, Philadelphia, pp. 63–74.
Makino, K., Berz, M., 2005. Suppression of the wrapping effect by Taylor model-based verified integrators: long-term stabilization by preconditioning. Int. J. Differ. Equ. Appl. 10 (4), 353–384.
Malaschonok, G.I., Satina, E.S., 2004. Fast multiplication and sparse structures. Program. Comput. Softw. 30 (2), 105–109.

Massey, J., Schaub, T., 1988. Linear complexity in coding theory. In: Cohen, G., Godlewski, P. (Eds.), Coding Theory and Applications. In: Lecture Notes in Comput. Sci., vol. 311. Springer-Verlag, pp. 19–32.

Matoušek, J., Schwarzkopf, O., 1992. Linear optimization queries. In: Proceedings of the Eighth Annual Symposium on Computational Geometry (SCG '92). ACM Press, pp. 16–25.

Moenck, R.T., 1976. Practical fast polynomial multiplication. In: Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation (SYMSAC '76). ACM Press, pp. 136–148.

Monagan, M., Pearce, R., 2007. Polynomial division using dynamic arrays, heaps, and packed exponent vectors. In: Computer Algebra in Scientific Computing (CASC 2007). In: Lecture Notes in Comput. Sci., vol. 4770. Springer-Verlag, pp. 295–315.

Monagan, M., Pearce, R., 2009. Parallel sparse polynomial multiplication using heaps. In: Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation (ISSAC '09). ACM Press, pp. 263–270.

Monagan, M., Pearce, R., 2010. Polynomial multiplication and division in Maple 14. ACM Commun. Comput. Algebra 44 (4), 205–209.

Ponder, C.G., 1991. Parallel multiplication and powering of polynomials. J. Symbolic Comput. 11 (4), 307–320.

Preparata, F.P., Shamos, M.I., 1985. Computational Geometry: An Introduction. Springer-Verlag.

Prony, R., 1795. Essai expérimental et analytique sur les lois de la dilatabilité des fluides élastiques et sur celles de la force expansive de la vapeur de l'eau et de la vapeur de l'alkool, à différentes températures. J. de l'École Polytechnique Floréal et Plairial, an III 1 (cahier 22), 24–76.

Schönhage, A., 1977. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. Acta Inform. 7, 395–398.

Schönhage, A., Strassen, V., 1971. Schnelle Multiplikation grosser Zahlen. Computing 7, 281–292.

Seidel, R., 1986. Constructing higher-dimensional convex hulls at logarithmic cost per face. In: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing (STOC '86). ACM Press, pp. 404–413.

Shaker, H., 2009. Topology and factorization of polynomials. Math. Scand. 104 (1), 51–59.

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

Storjohann, A., 2000. Algorithms for matrix canonical forms. PhD thesis, ETH, Zürich, Switzerland, http://www.scg.uwaterloo.ca/~astorjoh/publications.html.

Stoutemyer, D.R., 1984. Which polynomial representation is best? In: Proceedings of the 1984 MACSYMA Users' Conference. Schenectady, NY, July 23–25. General Electric, pp. 221–243.

Strassen, V., 1973. Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoeffizienten. Numer. Math. 20, 238–251.

Weimann, M., 2010. A lifting and recombination algorithm for rational factorization of sparse polynomials. J. Complexity 26 (6), 608–628.

Weimann, M., 2012. Algebraic osculation and application to factorization of sparse polynomials. Found. Comput. Math. 12, 173–201.

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

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

Zanoni, A., 2009. Iterative Karatsuba for multivariate polynomial multiplication. Preprint No. 624. Centro Interdipartimentale "Vito Volterra", Università di Roma "Tor Vergata".

Zippel, R., 1979. Probabilistic algorithms for sparse polynomials. In: Symbolic and Algebraic Computation. In: Lecture Notes in Comput. Sci., vol. 72. Springer-Verlag, pp. 216–226.