Fast Arithmetic for Triangular Sets: From Theory to Practice

Xin Li, Marc Moreno Maza, Éric Schost
ORCCA and Computer Science Department
University of Western Ontario
London, ON, Canada
{xli96,moreno,schost}@scl.csd.uwo.ca

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 even reach quasi-linear complexity.

The main outcome we have in mind for this study is the acceleration of a family of higher-level algorithms, by interfacing our lower-level implementation with langagues 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 [8].

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 [18].

Triangular sets. In this article, we adopt the following convention: a *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 ring. We impose that for all i, T_i is in $R[X_1, \ldots, X_i]$, is monic in X_i and reduced with respect to $T_1, \ldots, T_{i-1}$.

The natural approach to arithmetic modulo triangular sets is recursive: to work in the residue class ring $\mathbb{L} = R[X_1, \ldots, X_n]/\langle T_1, \ldots, T_n\rangle$, we see it as $\mathbb{L}_-[X_n]/\langle T_n\rangle$, where $\mathbb{L}_-$ 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 languages supporting generic programming. 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 $\mathbb{L}$, compute their product; here, one assumes that the input and output are reduced with respect to $\mathbf{T}$.

In one variable, the usual approach consists in multiplying A and B and reducing them by Euclidean division. Using classical arithmetic, the cost is approximately $2d_1^2$ 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 ([5], after [24, 23]) 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 [6, 27, 13].

In n variables, the measure of complexity of the problem is $\delta = \deg(T_1, X_1) \cdots \deg(T_n, X_n)$, since representing a polynomial reduced modulo $\mathbf{T}$ requires storing δ elements. Then, applying the previous results recursively leads to bounds of order $2^n\delta^2$ for the standard approach, and $(3\mathsf{k})^n\delta$ for the fast one, neglecting logarithmic factors and lower-order terms.

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 that a mixed dense / recursive approach reduces the previous cost to an order of $4^n\delta$, neglecting again all lower order terms and logarithmic factors. Building upon previous work [9], 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. This can be seen by doing a precise complexity analysis of our algorithm, and clearly appears in our experimental results.

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 [25]. We extend this result to the case of arbitrary $T_i \in \mathbb{K}[X_i]$; we prove that for $\mathbb{K}$ of cardinality large enough and for $\varepsilon > 0$, there exists 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 $\mathsf{K}_\varepsilon \delta^{1+\varepsilon}$ operations, with δ as before.

Following [2, 3, 1, 25], the algorithm uses deformation techniques, and is not expected to be very practical. However, it shows that for a substantial family of examples, 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 [9, 17]) 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 [20], which is implemented in AXIOM and MAPLE [15].

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 algorithms for multiplication, for respectively general triangular sets, and triangular sets containing univariate polynomials. We next describe our implementation of the former algorithm in Section 3; experimental results, including comparison with other systems, are given in Section 4.

Acknowledgements. We wish to thank R. Rasheed for his help with obtaining the MAPLE timings.

2. ALGORITHMS AND COMPLEXITY

In this section, we give the details of our algorithms for multiplication modulo a triangular set. We start by some notation.

Notation. Triangular sets will be written $\mathbf{T} = (T_1, \ldots, T_n)$. The *multi-degree* of $\mathbf{T}$ is the sequence $(d_i = \deg(T_i, X_i))_{1 \le i \le n}$. We will write $\delta_\mathbf{T} = d_1 \cdots d_n$ and, in Subsection 2.2, we 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 $R[\mathbf{X}]/\langle \mathbf{T} \rangle$, for R a ring. Let $M_\mathbf{T}$ be the set of monomials

$$M_\mathbf{T} = \left\{ X_1^{e_1} \cdots X_n^{e_n} \mid 0 \le e_i < d_i \text{ for all } i \right\};$$

then, the free R-submodule $\mathsf{Span}(M_\mathbf{T})$ of $R[\mathbf{X}]$ generated by $M_\mathbf{T}$ is isomorphic to $\mathbb{L}_\mathbf{T}$, so that in our algorithms, elements of $\mathbb{L}_\mathbf{T}$ are represented on the monomial basis $M_\mathbf{T}$.

Without loss of generality, we will assume in all that follows 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 T_i can be discarded.

Standard and fast modular multiplication. As said before, using standard algorithms leads to 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 δ, due to the exponential overhead in n. However, since all degrees $\deg(T_i, X_i)$ are at least two, any bound of the form $\mathsf{K}^n \delta^\ell$ is actually polynomial in δ, since it is upper-bounded by $\delta^{\log_2(\mathsf{K})+\ell}$.

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, with a small constant K. 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 [10, Chapter 8]. Using the algorithm of [5], one can take $\mathsf{M}(d) \in O(d \log(d) \log\log(d))$. Precisely, we will denote by k a constant such that $\mathsf{M}(d) \le \mathsf{k}\, d \lg(d) \lg\lg(d)$ 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 [6, 27, 13]. 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 is the reciprocal polynomial of T_1. This is done using Newton iteration, and can be performed as a precomputation, for a cost of $3\mathsf{M}(d_1) + O(d_1)$. Once S_1 is known, it enables one to recover first the reciprocal of Q, then the remainder C, using two polynomial products. Taking into account the cost of computing AB, these operations have cost $3\mathsf{M}(d_1) + d_1$.

Applying this result recursively leads to a rough upper bound of $\Pi_{i \le n}(3\mathsf{M}(d_i) + d_i)$ for a product in $\mathbb{L}_\mathbf{T}$, without taking into account the similar cost of precomputation (see [14] for similar considerations); this gives a total estimate of roughly $(3\mathsf{k} + 1)^n \delta_\mathbf{T}$, neglecting logarithmic factors. One can reduce this $(3\mathsf{k} + 1)^n$ exponential overhead: since additions and constant multiplications in $\mathbb{L}_\mathbf{T}$ can be done in linear time, it is mainly the *bilinear* cost of univariate multiplication which governs the cost of the above recursive process. 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 $(3 \times 2)^n \delta_\mathbf{T} = 6^n \delta_\mathbf{T}$.

Main results. By studying more precisely the multivariate multiplication process, we prove in Theorem 1 that one can compute products in $\mathbb{L}_\mathbf{T}$ in time at most $\mathsf{K}\, 4^n \delta_\mathbf{T} \lg^3(\delta_\mathbf{T})$, for a universal constant K. This is a synthetic but rough upper bound; we give more precise estimates within the proof, and in Section 4 in the case $n = 2$.

Obtaining results linear in $\delta_\mathbf{T}$ that would not involve an exponential factor in n is a major open problem. When the base ring R 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.

Complexity. 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. Hence, with a view to improve readability, we keep some estimates sub-optimal regarding logarithmic factors, relying on rough upper bounds like $\lg\lg(d) \le \lg(d)$.

Recall that k is such that the univariate multiplication cost is bounded by $\mathsf{k}\, d \lg(d) \lg\lg(d)$ for all d. Up to increasing k, we can also assume that over any ring R, evaluation and interpolation of a polynomial of degree less than d on d points $a_0, \dots, a_{d-1}$ can be done in at most $\mathsf{k}\, d \lg^2(d) \lg\lg(d)$ operations, assuming $a_i - a_j$ is a unit for $i \ne j$, see [10, Chapter 10]. Following our policy above, we use the upper bound $\mathsf{k}\, d \lg^3(d)$.

We let $\mathsf{MM}(d_1, \dots, d_n)$ be such that over any ring R, polynomials in $R[X_1, \dots, X_n]$ of degree in X_i less than d_i for all i can be multiplied in $\mathsf{MM}(d_1, \dots, d_n)$ operations. One can take $\mathsf{MM}(d_1, \dots, d_n) \le \mathsf{M}((2d_1 - 1)\cdots(2d_n - 1))$ using Kronecker's substitution. Letting $\delta = d_1 \cdots d_n$, and up to increasing k, this is seen to be at least δ and at most $\mathsf{k}\, 2^n \delta \lg^3(\delta)$, assuming $d_i \ge 2$ for all i. Pan [21] 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 a noticeable improvement, due to our rough over-simplifications.

2.1 The main algorithm

We describe here our main algorithm for modular multiplication. While relying on the Cook-Sieveking-Kung idea, this algorithm differs from a direct recursive implementation.

Theorem 1. *There exist 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* $\mathsf{K}\, 4^n \delta_{\mathbf{T}} \lg^3(\delta_{\mathbf{T}})$ *operations* $(+, \times)$ *in* R.

Proof. Let $\mathbf{T} = (T_1, \dots, T_n)$ have multi-degree $(d_1, \dots, d_n)$ in $R[\mathbf{X}] = R[X_1, \dots, X_n]$. We write $\mathbf{T}_- = (T_1, \dots, T_{n-1})$, so that $\mathbb{L}_{\mathbf{T}_-} = R[X_1, \dots, X_{n-1}]/\langle \mathbf{T}_- \rangle$.

For $i \le n$, we denote by U_i the reciprocal polynomial of T_i and by S_i the inverse of U_i modulo $\langle T_1, \dots, T_{i-1}, X_i^{d_i - 1}\rangle$. The sequences $(S_1, \dots, S_n)$ and $(S_1, \dots, S_{n-1})$ are denoted by $\mathbf{S}$ and $\mathbf{S}_-$.

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

Our second subroutine is called $\mathsf{MulTrunc}(A, B, \mathbf{T}, \mathbf{S}, d_{n+1})$, 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 $i \le n + 1$. If $n = 0$, $\mathbf{T}$ is empty, so this function return $AB \bmod X_1^{d_1}$.

To compute $\mathsf{Rem}(A, \mathbf{T})$, we use the Cook-Sieveking-Kung approach in $\mathbb{L}_{\mathbf{T}_-}[X_n]$. We first reduce all coefficients of A modulo $\mathbf{T}_-$ and perform two truncated products in $\mathbb{L}_{\mathbf{T}_-}[X_n]$ using the function $\mathsf{MulTrunc}$. The operation $\mathsf{MulTrunc}$ is performed by multiplying A and B as polynomials, before truncating in X_{n+1} and reducing all coefficients modulo $\mathbf{T}$, using the function Rem. In Figure 1, $\mathsf{Coeff}(D, X_i, e)$ denotes the coefficient of X_i^e in a polynomial D and $\mathsf{Rev}(D, X_i, e)$ denotes the reciprocal polynomial $X_i^e D[X_i \leftarrow 1/X_i]$, assuming $\deg(D, X_i) \le e$.

```
Rem(A, T, 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 $A' \bmod X_n^{d_n} - \mathsf{MulTrunc}(Q, T_n, \mathbf{T}_-, \mathbf{S}_-, d_n)$

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

Figure 1: Algorithms Rem and $\mathsf{MulTrunc}$.

Assuming for a start that all inverses $\mathbf{S}$ have been precomputed, we write $\mathsf{C}_{\mathsf{Rem}}(d_1, \dots, d_n)$ for an upper bound on the cost of $\mathsf{Rem}(A, \mathbf{T}, \mathbf{S})$ and $\mathsf{C}_{\mathsf{MulTrunc}}(d_1, \dots, d_{n+1})$ 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

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

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

$$\begin{aligned}
\mathsf{C}_{\mathsf{Rem}}(d_1, \dots, d_n) \le 4\mathsf{C}_{\mathsf{Rem}}(d_1, \dots, d_{n-1}) d_n \\
+\ 2\mathsf{MM}(d_1, \dots, d_n) + d_1 \cdots d_n,
\end{aligned}$$

for $n \ge 1$. Write $\mathsf{MM}'(d_1, \dots, d_n) = 2\mathsf{MM}(d_1, \dots, d_n) + d_1 \cdots d_n$. Then, this recurrence yields the upper bound

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

In view of the bound on MM given above and taking e.g. $\mathsf{K} = 3\mathsf{k}$, we get after a few simplifications

$$\mathsf{C}_{\mathsf{Rem}}(d_1, \dots, d_n) \le \mathsf{K}\, 4^n \delta_{\mathbf{T}} \lg^3(\delta_{\mathbf{T}}).$$

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 $\mathsf{K} + \mathsf{k}$. Observe for future use that multiplication in degree $d'_n \le d_n$ in $\mathbb{L}_{\mathbf{T}_-}[X_n]$ can be performed in $\mathsf{K}\, 4^n \delta' \lg^3(\delta_{\mathbf{T}})$, with $\delta' = d_1 \cdots d_{n-1} d'_n$.

To conclude, we estimate the cost of precomputing $\mathbf{S}$. Supposing that $S_1, \dots, S_{n-1}$ are known, we detail the cost of computing S_n. Let $\ell = \lceil \log_2(d_n - 1) \rceil$. Using Newton iteration in $\mathbb{L}_{\mathbf{T}_-}[X_n]$, we obtain S_n by performing 2 multiplications $\mathbb{L}_{\mathbf{T}_-}[X_n]$ in degrees less than m and $m/2$ negations, for $m = 2, 4, \dots, 2^{\ell-1}$. By the remark above, the cost is at most $\mathsf{t}(n) = 3\mathsf{K}\, 4^n \delta_{\mathbf{T}} \lg^3(\delta_{\mathbf{T}})$. 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. $\square$

2.2 The case of univariate polynomials

We next discuss a special case, that of triangular sets consisting of univariate polynomials: while this may seem like an easy problem, no fast algorithm was known to us.

We provide here a first quasi-linear algorithm, that works under mild assumptions. However, the techniques used (deformation ideas, see [2, 3, 1]) induce huge logarithmic factors (as they do in matrix multiplication algorithms of low exponent). Hence, this should mainly be considered as a feasibility result.

Theorem 2. *For any $\varepsilon > 0$, there exists a constant K_ε such that the following holds. Let $\mathbb{K}$ be a field and $\mathbf{T} = (T_1, \dots, T_n)$ be a triangular set of multi-degree $(d_1, \dots, d_n)$ in $\mathbb{K}[X_1] \times \cdots \times \mathbb{K}[X_n]$, with $2 \le d_i \le |\mathbb{K}|$ 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 $\mathbb{K}$.*

Proof. The proof is divided into three steps: we first discuss the particular case where all polynomials factor, and use it in a second time to obtain estimates involving the regularity $r_\mathbf{T}$. The final step consists in combining these results to the one of the previous subsection, obtaining the claimed bound using suitable thresholds, as in [25].

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

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

with $\alpha_{i,j} - \alpha_{i,j'}$ a unit in R for $j \ne j$'. Then, we prove that *given A, B in $\mathbb{L}_\mathbf{T}$, one can compute $AB \in \mathbb{L}_\mathbf{T}$ using at most $\mathsf{K}'\, \delta_\mathbf{T} \lg^3(\delta_\mathbf{T})$ operations $(+, \times, \div)$ in R, for a universal constant K'*. The proof reduces to an evaluation / interpolation process. Let $V \subset R^n$ be the grid $[(\alpha_{1,\ell_1}, \dots, \alpha_{n,\ell_n}) \mid 0 \le \ell_i < d_i]$ and define the evaluation map

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

In view of our assumption that all $\alpha_{i,j} - \alpha_{i,j'}$ are units, the map Eval is invertible. To perform evaluation and interpolation, we use the algorithm in [21, Section 2]. This gives the recursion for the cost $\mathsf{C}_{\mathsf{Eval}}$ of evaluation

$$\mathsf{C}_{\mathsf{Eval}}(d_1, \dots, d_n) \le \mathsf{C}_{\mathsf{Eval}}(d_1, \dots, d_{n-1})\, d_n + \mathsf{k}\, d_1 \cdots d_{n-1} d_n \lg^3(d_n)$$

and implies the bounds

$$\mathsf{C}_{\mathsf{Eval}}(d_1, \dots, d_n) \le \mathsf{k}\, \delta_\mathbf{T} \textstyle\sum_{i \le n} \lg^3(d_i)$$

which is bounded by $\mathsf{k}\, \delta_\mathbf{T} \lg^3(\delta_\mathbf{T})$. The recursion (and thus the bounds) for interpolation are the same as for evaluation.

It is then easy to prove our claim: 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 claimed estimate follows by taking $\mathsf{K}' = 2\mathsf{k} + 1$.

Step 2. We continue with by an approach using deformation techniques, giving good results for triangular sets made of many polynomials of low degree. Under the assumptions of Theorem 2 (thus, without supposing any factorization property), we prove that *given A, B in $\mathbb{L}_\mathbf{T}$, one can compute the product $AB \in \mathbb{L}_\mathbf{T}$ using at most*

$$\mathsf{K}''\, \delta_\mathbf{T}\, r_\mathbf{T} \left(\log(\delta_\mathbf{T}) \log(r_\mathbf{T}) \right)^3 \qquad (1)$$

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

Let thus $\mathbf{T} = (T_1, \dots, T_n)$ be a triangular set with T_i in $\mathbb{K}[X_i]$ of degree d_i for all i. Let $\mathbf{U} = (U_1, \dots, U_n)$ be the set of 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 η be a new variable, and define $\mathbf{V} \subset \mathbb{K}[\eta][\mathbf{X}]$ by $V_i = \eta T_i + (1-\eta) U_i$, so that V_i is in 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 1. *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 $AB \bmod \langle \mathbf{T} \rangle$.*

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 $A(\mathsf{Mat}(X_1, \mathbf{V}), \dots, \mathsf{Mat}(X_n, \mathbf{V}))$; 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. $\square$

Let R be the truncated series 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 $AB \bmod \langle \mathbf{T} \rangle$.

Lemma 2. *One can compute $[\alpha_{i,j} \in R \mid 1 \le i \le n,\ 0 \le j < d_i]$ with $\alpha_{i,j} - \alpha_{i,j'}$ invertible for $j \ne j'$, and such that*

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

holds in $R[X_i]$ for all i, using $\mathsf{k}' \delta_\mathbf{T}\, r_\mathbf{T} \left(\lg(\delta_\mathbf{T}) \lg(r_\mathbf{T})\right)^3$ operations in $\mathbb{K}$, for a constant k'.

Proof. The polynomial $U_i = V_i(0)(\mathbf{X})$ splits into a product of linear terms in $\mathbb{K}[X_i]$, with no repeated root, so the conclusion follows by Hensel's lemma. Theorem 15.18 in [10] gives a cost in $O(\mathsf{M}(d_i)\mathsf{M}(r_\mathbf{T}) \log(d_i))$, from which our claim follows after a few (very rough) simplifications. $\square$

We can now conclude the proof of estimate (1). To compute $AB \bmod \langle \mathbf{T} \rangle$, we compute $C_\eta = AB \bmod \langle \mathbf{V} \rangle$ in $R[\mathbf{X}]$, deduce $C \in \mathbb{K}[\eta][\mathbf{X}]$ and evaluate it at 1. By the previous lemma, we can apply the previous evaluation / interpolation techniques over R to compute C_η. An operation $(+, \times, \div)$ in R has cost $O(\mathsf{M}(r_\mathbf{T}))$. Combining this with the costs given in Step 1 and Lemma 2 gives the claimed result after a few simplifications.

Step 3: conclusion. To prove Theorem 2, we combine the two previous approaches (the general case and the deformation approach). Let ε be a positive real, and define

$\omega = 2/\varepsilon$. We can assume that the degrees in $\mathbf{T}$ are ordered as $2 \le d_1 \cdots \le d_n$, with in particular $\delta_{\mathbf{T}} \ge 2^n$. Define an index ℓ by the condition that $d_\ell \le 4^\omega \le d_{\ell+1}$, taking $d_0 = 0$ and $d_{n+1} = \infty$, 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 cost estimate (1), a product in R can be done in

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

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 $\mathsf{K}\, 4^{n-\ell} \delta_{\mathbf{T}''} \lg^3(\delta_{\mathbf{T}''})$ operations in R. Hence, taking into account that $\delta_{\mathbf{T}} = \delta_{\mathbf{T}'} \delta_{\mathbf{T}''}$, the total cost for one operation in $\mathbb{L}_{\mathbf{T}}$ is at most

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

operations in $\mathbb{K}$. Now, observe that $r_{\mathbf{T}'}$ is upper-bounded by $d_\ell n \le 2^\omega \lg(\delta_{\mathbf{T}})$. This implies that the factor

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

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 $d_{\ell+1} \cdots d_n \le \delta_{\mathbf{T}}$. Raising to the power $\varepsilon/2$ yields $4^{n-\ell} \le \delta_{\mathbf{T}}^{\varepsilon/2}$; thus, the previous estimate admits the upper bound

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

To conclude, it suffices to observe that $\lg^{10}(\delta_{\mathbf{T}})$ admits an upper bound of the form $\mathsf{H}' \delta_{\mathbf{T}}^{\varepsilon/2}$, where H' depends on ε.

3. IMPLEMENTATION TECHNIQUES

The algorithm of Subsection 2.1 was implemented in C (that in Subsection 2.2 is not expected to be very practical, except for very large number of variables and very low degree). As in [9, 17], the C code was then interfaced with AXIOM. In this section, we describe the features of our implementation in a bottom-up manner.

Arithmetic in $\mathbb{F}_p$. Our implementation is devoted to small finite fields $\mathbb{F}_p$, with p a prime that fits in a machine word.

Multiplications in $\mathbb{F}_p$ are done using Montgomery's REDC routine [19], trading one integer division for two products. A straightforward implementation does not bring better performance than Shoup's floating point techniques [26]. However, we designed an improved scheme which decomposes Montgomery's formula into three easily computable parts. This lowers the operations done in the reduction algorithm by 2 double word shifts and 2 single word shifts [16].

Compared with the implementation in [9], this approach is more portable, as it does not rely on the use of machine features like SSE2 registers, and brings a speed-up of about 50% on Pentium 4 processors.

Arithmetic in $\mathbb{F}_p[X]$. As can be seen in Subsection 2.1, multiplication modulo a triangular set boils down to multivariate polynomial multiplications; eventually, these can be reduced to univariate multiplications through Kronecker's substitution.

We use classical and FFT multiplication for univariate polynomial arithmetic over $\mathbb{F}_p$, for p a Fourier prime. For base fields without roots of unity, we use Chinese Remaindering techniques as in [26].

Our FFT multiplication routine is the one presented in [7]; the implementation is essentially the one described in [9], up to a few modifications to improve cache-friendliness. We tried several data accessing patterns in our FFT implementation; 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 [12, 11, 22].

Multivariate arithmetic over $\mathbb{F}_p$. At the C level, we use a dense representation for multivariate polynomials: important applications of modular multiplication are GCD computations and Hensel lifting for triangular sets, which tend to produce dense polynomials. Therefore, 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 [9] or standard multi-dimensional 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 hindered for larger number of variables by the cost of matrix transposition.

Triangular sets over $\mathbb{F}_p$. Triangular sets are represented in C by an array of multivariate polynomials. Two strategies for modular multiplication were implemented, a plain one and the one relying on the algorithm of Subsection 2.1.

Both of them 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 $\mathsf{MulTrunc}$ of Subsection 2.1. 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 relies on some precomputation (of the power series inverses of the reciprocals of the polynomials T_i). While the complexity analysis takes the cost of this precomputation into account, 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 FFT transforms; this is work in progress.

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 itself still far from being optimal.

The AXIOM level. Integrating our fast arithmetic into

AXIOM is straightforward; most of the effort was concentrated on the following aspects. First, AXIOM is a Lisp-based system (we use Open AXIOM, which is based on GNU common Lisp), whereas our package is implemented in C. Second, in AXIOM, dense multivariate polynomials are represented by recursive trees. However, in our C package, they are encoded as multi-dimensional arrays.

Regarding inter-language communications, objects in Open AXIOM are constructed as GCL data structures, which are themselves implemented by C data types. Therefore, we can manipulate any AXIOM data object at the C level. Conversations from tree to array structures (and backward) are done in C. This converter is linked to the GCL kernel, becoming an AXIOM system routine which can be used at run-time. The overhead induced by data conversion is negligible.

4. EXPERIMENTAL RESULTS

4.1 Comparing different strategies

We start by reporting on experiments comparing different strategies for computing products modulo triangular sets in $n = 1, 2, 3$ variables. For the entire set of benchmarks, we use random dense polynomials. Our experiments were conducted on a PC 2.80 GHz Pentium 4, with 1GB memory and 1024 KB cache.

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 $(T_1, \ldots, T_n)$ of multi-degree $(d_1, \ldots, d_n)$. As explained in Subsection 2.1, 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 will discuss three strategies for computing C.

PLAIN. We use univariate Euclidean division for reducing P; 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 and we assume that the inverses $\mathbf{S}$ have been precomputed.

FAST WITHOUT PRECOMPUTATIONS. We apply again the algorithm $\mathsf{Rem}(C, \mathbf{T}, \mathbf{S})$, but in this case, we do not assume that $\mathbf{S}$ has been precomputed, so we compute it beforehand.

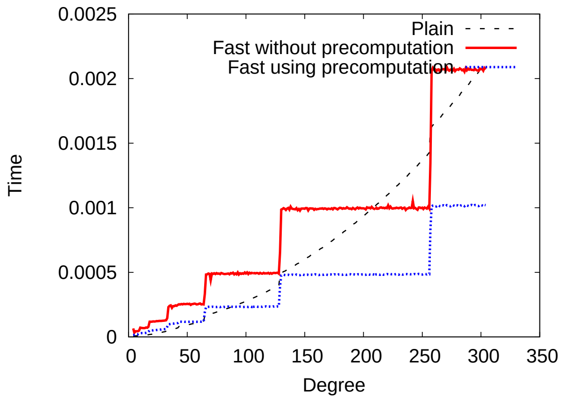

Figure 2: Multiplication in $\mathbb{L}_1$, all strategies.

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. Actually, the alternative is either between PLAIN and FAST USING PRECOMPUTATIONS, or PLAIN and FAST WITHOUT PRECOMPUTATIONS, depending on the assumption regarding precomputations. For applications discussed before (quasi-inverses, polynomial GCDs modulo a triangular set), using precomputations is realistic.

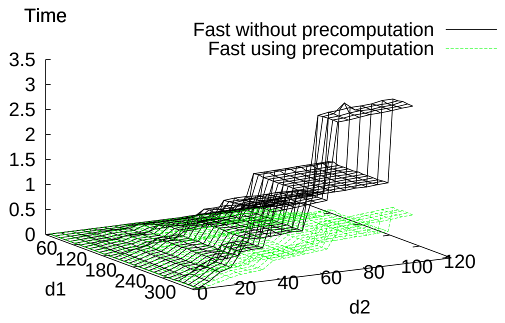

Figure 3: Multiplication in $\mathbb{L}_2$, PLAIN vs. fast without precomputations.

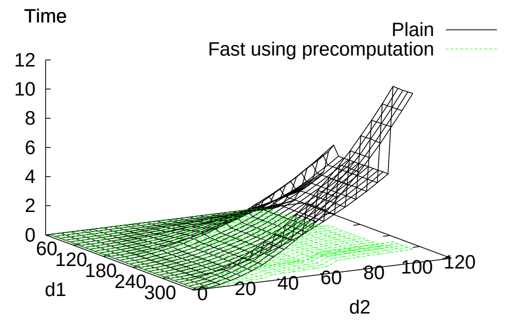

Figure 4: Multiplication in $\mathbb{L}_2$, PLAIN vs. fast using precomputations.

Univariate results. In the case of $\mathbb{L}_1$, finding the cut-offs is straightforward. Figure 2 shows that the FAST USING PRECOMPUTATIONS strategy takes the lead for $d_1 \geq 146$ over the PLAIN one. If precomputations are not assumed, then this cut-off doubles. From now on, all computations in $\mathbb{L}_1$ will use these thresholds.

Bivariate results. For $n = 2$, we let d_1 and d_2 vary in the ranges $4, \ldots, 304$ and $2, \ldots, 102$. Figures 3 and 4 illustrate the performances of our different strategies in these degree

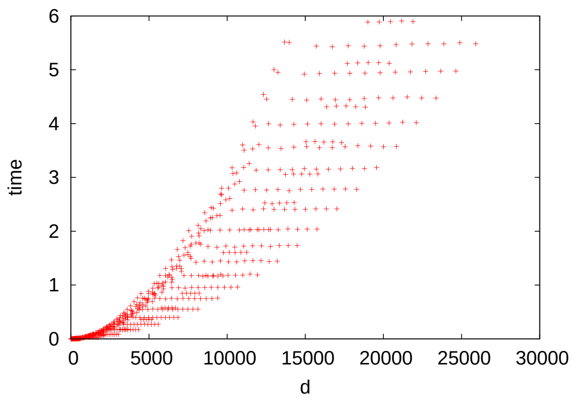

Figure 5: PLAIN multiplication in $\mathbb{L}_2$, time as a function of $d = d_1 d_2$.

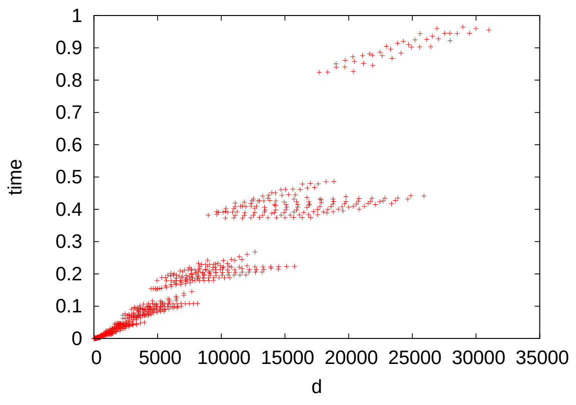

Figure 6: Fast multiplication in $\mathbb{L}_2$ using precomputations, time as a function of $d = d_1 d_2$.

ranges. This allows us to determine a cut-off for d_2 as a function of d_1. Surprisingly, our experiments show that this cut-off is essentially independent of d_1 and can be chosen equal to 5. We discuss this point below.

In order to continue our benchmarks in $\mathbb{L}_3$ as we did for $\mathbb{L}_2$, 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 following 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 $2 \leq d_2 \leq 102$? Figures 5 and 6 give all timings obtained for this sample set; they show that the time does vary, mostly for the PLAIN strategy (the levels appearing in the fast case are due to our FFT multiplication). This observation will guide our experiments in $\mathbb{L}_3$.

Trivariate results. For our experiments with the multiplication in $\mathbb{L}_3$, following the previous observation, we consider three patterns for (d_1, d_2). Pattern 1 has $d_1 = 2$, Pattern 2 has $d_1 = d_2$ and Pattern 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 restrict our report to the comparison between the strategies PLAIN and FAST USING PRECOMPUTATIONS. The timings are reported in Figures 7 to 9.

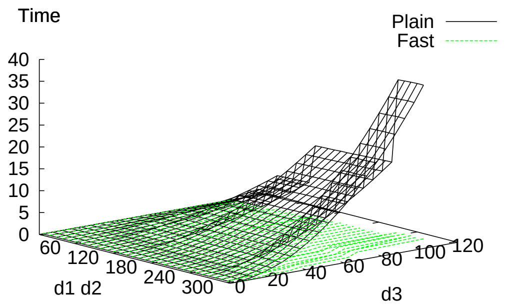

Figure 7: Multiplication in $\mathbb{L}_3$, pattern 1, PLAIN vs. fast.

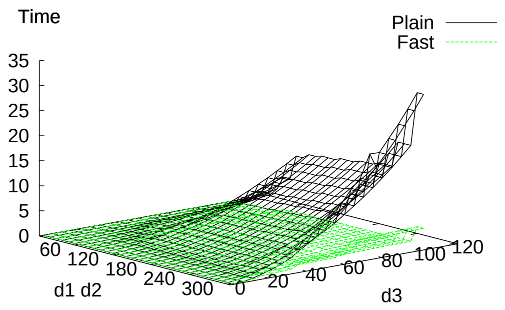

Figure 8: Multiplication in $\mathbb{L}_3$, pattern 2, PLAIN vs. fast.

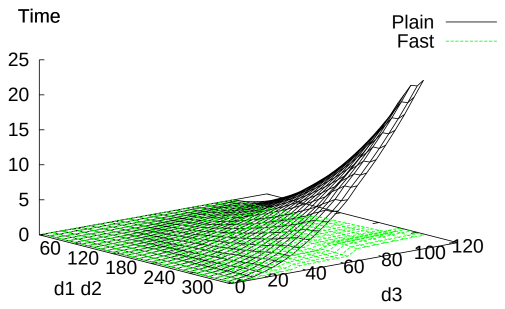

Figure 9: Multiplication in $\mathbb{L}_3$, pattern 3, PLAIN vs. fast.

For the extreme case $(d_1 d_2, d_3) = (304, 102)$ the ratio between the timings for PLAIN and FAST are 16.8, 7.2 and 12.4, respectively, showing 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. Performing experiments as in Figures 5 and 6 gives similar conclusion: the timing depends not only on the product $d_1 d_2$ 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 $\simeq (d^2, d^2)$ operations $(\times, +)$ respectively; 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 $\simeq (2d^2, 2d^2)$ operations $(\times, +)$ in R. Additions modulo $\langle T \rangle$ take time d.

Thus, a purely recursive approach for multiplication in $\mathbb{L}_2$ uses approximately $(4d_1^2 d_2^2, 4d_1^2 d_2^2)$ 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 $\simeq (2d_1^2 d_2^2, 2d_1^2 d_2^2)$. Hence, we can already make some advantage of fast FFT-based multiplication, since we traded $2d_1^2 d_2^2$ base ring multiplications and as many addition for a bivariate product.

Using precomputations, the Fast approach performs 3 bivariate products in degrees $\simeq (d_1, d_2)$ and $\simeq 4d_2$ 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 $d_1^2 d_2^2$. 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 $\simeq (4d_1^2 d_2, 4d_1^2 d_2)$, 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 [4], which has set a standard of efficient implementation of low-level algorithms.

We compared multiplication in $\mathbb{L}_3$ for the three patterns of the previous subsection; the degree ranges are the same as above. Figure 10 gives the timings for Pattern 3. The Magma implementation uses several `quo` constructs over `UnivariatePolynomial`, as this was the most efficient configuration we found. We include only the time for performing the actual computation; sometimes, the time spent in the construction of the structure was more important. For our code, we use the strategy Plain using precomputations. On this example, our code outperforms Magma by factors up to 7.4; other patterns yield similar behaviour.

Comparison with Maple. One of our main purposes in the long term is to design high-performance implementations of higher-level algorithms, such triangular decomposition algorithms [20, 8], in languages such as AXIOM or Maple, by replacing built-in arithmetic by our C implementation. While it is easy to put this idea to practice with AXIOM (see below), putting it to use within Maple is not straightforward as of now.

Hence, our Maple experimentations stayed at the level of GCD and inversions. We compared our code with Maple's `recden` library for computing inverses in $\mathbb{L}_3$. We used the same degree patterns as before, but we were led to reduce the degree ranges to $4 \leq d_1 d_2 \leq 204$ and $2 \leq d_3 \leq 20$. Figure 11 gives the timings for pattern 3, where our code uses the strategy Fast using precomputations. There is a huge performance gap, such that our timing surface is very close the bottom. The other results are similar.

The Maple `recden` library is a C library which implements multivariate dense recursive polynomials and which

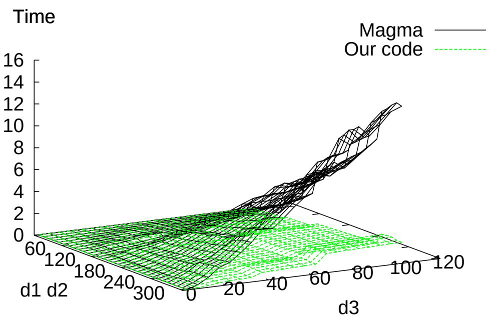

Figure 10: Multiplication in $\mathbb{L}_3$, pattern 3, Magma vs. our code.

be can 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`.

We observe a huge gap between the two implementations. For instance, for $(d_1 d_2, d_3) = (204, 20)$ for patterns 1 and 2, the ratio between the `recden` timings and ours is 506.4/2.0 and 36.7/1.6 respectively. The Maple code could not reach these degrees for pattern 3, as too much memory was needed when constructing the algebraic structure $\mathbb{L}_3$. When using our Plain strategy, our code remains faster, but the ratio diminishes by a factor of about 4.

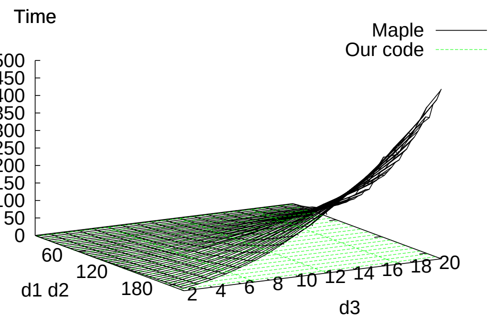

Figure 11: Inversion in $\mathbb{L}_3$, pattern 1, Maple vs. our code.

Comparison with AXIOM. As explained in Section 3, using our lower-level arithmetic is AXIOM is made easy by the multiple-level C/GCL structure.

In [17], the modular algorithm by van Hoeij and Monagan [18] 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 prime numbers p.

We implemented the top-level of this algorithm in AXIOM. Then, two strategies were used: one relying on the built-in AXIOM modular arithmetic, and the other on our C code. Since our C code provides all conversion functions from and to AXIOM, the only difference between the two strategies at the top-level resides in which GCD function to call. The results are reported in Figure 12, where we consider polynomials in $\mathbb{Q}[X_1, X_2, X_3]/\langle T_1, T_2, T_3\rangle[X_4]$, with input coefficients of absolute value bounded by 2. As shown in Figure 12 the performance gap between the two implementations increases dramatically.

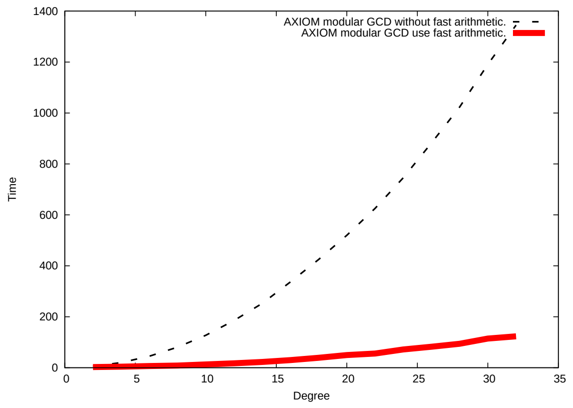

Figure 12: GCD computations $\mathbb{L}_3[X_4]$, pure AXIOM code vs. combined C-AXIOM code.

5. CONCLUSION

On the theoretical level, this article provides new estimates for the cost of multiplication modulo a triangular set. The outstanding challenge for this question remains the suppression of exponential overheads; even though we were able to do so in a particular case, the general case still resists. An tempting approach would consist in a higher-dimensional extension of the Cook-Sieveking-Kung idea, or the related Montgomery approach; the main stumbling block is that multivariate power series ideals are usually not principal.

On the software level, our experiments show the importance of both fast algorithms and implementation techniques. While most of our efforts were limited to multiplication, the next steps are well-tuned inversion and GCD computations. Theory and practice revealed that, as far as multivariate multiplication is concerned, fast algorithms become faster than plain ones for very low degrees. An outstanding question is whether this phenomenon extends to half-GCD techniques in several variable.

6. REFERENCES

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

[2] 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.

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

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

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

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

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* McGraw-Hill, 2002.

[8] 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.

[9] 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.

[10] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.

[11] 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.

[12] J. R. Johnson, W. Krandick, and A. D. Ruslanov. Architecture-aware classical taylor shift by 1. In *ISSAC'05*, pages 200–207. ACM, 2005.

[13] H. T. Kung. On computing reciprocals of power series. *Numerische Mathematik*, 22:341–348, 1974.

[14] L. Langemyr. Algorithms for a multiple algebraic extension. In *Effective methods in algebraic geometry)*, volume 94 of *Progr. Math.*, pages 235–248. Birkhäuser, 1991.

[15] F. Lemaire, M. Moreno Maza, and Y. Xie. The `RegularChains` library. In Ilias S. Kotsireas, editor, Maple Conference 2005, pages 355–368, 2005.

[16] X. Li. Low-level techniques for Montgomery multiplication, 2007. `http://www.csd.uwo.ca/People/gradstudents/xli96/`.

[17] 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.

[18] 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.

[19] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[20] 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/`.

[21] V. Y. Pan. Simple multivariate polynomial multiplication. *J. Symbolic Comput.*, 18(3):183–186, 1994.

[22] 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.

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

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

[25] É. Schost. Multivariate power series multiplication. In *ISSAC'05*, pages 293–300. ACM, 2005.

[26] V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symb. Comp.*, 20(4):363–397, 1995.

[27] M. Sieveking. An algorithm for division of powerseries. *Computing*, 10:153–156, 1972.