Fast arithmetic for triangular sets: from theory to practice

Xin Li, Marc Moreno Maza, Éric Schost

Computer Science Department, The University of Western Ontario, London, Ontario, Canada

Abstract

We study arithmetic operations for triangular families of polynomials, concentrating on multiplication in dimension zero. By a suitable extension of fast univariate Euclidean division, we obtain theoretical and practical improvements over a direct recursive approach; for a family of special cases, we reach quasi-linear complexity. The main outcome we have in mind is the acceleration of higher-level algorithms, by interfacing our low-level implementation with languages such as AXIOM or MAPLE. We show the potential for huge speed-ups, by comparing two AXIOM implementations of van Hoeij and Monagan's modular GCD algorithm.

1. Introduction

Triangular representations are a useful data structure for dealing with a variety of problems, from computations with algebraic numbers to the symbolic solution of polynomial or differential systems. At the core of the algorithms for these objects, one finds a few basic operations, such as multiplication and division in dimension zero. Higher-level algorithms can be built on these subroutines, using for instance modular algorithms and lifting techniques (10).

Our goal in this article is twofold. First, we study algorithms for multiplication modulo a triangular set in dimension zero. All known algorithms involve an overhead exponential in the number n of variables; we show how to reduce this overhead in the general case, and how to remove it altogether in a special case. Our second purpose is to demonstrate how the combination of such fast algorithms and low-level implementation can readily improve the performance of environments like AXIOM or MAPLE in a significant manner, for a variety of higher-level algorithms. We illustrate this through the example of van Hoeij and Monagan's modular GCD algorithm for number fields (22).

⋆ This research was partly supported by NSERC and MITACS of Canada; the third author is also supported by the Canada Research Chair Program.

Email addresses: xli96@csd.uwo.ca (Xin Li,), moreno@csd.uwo.ca (Marc Moreno Maza,), eschost@uwo.ca (Éric Schost).

Triangular sets. Triangular representations are versatile data structures. The zero-dimensional case is discussed in detail (with worked-out examples) in (18); a general introduction (including positive dimensional situations) is given in (1). In this article, we adopt the following convention: a *monic triangular set* is a family of polynomials $\mathbf{T} = (T_1, \ldots, T_n)$ in $R[X_1, \ldots, X_n]$, where R is a commutative ring with 1. For all i, we impose that T_i is in $R[X_1, \ldots, X_i]$, is *monic* in X_i and *reduced* with respect to $T_1, \ldots, T_{i-1}$. Since all our triangular sets will be monic, we simply call them triangular sets.

The natural approach to arithmetic modulo triangular sets is recursive: to work in the residue class ring $\mathbb{L}_{\mathbf{T}} = R[X_1, \ldots, X_n]/\langle T_1, \ldots, T_n\rangle$, we regard it as $\mathbb{L}_{\mathbf{T}_-}[X_n]/\langle T_n\rangle$, where $\mathbb{L}_{\mathbf{T}_-}$ is the ring $R[X_1, \ldots, X_{n-1}]/\langle T_1, \ldots, T_{n-1}\rangle$. This point of view allows one to design elegant recursive algorithms, whose complexity is often easy to analyze, and which can be implemented in a straightforward manner in high-level languages such as AXIOM or MAPLE (19). However, as shown below, this approach is not necessarily optimal, regarding both complexity and practical performance.

Complexity issues. The core of our problematic is *modular multiplication*: given A and B in the residue class ring $\mathbb{L}_{\mathbf{T}}$, compute their product; here, one assumes that the input and output are reduced with respect to the polynomials $\mathbf{T}$. Besides, one can safely suppose that all degrees are at least 2 (see the discussion in the next section).

In one variable, the usual approach consists in multiplying A and B and reducing them by Euclidean division. Using classical arithmetic, the cost is about $2d_1^2$ multiplications and $2d_1^2$ additions in R, with $d_1 = \deg(T_1, X_1)$. Using fast arithmetic, polynomial multiplication becomes essentially linear, the best known result ((7), after (28; 27)) being of the form $\mathsf{k}\, d_1 \lg(d_1) \lg\lg(d_1)$, with k a constant and $\lg(x) = \log_2 \max(2, x)$. A Euclidean division can then be reduced to two polynomial multiplications, using Cook-Sieveking-Kung's algorithm (8; 31; 16). In n variables, the measure of complexity is $\delta_{\mathbf{T}} = \deg(T_1, X_1) \cdots \deg(T_n, X_n)$, since representing a polynomial modulo $\mathbf{T}$ requires storing $\delta_{\mathbf{T}}$ elements. Then, applying the previous results recursively leads to bounds of order $2^n \delta_{\mathbf{T}}^2$ for the standard approach, and $(3\mathsf{k}+1)^n \delta_{\mathbf{T}}$ for the fast one, neglecting logarithmic factors and lower-order terms. An important difference with the univariate case is the presence of the overheads 2^n and $(3\mathsf{k}+1)^n$, which cannot be absorbed in a big-Oh estimate anymore (unless n is bounded).

Improved algorithms and the virtues of fast arithmetic. Our first contribution is the design and implementation of a faster algorithm: while still relying on the techniques of fast Euclidean division, we show in Theorem 1 that a mixed dense / recursive approach yields a cost of order $4^n \delta_{\mathbf{T}}$, neglecting again all lower order terms and logarithmic factors; this is better than the previous bound for $\delta_{\mathbf{T}} \geq 2^n$. Building upon previous work (11), the implementation is done in C, and is dedicated to small finite field arithmetic.

The algorithm uses fast polynomial multiplication and Euclidean division. For univariate polynomials over $\mathbb{F}_p$, such fast algorithms become advantageous for degrees of approximately 100. In a worst-case scenario, this may suggest that for multivariate polynomials, fast algorithms become useful when the partial degree in *each variable* is at least 100, which would be a severe restriction. Our second contribution is to contradict this expectation, by showing that the cut-off values for which the fast algorithm becomes advantageous *decrease* with the number of variables.

A quasi-linear algorithm for a special case. We next discuss a particular case, where all polynomials in the triangular set are actually univariate, that is, with T_i in $\mathbb{K}[X_i]$ for

all i. Despite its apparent simplicity, this problem already contains non-trivial questions, such as power series multiplication modulo $\langle X_1^{d_1}, \ldots, X_n^{d_n} \rangle$, taking $T_i = X_i^{d_i}$.

For the question of power series multiplication, no quasi-linear algorithm was known until (29). We extend this result to the case of arbitrary $T_i \in \mathbb{K}[X_i]$, the crucial question being how to *avoid* expanding the (polynomial) product AB before reducing it. Precisely, we prove that for $\mathbb{K}$ of cardinality greater than, or equal to, $\max_{i \leq n} d_i$, and for $\varepsilon > 0$, there exists a constant K_ε such that for all n, products modulo $\langle T_1(X_1), \ldots, T_n(X_n) \rangle$ can be done in at most $\mathsf{K}_\varepsilon \delta_{\mathbf{T}}^{1+\varepsilon}$ operations, with $\delta_{\mathbf{T}}$ as before.

Following (3; 4; 2; 29), the algorithm uses deformation techniques, and is unfortunately not expected to be very practical, except e.g. when all degrees equal 2. However, this shows that for a substantial family of examples, and in suitable (large enough) fields, one can suppress the exponential overhead seen above. Generalizing this result to an arbitrary $\mathbf{T}$ is a major open problem.

Applications to higher-level algorithms. Fast arithmetic for basic operations modulo a triangular set is fundamental for a variety of higher-level operations. By embedding fast arithmetic in high-level environments like AXIOM (see (11; 21)) or MAPLE, one can obtain a substantial speed-up for questions ranging from computations with algebraic numbers (GCD, factorization) to polynomial system solving via triangular decomposition, such as in the algorithm of (24), which is implemented in AXIOM and MAPLE (19).

Our last contribution is to demonstrate such a speed-up on the example of van Hoeij and Monagan's algorithm for GCD computation in number fields. This algorithm is modular, most of the effort consisting in GCD computations over small finite fields. We compare a direct AXIOM implementation to one relying on our low-level C implementation, and obtain improvement of orders of magnitude.

Outline of the paper. Section 2 presents our multiplication algorithms, for general triangular sets and triangular sets consisting of univariate polynomials. We next describe our implementation in Section 3; experiments and comparisons with other systems are given in Section 4.

Acknowledgments. We thank R. Rasheed for his help with obtaining the MAPLE timings, and the referees for their helpful remarks. This article is an expanded version of (20).

2. Algorithms

We describe here our main algorithm. It relies on the Cook-Sieveking-Kung idea but differs from a direct recursive implementation: recalling that we handle multivariate polynomials makes it possible to base our algorithm on fast multivariate multiplication.

2.1. Notation and preliminaries

Notation. Triangular sets will be written as $\mathbf{T} = (T_1, \ldots, T_n)$. The multi-degree of a triangular set $\mathbf{T}$ is the n-uple $d_i = \deg(T_i, X_i)_{1 \leq i \leq n}$. We will write $\delta_{\mathbf{T}} = d_1 \cdots d_n$; in Subsection 2.3, we will use the notation $r_{\mathbf{T}} = \sum_{i=1}^{n} (d_i - 1) + 1$. Writing $\mathbf{X} = X_1, \ldots, X_n$, we let $\mathbb{L}_{\mathbf{T}}$ be the residue class ring $R[\mathbf{X}]/\langle \mathbf{T} \rangle$, where R is our base ring. Let $M_{\mathbf{T}}$ be the set of monomials $M_{\mathbf{T}} = \left\{ X_1^{e_1} \cdots X_n^{e_n} \mid 0 \leq e_i < d_i \text{ for all } i \right\}$; then, because of our monicity assumption, the free R-submodule generated by $M_{\mathbf{T}}$ in $R[\mathbf{X}]$, written

$$\mathsf{Span}(M_{\mathbf{T}}) = \Big\{ \sum_{m \in M_{\mathbf{T}}} a_m m \mid a_m \in R \Big\},$$

is isomorphic to $\mathbb{L}_{\mathbf{T}}$. Hence, in our algorithms, elements of $\mathbb{L}_{\mathbf{T}}$ are represented on the monomial basis $M_{\mathbf{T}}$. Without loss of generality, we always assume that *all degrees d_i are at least 2*. Indeed, if T_i has degree 1 in X_i, the variable X_i appears neither in the monomial basis $M_{\mathbf{T}}$ nor in the other polynomials T_j, so one can express it as a function of the other variables, and T_i can be discarded.

Standard and fast modular multiplication. As said before, standard algorithms have a cost of roughly $2^n\delta_{\mathbf{T}}^2$ operations in R for multiplication in $\mathbb{L}_{\mathbf{T}}$. This bound seems not even polynomial in $\delta_{\mathbf{T}}$, due to the exponential overhead in n. However, since all degrees d_i are at least 2, $\delta_{\mathbf{T}}$ is at least 2^n; hence, any bound of the form $\mathsf{K}^n\delta_{\mathbf{T}}^{\ell}$ is actually polynomial in $\delta_{\mathbf{T}}$, since it is upper-bounded by $\delta_{\mathbf{T}}^{\log_2(\mathsf{K})+\ell}$.

Our goal is to obtain bounds of the form $\mathsf{K}^n\delta_{\mathbf{T}}$ (up to logarithmic factors), that are thus softly linear in $\delta_{\mathbf{T}}$ for fixed n; of course, we want the constant K as small as possible. We will use fast polynomial multiplication, denoting by $\mathsf{M} : \mathbb{N} \to \mathbb{N}$ a function such that over any ring, polynomials of degree less than d can be multiplied in $\mathsf{M}(d)$ operations, and which satisfies the super-linearity conditions of (12, Chapter 8). Using the algorithm of Cantor-Kaltofen (7), one can take $\mathsf{M}(d) \in O(d\log(d)\log\log(d))$. Precisely, we will denote by k a constant such that $\mathsf{M}(d) \leq \mathsf{k}\, d\lg(d)\lg\lg(d)$ holds for all d, with $\lg(d) = \log_2\max(d, 2)$

In one variable, fast modular multiplication is done using the Cook-Sieveking-Kung algorithm (8; 31; 16). Given T_1 monic of degree d_1 in $R[X_1]$ and A, B of degrees less than d_1, one computes first the product AB. To perform the Euclidean division $AB = QT_1+C$, one first computes the inverse $S_1 = U_1^{-1} \bmod X_1^{d_1-1}$, where $U_1 = X_1^{d_1}T_1(1/X_1)$ is the reciprocal polynomial of T_1. This is done using Newton iteration, and can be performed as a precomputation, for a cost of $3\mathsf{M}(d_1)+O(d_1)$. One recovers first the reciprocal of Q, then the remainder C, using two polynomial products. Taking into account the cost of computing AB, but leaving out precomputations, these operations have cost $3\mathsf{M}(d_1)+d_1$. Applying this result recursively leads to a rough upper bound of $\prod_{i\leq n}(3\mathsf{M}(d_i)+d_i)$ for a product in $\mathbb{L}_{\mathbf{T}}$, without taking into account the similar cost of precomputation (see (17) for similar considerations); this gives a total estimate of roughly $(3\mathsf{k}+1)^n\delta_{\mathbf{T}}$, neglecting logarithmic factors.

One can reduce the $(3\mathsf{k}+1)^n$ overhead: since additions and constant multiplications in $\mathbb{L}_{\mathbf{T}}$ can be done in linear time, it is the *bilinear* cost of univariate multiplication which governs the overall cost. Over a field of large enough cardinality, using evaluation / interpolation techniques, univariate multiplication in degree less than d can be done using $2d-1$ bilinear multiplications; this yields estimates of rough order $(3\times 2)^n\delta_{\mathbf{T}} = 6^n\delta_{\mathbf{T}}$. Studying more precisely the multiplication process, we prove in Theorem 1 that one can compute products in $\mathbb{L}_{\mathbf{T}}$ using at most $\mathsf{K}\,4^n\delta_{\mathbf{T}}\lg(\delta_{\mathbf{T}})\lg\lg(\delta_{\mathbf{T}})$ operations, for a universal constant K. This is a synthetic but rough upper bound; we give more precise estimates within the proof. Obtaining results linear in $\delta_{\mathbf{T}}$, without an exponential factor in n, is a major open problem. When the base ring is a field of large enough cardinality, we obtain first results in this direction in Theorem 2: in the case of families of *univariate* polynomials, we present an algorithm of quasi-linear complexity $\mathsf{K}_\varepsilon\delta_{\mathbf{T}}^{1+\varepsilon}$ for all ε.

Basic complexity considerations. Since we are estimating costs that depend on an *a priori* unbounded number of parameters, big-Oh notation is delicate to handle. We rather use explicit inequalities when possible, all the more as an explicit control is required in the proof of Theorem 2. For similar reasons, we do not use $O^{\tilde{}}$ notation.

We denote by $\mathsf{C}_{\mathsf{Eval}}$ (resp. $\mathsf{C}_{\mathsf{Interp}}$) functions such that over any ring R, a polynomial of degree less than d can be evaluated (resp. interpolated) at d points $a_0, \ldots, a_{d-1}$ in $\mathsf{C}_{\mathsf{Eval}}(d)$ (resp. $\mathsf{C}_{\mathsf{Interp}}(d)$) operations, assuming $a_i - a_j$ is a unit for $i \neq j$ for interpolation. From (12, Chapter 10), we can take both quantities in $O(\mathsf{M}(d)\lg(d))$, where the constant in the big-Oh is universal. In Subsection 2.3, we will assume without loss of generality that $\mathsf{M}(d) \leq \mathsf{C}_{\mathsf{Eval}}(d)$ for all d.

Recall that k is such that $\mathsf{M}(d)$ is bounded by $\mathsf{k}\, d \lg(d) \lg\lg(d)$ for all d. Up to maybe increasing k, we will thus assume that both $\mathsf{C}_{\mathsf{Eval}}(d)$ and $\mathsf{C}_{\mathsf{Interp}}(d)$ are bounded by $\mathsf{k}\, d \lg^2(d) \lg\lg(d)$. Finally, we let $\mathsf{MM}(d_1, \ldots, d_n)$ be such that over any ring R, polynomials in $R[X_1, \ldots, X_n]$ of degree in X_i less than d_i for all i can be multiplied in $\mathsf{MM}(d_1, \ldots, d_n)$ operations. One can take

$$\mathsf{MM}(d_1, \ldots, d_n) \leq \mathsf{M}((2d_1 - 1) \cdots (2d_n - 1))$$

using Kronecker's substitution. Let $\delta = d_1 \cdots d_n$. Assuming $d_i \geq 2$ for all i, we deduce the inequalities

$$(2d_1 - 1) \cdots (2d_n - 1) \leq 2^n \delta \leq \delta^2,$$

which imply that $\mathsf{MM}(d_1, \ldots, d_n)$ admits the upper bound

$$\mathsf{k} 2^n \delta \lg(2^n \delta) \lg\lg(2^n \delta) \;\; \leq \;\; 4\mathsf{k} 2^n \delta \lg(\delta) \lg\lg(\delta).$$

Up to replacing k by $4\mathsf{k}$, we thus have

$$\delta \leq \mathsf{MM}(d_1, \ldots, d_n) \leq \mathsf{k}\, 2^n \delta \lg(\delta) \lg\lg(\delta). \tag{1}$$

Pan (25) proposed an alternative algorithm, that requires the existence of interpolation points in the base ring. This algorithm is more efficient when e.g. d_i are fixed and $n \to \infty$. However, using it below would not bring any improvement, due to our simplifications.

2.2. The main algorithm

Theorem 1. *There exists a constant K such that the following holds. Let R be a ring and let $\mathbf{T}$ be a triangular set in $R[\mathbf{X}]$. Given A, B in $\mathbb{L}_{\mathbf{T}}$, one can compute $AB \in \mathbb{L}_{\mathbf{T}}$ in at most $\mathsf{K}\, 4^n \delta_{\mathbf{T}} \lg(\delta_{\mathbf{T}}) \lg\lg(\delta_{\mathbf{T}})$ operations $(+, \times)$ in R.*

Proof. Let $\mathbf{T} = (T_1, \ldots, T_n)$ be a triangular set of multi-degree $(d_1, \ldots, d_n)$ in $R[\mathbf{X}] = R[X_1, \ldots, X_n]$. We then introduce the following objects:

- We write $\mathbf{T}_- = (T_1, \ldots, T_{n-1})$, so that $\mathbb{L}_{\mathbf{T}_-} = R[X_1, \ldots, X_{n-1}]/\langle \mathbf{T}_- \rangle$.
- For $i \leq n$, the polynomial $U_i = X_i^{d_i} T_i(X_1, \ldots, X_{i-1}, 1/X_i)$ is the reciprocal polynomial of T_i; S_i is the inverse of U_i modulo $\langle T_1, \ldots, T_{i-1}, X_i^{d_i - 1} \rangle$. We write $\mathbf{S} = (S_1, \ldots, S_n)$ and $\mathbf{S}_- = (S_1, \ldots, S_{n-1})$.

Two subroutines are used, which we describe in Figure 1. In these subroutines, we use the following notation:

- For D in $R[X_1, \ldots, X_i]$ such that $\deg(D, X_i) \leq e$, $\mathsf{Rev}(D, X_i, e)$ is the reciprocal polynomial $X_i^e D(X_1, \ldots, X_{i-1}, 1/X_i)$.
- For D in $R[\mathbf{X}]$, $\mathsf{Coeff}(D, X_i, e)$ is the coefficient of X_i^e.

We can now give the specification of these auxiliary algorithms. These algorithms make some assumptions, that will be satisfied when we call them from our main routine.

- The first one is $\mathsf{Rem}(A, \mathbf{T}, \mathbf{S})$, with A in $R[\mathbf{X}]$. This algorithm computes the normal form of A modulo $\mathbf{T}$, assuming that $\deg(A, X_i) \leq 2d_i - 2$ holds for all i. When $n = 0$, A is in R, $\mathbf{T}$ is empty and $\mathsf{Rem}(A, \mathbf{T}, \mathbf{S}) = A$.

$\underline{\mathsf{Rem}(A, \mathbf{T}, \mathbf{S})}$
1 if $n = 0$ return A
2 $A' \leftarrow \sum_{i=0}^{2d_n-2} \mathsf{Rem}(\mathsf{Coeff}(A, X_n, i), \mathbf{T}_-, \mathbf{S}_-)X_n^i$
3 $B \leftarrow \mathsf{Rev}(A', X_n, 2d_n - 2) \bmod X_n^{d_n-1}$
4 $P \leftarrow \mathsf{MulTrunc}(B, S_n, \mathbf{T}_-, \mathbf{S}_-, d_n - 1)$
5 $Q \leftarrow \mathsf{Rev}(P, X_n, d_n - 2)$
6 return $A' \bmod X_n^{d_n} - \mathsf{MulTrunc}(Q, T_n, \mathbf{T}_-, \mathbf{S}_-, d_n)$

$\underline{\mathsf{MulTrunc}(A, B, \mathbf{T}, \mathbf{S}, d_{n+1})}$
1 $C \leftarrow AB$
2 if $n = 0$ return $C \bmod X_1^{d_1}$
3 return $\sum_{i=0}^{d_{n+1}-1} \mathsf{Rem}(\mathsf{Coeff}(C, X_{n+1}, i), \mathbf{T}, \mathbf{S})X_{n+1}^i$

Fig. 1. Algorithms Rem and MulTrunc.

- The next subroutine is $\mathsf{MulTrunc}(A, B, \mathbf{T}, \mathbf{S}, d_{n+1})$, with A, B in $R[\mathbf{X}, X_{n+1}]$; it computes the product AB modulo $\langle \mathbf{T}, X_{n+1}^{d_{n+1}} \rangle$, assuming that $\deg(A, X_i)$ and $\deg(B, X_i)$ are bounded by $d_i - 1$ for $i \leq n + 1$. If $n = 0$, $\mathbf{T}$ is empty, so this function return $AB \bmod X_1^{d_1}$.

To compute $\mathsf{Rem}(A, \mathbf{T}, \mathbf{S})$, we use the Cook-Sieveking-Kung idea in $\mathbb{L}_{\mathbf{T}_-}[X_n]$: we reduce all coefficients of A modulo $\mathbf{T}_-$ and perform two truncated products in $\mathbb{L}_{\mathbf{T}_-}[X_n]$ using MulTrunc. The operation MulTrunc is performed by multiplying A and B as polynomials, truncating in X_{n+1} and reducing all coefficients modulo $\mathbf{T}$, using Rem.

For the complexity analysis, assuming for a start that all inverses $\mathbf{S}$ have been precomputed, we write $\mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_n)$ for an upper bound on the cost of $\mathsf{Rem}(A, \mathbf{T}, \mathbf{S})$ and $\mathsf{C}_{\mathsf{MulTrunc}}(d_1, \ldots, d_{n+1})$ for a bound on the cost of $\mathsf{MulTrunc}(A, B, \mathbf{T}, \mathbf{S}, d_{n+1})$. Setting $\mathsf{C}_{\mathsf{Rem}}(\,) = 0$, the previous algorithms imply the estimates

$$\begin{aligned}
\mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_n) \;&\leq\; (2d_n - 1)\mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_{n-1}) \;+\; \mathsf{C}_{\mathsf{MulTrunc}}(d_1, \ldots, d_n - 1) \\
&\quad + \mathsf{C}_{\mathsf{MulTrunc}}(d_1, \ldots, d_n) \;+\; d_1 \cdots d_n; \\
\mathsf{C}_{\mathsf{MulTrunc}}(d_1, \ldots, d_n) \;&\leq\; \mathsf{MM}(d_1, \ldots, d_n) + \mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_{n-1})d_n.
\end{aligned}$$

Assuming that $\mathsf{C}_{\mathsf{MulTrunc}}$ is non-decreasing in each d_i, we deduce the upper bound

$$\mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_n) \quad\leq\quad 4\mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_{n-1})d_n \;+\; 2\mathsf{MM}(d_1, \ldots, d_n) \;+\; d_1 \cdots d_n,$$

for $n \geq 1$. Write $\mathsf{MM}'(d_1, \ldots, d_n) = 2\mathsf{MM}(d_1, \ldots, d_n) + d_1 \cdots d_n$. This yields

$$\mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_n) \;\leq\; \sum_{i=1}^{n} 4^{n-i}\; \mathsf{MM}'(d_1, \ldots, d_i)d_{i+1} \cdots d_n,$$

since $\mathsf{C}_{\mathsf{Rem}}(\,) = 0$. In view of the bound on MM given in Equation (1), we obtain

$$\mathsf{MM}'(d_1, \ldots, d_i)d_{i+1} \cdots d_n \;\leq\; 3\mathsf{k}2^i\delta_{\mathbf{T}} \lg(\delta_{\mathbf{T}}) \lg\lg(\delta_{\mathbf{T}}).$$

Taking e.g. $\mathsf{K} = 3\mathsf{k}$ gives the bound $\mathsf{C}_{\mathsf{Rem}}(d_1, \ldots, d_n) \leq \mathsf{K}\, 4^n\delta_{\mathbf{T}} \lg(\delta_{\mathbf{T}}) \lg\lg(\delta_{\mathbf{T}})$. The product $A, B \mapsto AB$ in $\mathbb{L}_{\mathbf{T}}$ is performed by multiplying A and B as polynomials and

returning $\mathsf{Rem}(AB, \mathbf{T}, \mathbf{S})$. Hence, the cost of this operation admits a similar bound, up to replacing K by $\mathsf{K} + \mathsf{k}$. This concludes our cost analysis, excluding the cost of the precomputations. We now estimate the cost of precomputing the inverses $\mathbf{S}$: supposing that $S_1, \ldots, S_{n-1}$ are known, we detail the cost of computing S_n. Our upper bound on $\mathsf{C}_{\mathsf{MulTrunc}}$ shows that, assuming $S_1, \ldots, S_{n-1}$ are known, one multiplication modulo $X_n^{d'_n}$ in $\mathbb{L}_{\mathbf{T}_-}[X_n]$ can be performed in

$$\mathsf{k} 2^n \delta' \lg(\delta') \lg\lg(\delta') + \mathsf{K}\, 4^n \delta' \lg(\delta_{\mathbf{T}_-}) \lg\lg(\delta_{\mathbf{T}_-})$$

operations, with $\delta_{\mathbf{T}_-} = d_1 \cdots d_{n-1}$ and $\delta' = \delta_{\mathbf{T}_-} d'_n$. Up to replacing K by $\mathsf{K} + \mathsf{k}$, and assuming $d'_n \leq d_n$, this yields the upper bound $\mathsf{K}\, 4^n \delta' \lg(\delta_{\mathbf{T}}) \lg\lg(\delta_{\mathbf{T}})$. Let now $\ell = \lceil \log_2(d_n - 1) \rceil$. Using Newton iteration in $\mathbb{L}_{\mathbf{T}_-}[X_n]$, we obtain S_n by performing 2 multiplications in $\mathbb{L}_{\mathbf{T}_-}[X_n]$ in degrees less than m and $m/2$ negations, for $m = 2, 4, \ldots, 2^{\ell-1}$, see (12, Chapter 9). By the remark above, the cost is at most

$$\mathsf{t}(n) \;=\; \sum_{m=2,\ldots,2^{\ell-1}} 3\mathsf{K}\, 4^n d_1 \cdots d_{n-1} m \lg(\delta_{\mathbf{T}}) \lg\lg(\delta_{\mathbf{T}}) \;\leq\; 3\mathsf{K}\, 4^n \delta_{\mathbf{T}} \lg(\delta_{\mathbf{T}}) \lg\lg(\delta_{\mathbf{T}}).$$

The sum $\mathsf{t}(1) + \cdots + \mathsf{t}(n)$ bounds the total precomputation time; one sees that it admits a similar form of upper bound. Up to increasing K, this gives the desired result. □

2.3. The case of univariate polynomials

To suppress the exponential overhead, it is necessary to avoid expanding the product AB. We discuss here the case of triangular sets consisting of univariate polynomials, where this is possible. We provide a quasi-linear algorithm, that works under mild assumptions. However, the techniques used (deformation ideas, coming from fast matrix multiplication algorithms (3; 4; 2)) induce large sublinear factors.

Theorem 2. *For any $\varepsilon > 0$, there exists a constant K_ε such that the following holds. Let $\mathbb{K}$ be a field and $\mathbf{T} = (T_1, \ldots, T_n)$ be a triangular set of multi-degree $(d_1, \ldots, d_n)$ in $\mathbb{K}[X_1] \times \cdots \times \mathbb{K}[X_n]$, with $2 \leq d_i \leq |\mathbb{K}|$ for all i. Given A, B in $\mathbb{L}_{\mathbf{T}}$, one can compute $AB \in \mathbb{L}_{\mathbf{T}}$ using at most $\mathsf{K}_\varepsilon\, \delta_{\mathbf{T}}^{1+\varepsilon}$ operations $(+, \times, \div)$ in $\mathbb{K}$.*

Step 1. We start by a special case. Let $\mathbf{T} = (T_1, \ldots, T_n)$ be a triangular set of multi-degree $(d_1, \ldots, d_n)$; for later applications, we suppose that it has coefficients in a ring R. Our main assumption is that for all i, T_i is in $R[X_i]$ and factors as

$$T_i = (X_i - \alpha_{i,0}) \cdots (X_i - \alpha_{i,d_i-1}),$$

with $\alpha_{i,j} - \alpha_{i,j'}$ a unit in R for $j \neq j$'. Let $V \subset R^n$ be the grid

$$V = [\ (\alpha_{1,\ell_1}, \ldots, \alpha_{n,\ell_n}) \mid 0 \leq \ell_i < d_i\ ],$$

which is the zero-set of $(T_1, \ldots, T_n)$ (when the base ring is a domain). Remark that T_i and T_j can have non-trivial common factors: all that matters is that for a given i, evaluation and interpolation at the roots of T_i is possible.

Proposition 3. *Given A, B in $\mathbb{L}_{\mathbf{T}}$, as well as the set of points V, one can compute $AB \in \mathbb{L}_{\mathbf{T}}$ using at most*

$$\delta_{\mathbf{T}} \left(1 + \sum_{i \leq n} \frac{2\mathsf{C}_{\mathsf{Eval}}(d_i) + \mathsf{C}_{\mathsf{Interp}}(d_i)}{d_i}\right)$$

operations $(+, \times, \div)$ *in* R.

In view of our remarks on the costs of evaluation and interpolation, this latter cost is at most $\mathsf{K}'\,\delta_{\mathbf{T}}\,\lg^2(\delta_{\mathbf{T}})\lg\lg(\delta_{\mathbf{T}})$, for a universal constant K', which can be taken as $\mathsf{K}' = 3\mathsf{k}{+}1$.

Proof. The proof uses an evaluation / interpolation process. Define the evaluation map

$$\begin{aligned}\mathsf{Eval} : \mathsf{Span}(M_{\mathbf{T}}) &\to R^{\delta_{\mathbf{T}}}\\ F &\mapsto [F(\alpha) \mid \alpha \in V].\end{aligned}$$

Since all $\alpha_{i,j} - \alpha_{i,j'}$ are units, the map Eval is invertible. To perform evaluation and interpolation, we use the algorithm in (25, Section 2), which generalizes the multidimensional Fourier Transform: to evaluate F, we see it as a polynomial in $\mathbb{K}[X_1, \ldots, X_{n-1}][X_n]$, and evaluate recursively its coefficients at $V' = [(\alpha_{1,\ell_1}, \ldots, \alpha_{n-1,\ell_{n-1}}) \mid 0 \le \ell_i < d_i]$. We conclude by performing $d_1 \cdots d_{n-1}$ univariate evaluations in X_n in degree d_n.

Extending our previous notation, we immediately deduce the recursion for the cost $\mathsf{C}_{\mathsf{Eval}}$ of multivariate evaluation

$$\mathsf{C}_{\mathsf{Eval}}(d_1, \ldots, d_n) \le \mathsf{C}_{\mathsf{Eval}}(d_1, \ldots, d_{n-1})\, d_n + d_1 \cdots d_{n-1}\mathsf{C}_{\mathsf{Eval}}(d_n),$$

$$\text{so that} \quad \mathsf{C}_{\mathsf{Eval}}(d_1, \ldots, d_n) \;\le\; \delta_{\mathbf{T}} \sum_{i \le n} \frac{\mathsf{C}_{\mathsf{Eval}}(d_i)}{d_i}.$$

The inverse map of Eval is the interpolation map Interp. Again, we use Pan's algorithm; the recursion and the bounds for the cost are the same, yielding

$$\mathsf{C}_{\mathsf{Interp}}(d_1, \ldots, d_n) \;\le\; \delta_{\mathbf{T}} \sum_{i \le n} \frac{\mathsf{C}_{\mathsf{Interp}}(d_i)}{d_i}.$$

To compute $AB \bmod \mathbf{T}$, it suffices to evaluate A and B on V, multiply the $\delta_{\mathbf{T}}$ pairs of values thus obtained, and interpolate the result. The cost estimate follows. □

This algorithm is summarized in Figure 2, under the name $\mathsf{MulSplit}$ (since it refers to triangular sets which completely split into linear factors).

```
MulSplit(A, B, V)
1 Val_A ← Eval(A)
2 Val_B ← Eval(B)
3 Val_C ← [ Val_A(α)Val_B(α) | α ∈ V ]
4 return Interp(Val_C)
```

Fig. 2. Algorithm MulSplit.

Step 2. We continue with the case where the polynomials T_i do not split anymore. Recall our definition of the integer $r_{\mathbf{T}} = \sum_{i=1}^{n}(d_i - 1) + 1$; since the polynomials $\mathbf{T}$ form a Gröbner basis for any order, $r_{\mathbf{T}}$ is the regularity of the ideal $\langle \mathbf{T} \rangle$. In the following, the previous exponential overhead disappears, but we introduce a quasi-linear dependency in $r_{\mathbf{T}}$: these bounds are good for triangular sets made of many polynomials of low degree.

Proposition 4. *Under the assumptions of Theorem 2, given A, B in $\mathbb{L}_{\mathbf{T}}$, one can compute the product $AB \in \mathbb{L}_{\mathbf{T}}$ using at most*

$$\mathsf{k}'\ \delta_{\mathbf{T}}\ \mathsf{M}(r_{\mathbf{T}})\ \sum_{i \le n} \frac{\mathsf{C}_{\mathsf{Eval}}(d_i) + \mathsf{C}_{\mathsf{Interp}}(d_i)}{d_i},$$

operations $(+, \times, \div)$ in $\mathbb{K}$, for a universal constant k'.

As before, there exists a universal constant K'' such that this estimate simplifies as

$$\mathsf{K}''\, \delta_{\mathbf{T}}\, r_{\mathbf{T}}\, \big(\lg(\delta_{\mathbf{T}})\lg(r_{\mathbf{T}})\big)^3. \tag{2}$$

Proof. Let $\mathbf{T} = (T_1, \ldots, T_n)$ be a triangular set with T_i in $\mathbb{K}[X_i]$ of degree d_i for all i. Let $\mathbf{U} = (U_1, \ldots, U_n)$ be the polynomials

$$U_i = (X_i - a_{i,0}) \cdots (X_i - a_{i,d_i-1}),$$

where for fixed i, the values $a_{i,j}$ are pairwise distinct (these values exist due to our assumption on the cardinality of $\mathbb{K}$). Let finally η be a new variable, and define $\mathbf{V} = (V_1, \ldots, V_n) \subset \mathbb{K}[\eta][\mathbf{X}]$ by $V_i = \eta T_i + (1-\eta)U_i$, so that V_i is monic of degree d_i in $\mathbb{K}[\eta][X_i]$. Remark that the monomial bases $M_{\mathbf{T}}$, $M_{\mathbf{U}}$ and $M_{\mathbf{V}}$ are all the same, that specializing η at 1 in $\mathbf{V}$ yields $\mathbf{T}$ and that specializing η at 0 in $\mathbf{V}$ yields $\mathbf{U}$.

Lemma 5. *Let A, B be in $\mathsf{Span}(M_{\mathbf{T}})$ in $\mathbb{K}[\mathbf{X}]$ and let $C = AB \bmod \langle \mathbf{V} \rangle$ in $\mathbb{K}[\eta][\mathbf{X}]$. Then C has degree in η at most $r_{\mathbf{T}} - 1$, and $C(1, \mathbf{X})$ equals AB modulo $\langle \mathbf{T} \rangle$.*

Proof. Fix an arbitrary order on the elements of $M_{\mathbf{T}}$, and let $\mathsf{Mat}(X_i, \mathbf{V})$ and $\mathsf{Mat}(X_i, \mathbf{T})$ be the multiplication matrices of X_i modulo respectively $\langle \mathbf{V} \rangle$ and $\langle \mathbf{T} \rangle$ in this basis. Hence, $\mathsf{Mat}(X_i, \mathbf{V})$ has entries in $\mathbb{K}[\eta]$ of degree at most 1, and $\mathsf{Mat}(X_i, \mathbf{T})$ has entries in $\mathbb{K}$. Besides, specializing η at 1 in $\mathsf{Mat}(X_i, \mathbf{V})$ yields $\mathsf{Mat}(X_i, \mathbf{T})$. The coordinates of $C = AB \bmod \langle \mathbf{V} \rangle$ on the basis $M_{\mathbf{T}}$ are obtained by multiplying the coordinates of B by the matrix $\mathsf{Mat}(A, \mathbf{V})$ of multiplication by A modulo $\langle \mathbf{V} \rangle$. This matrix equals $A(\mathsf{Mat}(X_1, \mathbf{V}), \ldots, \mathsf{Mat}(X_n, \mathbf{V}))$; hence, specializing its entries at 1 gives the matrix $\mathsf{Mat}(A, \mathbf{T})$, proving our last assertion. To conclude, observe that since A has total degree at most $r_{\mathbf{T}} - 1$, the entries of $\mathsf{Mat}(A, \mathbf{V})$ have degree at most $r_{\mathbf{T}} - 1$ as well. □

Let R be the ring $\mathbb{K}[\eta]/\langle \eta^{r_{\mathbf{T}}} \rangle$ and let A, B be in $\mathsf{Span}(M_{\mathbf{T}})$ in $\mathbb{K}[\mathbf{X}]$. Define $C_\eta = AB \bmod \langle \mathbf{V} \rangle$ in $R[\mathbf{X}]$ and let C be its canonical preimage in $\mathbb{K}[\eta][\mathbf{X}]$. By the previous lemma, $C(1, \mathbf{X})$ equals $AB \bmod \langle \mathbf{T} \rangle$. To compute C_η, we will use the evaluation / interpolation techniques of Step 1, as the following lemma shows that the polynomials $\mathbf{V}$ split in $R[\mathbf{X}]$. The corresponding algorithm is in Figure 3; it uses a Newton-Hensel lifting algorithm, called Lift, whose last argument indicates the target precision.

Lemma 6. *Let i be in $\{1, \ldots, n\}$. Given $a_{i,0}, \ldots, a_{i,d_i-1}$ and T_i, one can compute $\alpha_{i,0}, \ldots, \alpha_{i,d_i-1}$ in R^{d_i}, with $\alpha_{i,j} - \alpha_{i,j'}$ invertible for $j \neq j'$, and such that*

$$V_i = (X_i - \alpha_{i,0}) \cdots (X_i - \alpha_{i,d_i-1})$$

holds in $R[X_i]$, using $O(\mathsf{M}(r_{\mathbf{T}})\mathsf{C}_{\mathsf{Eval}}(d_i))$ operations in $\mathbb{K}$. The constant in the big-Oh estimate is universal.

Proof. As shown in (6, Section 5), the cost of computing U_i from its roots is $\mathsf{C}_{\mathsf{Eval}}(d_i) + O(\mathsf{M}(d_i))$, which is in $O(\mathsf{C}_{\mathsf{Eval}}(d_i))$ by our assumption on $\mathsf{C}_{\mathsf{Eval}}$; from this, one deduces V_i with $O(d_i)$ operations. The polynomial $U_i = V_i(0, X_i)$ splits into a product of linear terms in $\mathbb{K}[X_i]$, with no repeated root, so V_i splits into $R[X_i]$, by Hensel's lemma. The power series roots $\alpha_{i,j}$ are computed by applying Newton-Hensel lifting to the constants $a_{i,j}$, for $j = 0, \ldots, d_i - 1$. Each lifting step then boils down to evaluate the polynomial V_i and its derivative on the current d_i-uple of approximate solutions and deducing the required correction. Hence, as in (12, Chapter 15), the total cost is $O(\mathsf{M}(r_{\mathbf{T}})\mathsf{C}_{\mathsf{Eval}}(d_i))$ operations; one easily checks that the constant hidden in this big-Oh is universal. □

$\mathsf{LiftRoots}(a_{i,0}, \ldots, a_{i,d_i-1}, T_i)$
1 $U_i \leftarrow (X_i - a_{i,0}) \cdots (X_i - a_{i,d_i-1})$
2 $V_i \leftarrow \eta T_i + (1 - \eta) U_i$
3 return $\mathsf{Lift}(a_{i,0}, \ldots, a_{i,d_i-1}, V_i, \eta^{r_{\mathbf{T}}})$

Fig. 3. Algorithm LiftRoots.

We can finally prove Proposition 4. To compute AB mod $\langle \mathbf{T} \rangle$, we compute $C_\eta = AB$ mod $\langle \mathbf{V} \rangle$ in $R[\mathbf{X}]$, deduce $C \in \mathbb{K}[\eta][\mathbf{X}]$ and evaluate it at 1. By the previous lemma, we can use Proposition 3 over the coefficient ring R to compute C_η. An operation $(+, \times, \div)$ in R has cost $O(\mathsf{M}(r_{\mathbf{T}}))$. Taking into account the costs of Step 1 and Lemma 6, one sees that there exists a constant k' such that the cost is bounded by

$$\mathsf{k}' \ \delta_{\mathbf{T}} \ \mathsf{M}(r_{\mathbf{T}}) \ \sum_{i \leq n} \frac{\mathsf{C}_{\mathsf{Eval}}(d_i) + \mathsf{C}_{\mathsf{Interp}}(d_i)}{d_i}.$$

□

The algorithm is given in Figure 4, under the name MulUnivariate; we use a function called $\mathsf{Choose}(\mathbb{K}, d)$, which returns d pairwise distinct elements from $\mathbb{K}$.

$\mathsf{MulUnivariate}(A, B, \mathbf{T})$
1 for $i = 1, \ldots, n$ do
1.1 $a_{i,0}, \ldots, a_{i,d_i-1} \leftarrow \mathsf{Choose}(\mathbb{K}, d_i)$
1.2 $\alpha_{i,0}, \ldots, \alpha_{i,d_i-1} \leftarrow \mathsf{LiftRoots}(a_{i,0}, \ldots, a_{i,d_i-1}, T_i)$
2 $V \leftarrow [\ (\alpha_{1,\ell_1}, \ldots, \alpha_{n,\ell_n}) \mid 0 \leq \ell_i < d_i\ ]$
3 $C_\eta \leftarrow \mathsf{MulSplit}(A, B, V)$ (computations done mod $\eta^{r_{\mathbf{T}}}$)
4 return $C_\eta(1, \mathbf{X})$ (C_η is seen in $\mathbb{K}[\eta][\mathbf{X}]$)

Fig. 4. Algorithm MulUnivariate.

Step 3: conclusion. To prove Theorem 2, we combine the previous two approaches (the general case and the deformation approach), using the former for large degrees and the latter for smaller ones. Let ε be a positive real, and define $\omega = 2/\varepsilon$. We can assume that the degrees in $\mathbf{T}$ are ordered as $2 \leq d_1 \cdots \leq d_n$, with in particular $\delta_{\mathbf{T}} \geq 2^n$. Define

an index ℓ by the condition that $d_\ell \leq 4^\omega \leq d_{\ell+1}$, taking $d_0 = 0$ and $d_{n+1} = \infty$ for definiteness, and let

$$\mathbf{T}' = (T_1, \ldots, T_\ell) \quad \text{and} \quad \mathbf{T}'' = (T_{\ell+1}, \ldots, T_n).$$

Then the quotient $\mathbb{L}_\mathbf{T}$ equals $R[X_{\ell+1}, \ldots, X_n]/\langle \mathbf{T}'' \rangle$, with $R = \mathbb{K}[X_1, \ldots, X_\ell]/\langle \mathbf{T}' \rangle$. By Equation (2), a product in R can be done in $\mathsf{K}''\, \delta_{\mathbf{T}'}\, r_{\mathbf{T}'} \left(\lg(\delta_{\mathbf{T}'}) \lg(r_{\mathbf{T}'})\right)^3$ operations in $\mathbb{K}$; additions are cheaper, since they can be done in time $\delta_{\mathbf{T}'}$. By Theorem 1, one multiplication in $\mathbb{L}_\mathbf{T}$ can be done in $\mathsf{K}\, 4^{n-\ell} \delta_{\mathbf{T}''} \lg(\delta_{\mathbf{T}''}) \lg\lg(\delta_{\mathbf{T}''})$ operations in R. Hence, taking into account that $\delta_\mathbf{T} = \delta_{\mathbf{T}'} \delta_{\mathbf{T}''}$, the total cost for one operation in $\mathbb{L}_\mathbf{T}$ can be roughly upper-bounded by

$$\mathsf{K}\, \mathsf{K}''\, 4^{n-\ell}\, \delta_\mathbf{T}\, r_{\mathbf{T}'} \left(\lg(\delta_{\mathbf{T}'}) \lg(r_{\mathbf{T}'}) \lg(\delta_{\mathbf{T}''})\right)^3.$$

Now, observe that $r_{\mathbf{T}'}$ is upper-bounded by $d_\ell n \leq 4^\omega \lg(\delta_\mathbf{T})$. This implies that the factor

$$r_{\mathbf{T}'} \left(\lg(\delta_{\mathbf{T}'}) \lg(r_{\mathbf{T}'}) \lg(\delta_{\mathbf{T}''})\right)^3$$

is bounded by $\mathsf{H} \lg^{10}(\delta_\mathbf{T})$, for a constant H depending on ε. Next, $(4^{n-\ell})^\omega = (4^\omega)^{n-\ell}$ is bounded by $d_{\ell+1} \cdots d_n \leq \delta_\mathbf{T}$. Raising to the power $\varepsilon/2$ yields $4^{n-\ell} \leq \delta_\mathbf{T}^{\varepsilon/2}$; thus, the previous estimate admits the upper bounds

$$\mathsf{K}\, \mathsf{K}''\, \mathsf{H}\, \delta_\mathbf{T}^{1+\varepsilon/2} \lg^{10}(\delta_\mathbf{T}) \leq \mathsf{K}\, \mathsf{K}''\, \mathsf{H}\, \mathsf{H}' \delta_\mathbf{T}^{1+\varepsilon},$$

where H' depends on ε.

3. Implementation techniques

The previous algorithms were implemented in C; most efforts were devoted to the generic algorithm of Subsection 2.2. As in (11; 21), the C code was interfaced with AXIOM. In this section, we describe this implementation.

Arithmetic in $\mathbb{F}_p$. Our implementation is devoted to small finite fields $\mathbb{F}_p$, with p a machine word prime of the form $c2^n + 1$, for $c < 2^n$. Multiplications in $\mathbb{F}_p$ are done using Montgomery's REDC routine (23). A straightforward implementation does not bring better performance than the floating point techniques of Shoup (30). We use an improved scheme, adapted to our special primes, presented in the appendix. Compared to a direct implementation of Montgomery's algorithm, it lowers the operation count by 2 double word shifts and 2 single word shifts. This approach performs better on our experimentation platform (Pentium 4) than Shoup's implementation, the gain being of 32%. It is also more efficient and more portable than the one in (11), which explicitly relied on special machine features like SSE registers of late IA-32 architectures.

Arithmetic in $\mathbb{F}_p[X]$. Univariate polynomial arithmetic is crucial: multiplication modulo a triangular set boils down to multivariate polynomial multiplications, which can then be reduced to univariate multiplications through Kronecker's substitution. We use classical and FFT multiplication for univariate polynomials over $\mathbb{F}_p$. We use two FFT multiplication routines: the first one is that from (9); its implementation is essentially the one described in (11), up to a few modifications to improve cache-friendliness. The second one is van der Hoeven's TFT (Truncated Fourier Transform) (13), which is less straightforward but can perform better for transform sizes that are not powers of 2. We tried

several data accessing patterns; the most suitable solution is platform-dependent, since cache size, associativity properties and register sets have huge impact. Going further in that direction will require automatic code tuning techniques, as in (15; 14; 26).

Multivariate arithmetic over $\mathbb{F}_p$. We use a dense representation for multivariate polynomials: important applications of modular multiplication (GCD computations, Hensel lifting for triangular sets) tend to produce dense polynomials. We use multidimensional arrays (encoded as a contiguous memory block of machine integers) to represent our polynomials, where the size in each dimension is bounded by the corresponding degree $\deg(T_i, X_i)$, or twice that much for intermediate products. Multivariate arithmetic is done using either Kronecker's substitution as in (11) or standard multidimensional FFT. While the two approaches share similarities, they do not access data in the same manner. In our experiments, multidimensional FFT performed better by 10-15% for bivariate cases, but was slower for larger number of variables with small FFT size in each dimension.

Triangular sets over $\mathbb{F}_p$. Triangular sets are represented in C by an array of multivariate polynomials. For the algorithm of Subsection 2.3, we only implemented the case where all degrees are 2; this mostly boils down to evaluation and interpolation on n-dimensional grids of size 2^n, over a power series coefficient ring.

More work was devoted to the algorithm of Subsection 2.2. Two strategies for modular multiplication were implemented, a plain one and that of Subsection 2.2. Both first perform a multivariate multiplication then do a multivariate reduction; the plain reduction method performs a recursive Euclidean division, while the faster one implements both algorithms Rem and MulTrunc of Subsection 2.2. Remark in particular that even the plain approach is not the entirely naive, as it uses fast multivariate multiplication for the initial multiplication. Both approaches are recursive, which makes it possible to interleave them. At each level $i = n, \ldots, 1$, a cut-off point decides whether to use the plain or fast algorithm for multiplication modulo $\langle T_1, \ldots, T_i \rangle$. These cut-offs are experimentally determined: as showed in Section 4, they are surprisingly low for $i > 1$.

The fast algorithm uses precomputations (of the power series inverses of the reciprocals of the polynomials T_i). In practice, it is of course better to store and reuse these elements: in situations such as GCD computation or Hensel lifting, we expect to do several multiplications modulo the same triangular set. We could push further these precomputations, by storing Fourier transforms; this is not done yet.

GCD's. One of the first applications of fast modular multiplication is GCD computation modulo a triangular set, which itself is central to higher-level algorithms for solving systems of equations. Hence, we implemented a preliminary version of such GCD computations using a plain recursive version of Euclid's algorithm. This implementation has not been thoroughly optimized. In particular, we have not incorporated any half-GCD technique, except for *univariate* GCD's; this univariate half-GCD is far from optimal.

The AXIOM level. Integrating our fast arithmetic into AXIOM is straightforward, after dealing with the following two problems. First, AXIOM is a Lisp-based system, whereas our package is implemented in C. Second, in AXIOM, dense multivariate polynomials are represented by recursive trees, but in our C package, they are encoded as multidimensional arrays. Both problems are solved by modifying the GCL kernel. For the first issue, we integrate our C package into the GCL kernel, so that our functions from can be used by AXIOM at run-time. For the second problem, we realized a tree / array polynomial data converter. This converter is also linked to GCL kernel; the conversations, happening at run-time, have negligible cost.

4. Experimental results

The main part of this section describes experimental results attached to our main algorithm of Subsection 2.2; we discuss the algorithm of Subsection 2.3 in the last paragraphs. For the entire set of benchmarks, we use random dense polynomials. Our experiments were done on a 2.80 GHz Pentium 4 PC, with 1GB memory and 1024 KB cache.

4.1. Comparing different strategies

We start by experiments comparing different strategies for computing products modulo triangular sets in $n = 1, 2, 3$ variables, using our general algorithm.

Strategies. Let $\mathbb{L}_0 = \mathbb{F}_p$ be a small prime field and let $\mathbb{L}_n$ be $\mathbb{L}_0[X_1, \ldots, X_n]/\langle \mathbf{T} \rangle$, with $\mathbf{T}$ a n-variate triangular set of multi-degree $(d_1, \ldots, d_n)$. To compute a product $C = AB \in \mathbb{L}_n$, we first expand $P = AB \in \mathbb{L}_0[\mathbf{X}]$, then reduce it modulo $\mathbf{T}$. The product P is always computed by the same method; we use three strategies for computing C.

- PLAIN. We use univariate Euclidean division; computations are done recursively in $\mathbb{L}_{i-1}[X_i]$ for $i = n, \ldots, 1$.
- FAST, USING PRECOMPUTATIONS. We apply the algorithm $\mathsf{Rem}(C, \mathbf{T}, \mathbf{S})$ of Figure 1, assuming that the inverses $\mathbf{S}$ have been precomputed.
- FAST, WITHOUT PRECOMPUTATIONS. We apply the algorithm $\mathsf{Rem}(C, \mathbf{T}, \mathbf{S})$ of Figure 1, but recompute the required inverses on the fly.

Our ultimate goal is to obtain a highly efficient implementation of the multiplication in $\mathbb{L}_n$. To do so, we want to compare our strategies in $\mathbb{L}_1$, $\mathbb{L}_2$, ..., $\mathbb{L}_n$. In this report we give details for $n \leq 3$ and leave for future work the case of $n > 3$, as the driving idea is to tune our implementation in $\mathbb{L}_i$ before investigating that of $\mathbb{L}_{i+1}$. This approach leads to determine cut-offs between our different strategies. The alternative is between PLAIN and FAST strategies, depending on the assumption regarding precomputations. For applications discussed before (quasi-inverses, polynomial GCDs modulo a triangular set), using precomputations is realistic.

Univariate multiplication. Figure 5 compares our implementation of the Truncated Fourier Transform (TFT) multiplication to the classical Fast Fourier Transform (FFT). Because the algorithm is more complex, especially the interpolation phase, the TFT approach does not outperform the classical FFT multiplication in all cases.

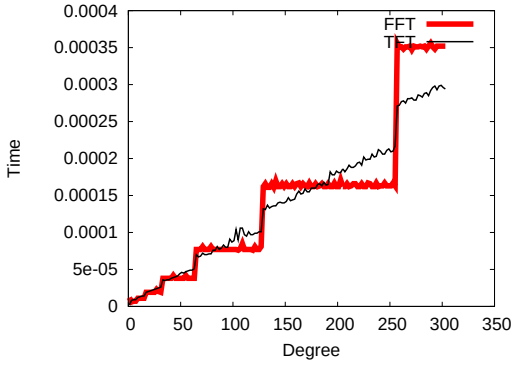

Fig. 5. TFT vs. FFT.

Univariate triangular sets. Finding the cut-offs between our strategies is straightforward. Figure 6 shows the result using classical FFT multiplication; the cut-off point is about 150. If precomputations are not assumed, then this cut-off doubles. Using Truncated Fourier Transform, one obtains roughly similar results.

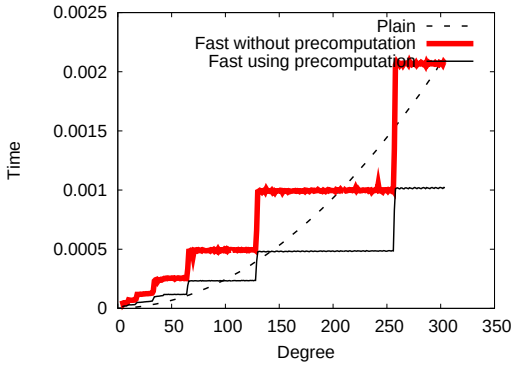

Fig. 6. Multiplication in $\mathbb{L}_1$, all strategies, using FFT multiplication.

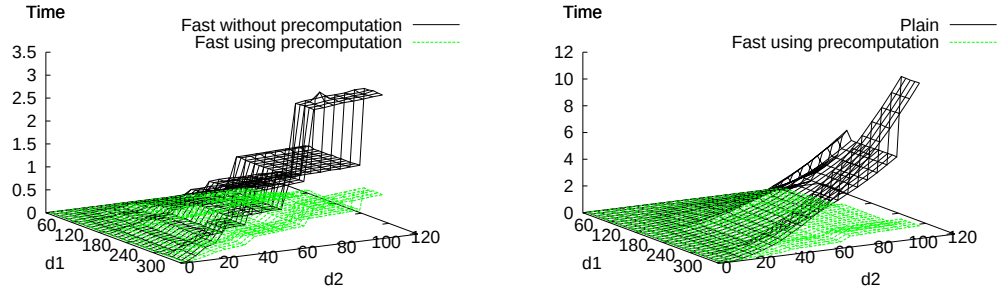

Fig. 7. Multiplication in $\mathbb{L}_2$, FAST without precomputations vs. FAST using precomputations (left) and PLAIN vs. FAST using precomputations (right).

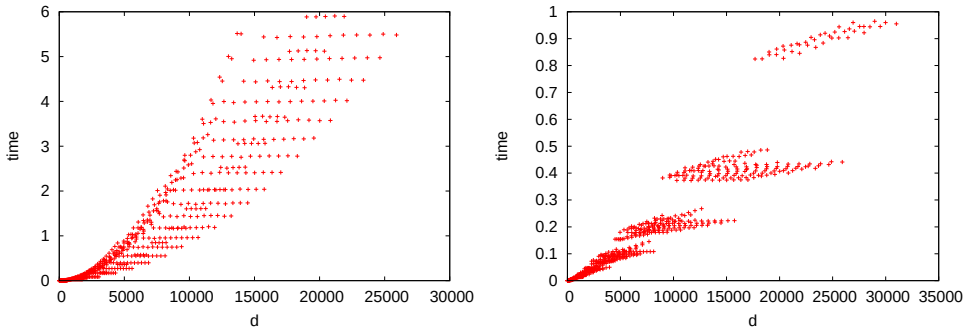

Fig. 8. Multiplication in $\mathbb{L}_2$, time vs. $d = d_1 d_2$, PLAIN (left) and FAST using precomputations (right).

Bivariate triangular sets. For $n = 2$, we let in Figure 7 d_1 and d_2 vary in the ranges $4, \ldots, 304$ and $2, \ldots, 102$. This allows us to determine a cut-off for d_2 as a function of d_1. Surprisingly, this cut-off is essentially independent of d_1 and can be chosen equal to 5. We discuss this point below. To continue our benchmarks in $\mathbb{L}_3$, we would like the product $d_1 d_2$ to play the role in $\mathbb{L}_3$ that d_1 did in $\mathbb{L}_2$, so as to determine the cut-off for d_3 as a function of $d_1 d_2$. This leads to the question: for a *fixed* product $d_1 d_2$, does the running time of the multiplication in $\mathbb{L}_2$ stay constant when (d_1, d_2) varies in the region $4 \leq d_1 \leq 304$ and $2 \leq d_2 \leq 102$? Figure 8 gives timings obtained for this sample set; it shows that the time varies mostly for the PLAIN strategy (the levels in the FAST case are due to our FFT multiplication). These results guided our experiments in $\mathbb{L}_3$.

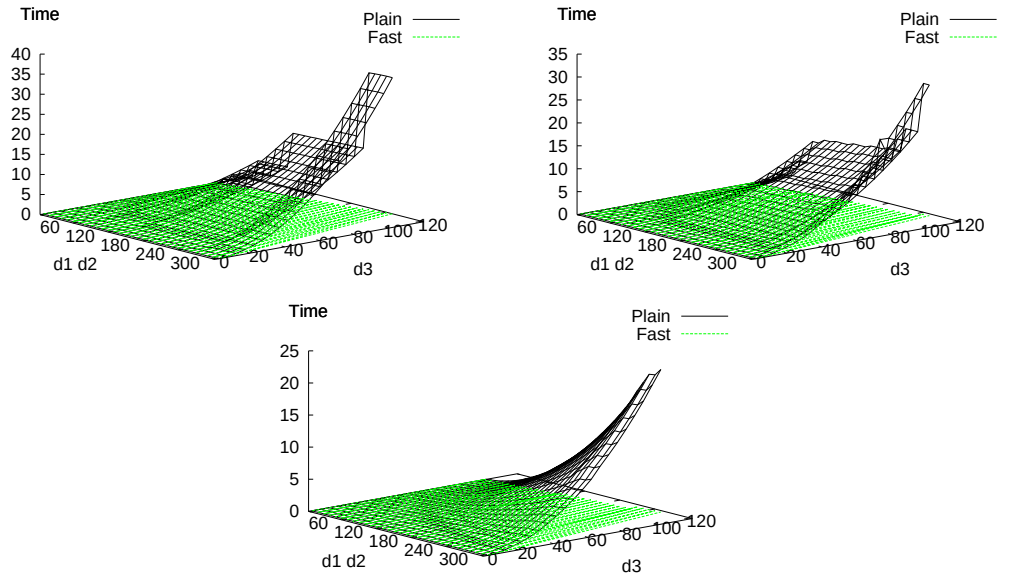

Fig. 9. Multiplication in $\mathbb{L}_3$, PLAIN vs. FAST, patterns 1–3 from top left to bottom.

Trivariate triangular sets. For our experiments with $\mathbb{L}_3$, we consider three patterns for (d_1, d_2). Pattern 1 has $d_1 = 2$, Pattern 2 has $d_1 = d_2$ and Pattern 3 has $d_2 = 2$. Then, we let $d_1 d_2$ vary from 4 to 304 and d_3 from 2 to 102. For simplicity, we also report only the comparison between the strategies PLAIN and FAST USING PRECOMPUTATIONS. The timings are in Figure 9; they show an impressive speed-up for the FAST strategy. We also observe that the cut-off between the two strategies can be set to 3 for each of the patterns. Experiments as in Figure 8 gives similar conclusion: the timing depends not only on $d_1 d_2$ and d_3 but also on the ratios between these degrees.

Discussion of the cut-offs. To understand the low cut-off points we observe, we have a closer look at the costs of several strategies for multiplication in $\mathbb{L}_2$. For a ring R, classical polynomial multiplication in $R[X]$ in degree less than d uses about (d^2, d^2) operations $(\times, +)$ respectively (we omit linear terms in d). Euclidean division of a polynomial of degree $2d-2$ by a monic polynomial T of degree d has essentially the same cost. Hence,

classical modular multiplication uses about $(2d^2, 2d^2)$ operations $(\times, +)$ in R. Additions modulo $\langle \mathbf{T} \rangle$ take d operations.

Thus, a pure recursive approach for multiplication in $\mathbb{L}_2$ uses about $(4d_1^2d_2^2, 4d_1^2d_2^2)$ operations $(\times, +)$ in $\mathbb{K}$. Our PLAIN approach is less naive. We first perform a bivariate product in degrees (d_1, d_2). Then, we reduce all coefficients modulo $\langle T_1 \rangle$ and perform Euclidean division in $\mathbb{L}_1[X_2]$, for a cost of about $(2d_1^2d_2^2, 2d_1^2d_2^2)$ operations. Hence, we can already make some advantage of fast FFT-based multiplication, since we traded $2d_1^2d_2^2$ base ring multiplications and as many additions for a bivariate product.

Using precomputations, the FAST approach performs 3 bivariate products in degrees about (d_1, d_2) and about $4d_2$ reductions modulo $\langle T_1 \rangle$. Even for moderate (d_1, d_2) such as in the range 20–30, bivariate products can already be done efficiently by FFT multiplication, for a cost much inferior to $d_1^2d_2^2$. Then, *even if reductions modulo* $\langle T_1 \rangle$ *are done by the* PLAIN *algorithm*, our approach performs better: the total cost of these reductions will be about $(4d_1^2d_2, 4d_1^2d_2)$, so we save a factor $\simeq d_2/2$ on them. This explains why we observe very low cut-offs in favor of the FAST algorithm.

4.2. Comparing implementations

Comparison with Magma. To evaluate the quality of our implementation of modular multiplication, we compared it with MAGMA v. 2-11 (5), which has set a standard of efficient implementation of low-level algorithms. We compared multiplication in $\mathbb{L}_3$ for the previous three patterns, in the same degree ranges. Figure 10 gives the timings for Pattern 3. The MAGMA code uses iterated `quo` constructs over `UnivariatePolynomial`'s, which was the most efficient configuration we found. For our code, we use the strategy FAST USING PRECOMPUTATIONS. On this example, our code outperforms MAGMA by factors up to 7.4; other patterns yield similar behavior.

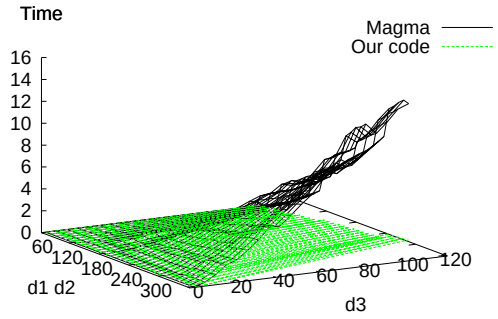

Fig. 10. Multiplication in $\mathbb{L}_3$, pattern 3, Magma vs. our code.

Comparison with Maple. Our future goal is to obtain high-performance implementations of higher-level algorithms in higher-level languages, replacing built-in arithmetic by our C implementation. Doing it within MAPLE is not straightforward; our MAPLE experiments stayed at the level of GCD and inversions in $\mathbb{L}_3$, for which we compared our code with MAPLE's `recden` library. We used the same degree patterns as before, but we were led to reduce the degree ranges to $4 \leq d_1d_2 \leq 204$ and $2 \leq d_3 \leq 20$. Our code uses the strategy FAST USING PRECOMPUTATIONS. The MAPLE `recden` library implements multivariate dense recursive polynomials and can be called from the MAPLE interpreter

via the `Algebraic` wrapper library. Our MAPLE timings, however, do not include the necessary time for converting MAPLE objects into the `recden` format: we just measured the time spent by the function `invpoly` of `recden`. Figure 11 gives the timings for Pattern 3 (the other results are similar). There is a significant performance gap (our timing surface is very close the bottom). When using our PLAIN strategy, our code remains faster, but the ratio diminishes by a factor of about 4 for the largest configurations.

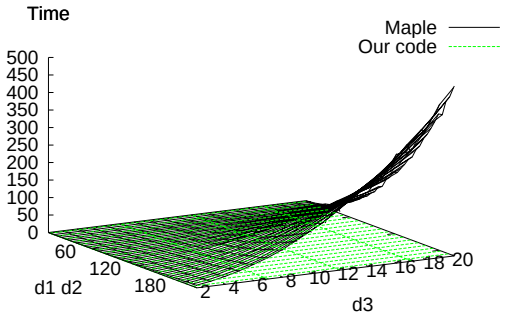

Fig. 11. Inverse in $\mathbb{L}_3$, pattern 1, Maple vs. our code.

Comparison with AXIOM. Using our arithmetic in AXIOM is made easy by the C/GCL structure. In (21), the modular algorithm by van Hoeij and Monagan (22) was used as a driving example to show strategies for such multiple-level language implementations. This algorithm computes GCD's of univariate polynomials with coefficients in a number field by modular techniques. The coefficient field is described by a tower of simple algebraic extensions of $\mathbb{Q}$; we are thus led to compute GCD's modulo triangular sets over $\mathbb{F}_p$, for several primes p. We implemented the top-level algorithm in AXIOM. Then, two strategies were used: one relying on the built-in AXIOM modular arithmetic, and the other on our C code; the only difference between the two strategies at the top-level resides in which GCD function to call. The results are given in Figure 12. We use polynomials A, B in $\mathbb{Q}[X_1, X_2, X_3]/\langle T_1, T_2, T_3\rangle[X_4]$, with coefficients of absolute value bounded by 2. As shown in Figure 12 the gap is dramatic.

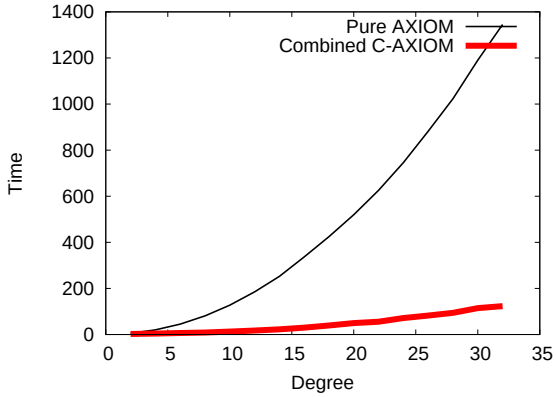

Fig. 12. GCD computations $\mathbb{L}_3[X_4]$, pure AXIOM code vs. combined C-AXIOM code.

4.3. The deformation-based algorithm

We conclude with the implementation of the algorithm of Subsection 2.3, devoted to triangular sets made of univariate polynomials only. We focus on the most favorable case for this algorithm, when all degrees d_i are 2: in this case, in n variables, the cost reported in Proposition 4 becomes $O(2^n n\mathsf{M}(n))$. This extreme situation is actually potentially useful, see for instance an application to the addition of algebraic numbers in characteristic 2 in (29). For most practical purposes, n should be in the range of about $1, \ldots, 20$; for such sizes, multiplication in degree n will rely on naive or at best Karatsuba multiplication; hence, a reasonable practical estimate for the previous bound is $O(2^n n^3)$, which we can rewrite as $O(\delta_{\mathbf{T}} \log(\delta_{\mathbf{T}})^3)$. We compare in Figure 13 the behavior of this algorithm to the general one. As expected, the former behaves better: the general algorithm starts by multiplying the two input polynomials, before reducing them. The number of monomials in the product before reduction is $3^n = \delta_{\mathbf{T}}^{\log_2(3)}$. Hence, for this family of problems, the general algorithm has a non-linear complexity.

variables	$\delta_{\mathbf{T}}$	general (Subsection 2.2)	specialized (Subsection 2.3)
3	8	0.000188	0.000043
4	16	0.001288	0.000126
5	32	0.007888	0.000337
6	64	0.045804	0.000983
7	128	0.254427	0.002720
8	256	1.434127	0.008141
9	512	7.682161	0.019928
10	1024	40.519331	0.052337
11	2048	204.719505	0.131778

Fig. 13. General vs. specialized algorithm.

References

[1] P. Aubry, D. Lazard, and M. Moreno Maza. On the theories of triangular sets. *J. Symbolic Comput.*, 28(1,2):45–124, 1999.

[2] D. Bini. Relations between exact and approximate bilinear algorithms. Applications. *Calcolo*, 17(1):87–97, 1980.

[3] D. Bini, M. Capovani, F. Romani, and G. Lotti. $O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication. *Inf. Proc. Lett.*, 8(5):234–235, 1979.

[4] D. Bini, G. Lotti, and F. Romani. Approximate solutions for the bilinear form computational problem. *SIAM J. Comput.*, 9(4):692–697, 1980.

[5] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997.

[6] A. Bostan and É. Schost. On the complexities of multipoint evaluation and interpolation. *Theor. Comput. Sci.*, 329(1-3): 223–235, 2004.

[7] D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28(7):693–701, 1991.

[8] S. Cook. *On the minimum computation time of functions.* PhD thesis, Harvard University, 1966.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2002.
[10] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In *ISSAC'05*, pages 108–115. ACM, 2005.
[11] A. Filatei, X. Li, M. Moreno Maza, and É. Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *ISSAC'06*, pages 93–100. ACM, 2006.
[12] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.
[13] J. van der Hoeven. The Truncated Fourier Transform and applications. In *ISSAC'04*, pages 290–296. ACM, 2004.
[14] J. R. Johnson, W. Krandick, K. Lynch, K. G. Richardson, and A. D. Ruslanov. High-performance implementations of the Descartes method. In *ISSAC'06*, pages 154–161. ACM, 2006.
[15] J. R. Johnson, W. Krandick, and A. D. Ruslanov. Architecture-aware classical Taylor shift by 1. In *ISSAC'05*, pages 200–207. ACM, 2005.
[16] H. T. Kung. On computing reciprocals of power series. *Numerische Mathematik*, 22:341–348, 1974.
[17] L. Langemyr. Algorithms for a multiple algebraic extension. In *Effective methods in algebraic geometry*, volume 94 of *Progr. Math.*, pages 235–248. Birkhäuser, 1991.
[18] D. Lazard. Solving zero-dimensional algebraic systems. *J. Symbolic Comput.*, 13(2):147–160, 1992.
[19] F. Lemaire, M. Moreno Maza, and Y. Xie. The `RegularChains` library. In Ilias S. Kotsireas, editor, Maple Conference 2005, pages 355–368, 2005.
[20] X. Li, M. Moreno Maza, and É. Schost. Fast arithmetic for triangular sets: from theory to practice. In *ISSAC'07*, pages 269–276. ACM, 2007.
[21] X. Li and M. Moreno Maza. Efficient implementation of polynomial arithmetic in a multiple-level programming environment. In *ICMS'06*, pages 12–23. Springer, 2006.
[22] M. van Hoeij and M. Monagan. A modular GCD algorithm over number fields presented with multiple extensions. In *ISSAC'02*, pages 109–116. ACM, 2002.
[23] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
[24] M. Moreno Maza. On triangular decompositions of algebraic varieties. Technical Report TR 4/99, NAG Ltd, Oxford, UK, 1999. `http://www.csd.uwo.ca/~moreno/`.
[25] V. Y. Pan. Simple multivariate polynomial multiplication. *J. Symbolic Comput.*, 18(3):183–186, 1994.
[26] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc' IEEE*, 93(2):232–275, 2005.
[27] A. Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica*, 7:395–398, 1977.
[28] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.
[29] É. Schost. Multivariate power series multiplication. In *ISSAC'05*, pages 293–300. ACM, 2005.
[30] V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symb. Comp.*, 20(4):363–397, 1995.
[31] M. Sieveking. An algorithm for division of powerseries. *Computing*, 10:153–156, 1972.

Appendix: multiplication modulo special primes

Let p be a prime of the form $p = c2^n + 1$, for $c < 2^n$ (in our code, n ranges from 20 to 23 and c is less than 1000). Let $\ell = \lceil \log_2(p) \rceil$ and let $R = 2^\ell$. Given a and ω, both reduced modulo p, Montgomery's REDC algorithm computes $a\omega/R \bmod p$. We present our tailor-made version here. Precomputations will be authorized for the argument ω (this is not a limitation for our main application, FFT polynomial multiplication, where ω is a fixed primitive root of unity). We compute

(1) $M_1 = a\omega$
(2) $(q_1, r_1) = (M_1 \text{ div } R, M_1 \bmod R)$
(3) $M_2 = r_1 c 2^n$
(4) $(q_2, r_2) = (M_2 \text{ div } R, M_2 \bmod R)$
(5) $M_3 = r_2 c 2^n$
(6) $q_3 = M_3 \text{ div } R$
(7) $A = q_1 - q_2 + q_3$.

Proposition 7. *Suppose that $c < 2^n$ and $\omega \neq -1 \bmod p$. Then A satisfies $A \equiv a\omega/R \bmod p$ and $-(p-1) < A < 2(p-1)$.*

Proof. By construction, we have the equalities $Rq_1 = M_1 - r_1$ and $Rq_2 = M_2 - r_2$. Remark next that 2^n divides M_2, and thus r_2 (since R is a power of two larger than 2^n). It follows that 2^{2n} divides M_3. Since we have $c < 2^n$, p is at most 2^{2n}, so R is at most 2^{2n} as well. Hence, R divides M_3, so that $Rq_3 = M_3$. Putting this together yields

$$RA = M_1 - r_1 - M_2 + r_2 + M_3.$$

Recall that $M_2 = r_1 c 2^n$, so that $M_2 = -r_1 \bmod p$. Similarly, $M_3 = r_2 c 2^n$, so $M_3 = -r_2 \bmod p$. Hence, $RA = M_1 \bmod p$, which means that $A = a\omega/R \bmod p$, as claimed. As to the bounds on A, we start by remarking that $M_1 < (p-1)^2$, since $\omega \neq -1 \bmod p$. We thus have $q_1 < p - 1$. Next, since $r_1 < R$, we deduce that $M_2 < c2^n R$ which implies that $q_2 < c2^n = p - 1$. Similarly, we obtain that $q_3 < p - 1$, which implies the requested inequalities. □

Let us now describe our implementation on 32-bit x86 processors. We use an assembly macro `MulHiLo`(a, b) from the GMP library; this macro computes the product d of two word-length integers a and b and puts the high part of the result ($d \text{ div } 2^{32}$) in the register `AX` and the lower part ($d \bmod 2^{32}$) in the register `DX`, avoiding shifts.

In our case, R does not equal 2^{32}. However, since we allow precomputations on ω, we will actually store and use $\omega' = 2^{32-\ell}\omega$ instead of ω; hence, `MulHiLo`(a, ω') directly gives us q_1 and $r_1' = 2^{32-\ell} r_1$. Similarly, we do not compute the product $r_1 c 2^n$; instead, we use `MulHiLo`(r_1', c'), where c' is the precomputed constant $c2^n$, to get q_2 and $r_2' = 2^{32-\ell} r_2$.

To compute q_3, it turned out to be better to do as follows. We write q_3 as $r_2 c / 2^{\ell - n}$. Now, recall from the proof of the previous proposition that 2^n divides r_2. Under the assumption that $c < 2^n$, we saw in the proof that $\ell \leq 2n$, so that $2^{\ell-n}$ divides r_2. Hence, we obtain q_3 by right-shifting r_2 by $\ell - n$ places, or, equivalently, r_2' by $32 - n$ places, and multiplying the result by c. Eventually, we need to bring the result A between 0 and $p - 1$. As in NTL (30), we avoid `if` statements: using the sign bit of A as a mask, one can add p to A in the case $A < 0$; by subtracting p and correcting once more, we obtain the correct remainder.