High-Performance Symbolic Computation in a Hybrid Compiled-Interpreted Programming Environment

Xin Li
Ontario Research Center for Computer Algebra
University of Western Ontario
London, Ontario, Canada
xli96@csd.uwo.ca

Marc Moreno Maza
Ontario Research Center for Computer Algebra
University of Western Ontario
London, Ontario, Canada
moreno@csd.uwo.ca

Raqeeb Rasheed
Ontario Research Center for Computer Algebra
University of Western Ontario
London, Ontario, Canada
rrasheed@uwo.ca

Éric Schost
Ontario Research Center for Computer Algebra
University of Western Ontario
London, Ontario, Canada
eschost@csd.uwo.ca

Abstract

We investigate the integration of C implementation of fast arithmetic operations into MAPLE*, focusing on triangular decomposition algorithms. We show substantial improvements over existing* MAPLE *implementations; our code also outperforms* MAGMA *on many examples. Profiling data show that data conversion can become a bottleneck for some algorithms, leaving room for further improvements.*

1 Introduction

Since the early days of computer algebra systems, their designers have investigated many aspects of this kind of software. For the systems born in the 70's and 80's, such as AXIOM and MAPLE, the primary concerns were probably the expressiveness of the programming language and the convenience of the user interface; the implementation of modular methods for operations such as polynomial factorization was also among these concerns.

Computer algebra systems born in the 90's, such as MAGMA and NTL, have brought forward a new priority: the implementation of asymptotically fast arithmetic for polynomials and matrices. They have demonstrated that, for relatively small input data size, FFT-based polynomial operations could outperform operations based on classical quadratic algorithms or on the Karatsuba trick. With this breakthrough, they increased in a spectacular manner the range of problems solvable by computer algebra systems.

Meanwhile AXIOM and MAPLE remain highly attractive: the former one by its programming environment and the latter one by its users community.

In previous work [6, 11, 15] we have investigated the integration of asymptotically fast arithmetic operations into AXIOM. Since AXIOM is based today on GNU Common Lisp (GCL), we took the following approach. We realized highly optimized implementations of these fast routines in C and made them available to the AXIOM programming environment through the kernel of GCL, Therefore, library functions written in the AXIOM high-level language could be compiled down to binary code and then linked against our C code. To observe significant speed-up factors, it was sufficient to extend existing AXIOM polynomial domain constructors with our fast routines (for univariate multiplication, division, GCD etc.) and call them in existing generic packages (for instance, for univariate squarefree factorization). See [15] for details.

Few other languages allow the integration of user-written C code in the kernel. For instance, MAGMA allows users to open pipes or sockets to communicate with external programs, but such programs cannot call MAGMA functions; within MAGMA, users can define *packages*, where functions are compiled in MAGMA internal pseudo-code, but not in C.

In the present paper, we investigate the integration of fast arithmetic operations implemented in C into MAPLE. Most of MAPLE library functions are high-level interpreted code. This is the case for those of the `RegularChains` library, our main focus here, which could greatly benefit from our fast routines for triangular decompositions [14, 12]. This question is made more difficult by the following factors.

First, and up to our knowledge, the connection between C and MAPLE code is simple but quite rudimentary. The only structured data which can be exchanged by the two sides are the simple ones such as strings, arrays, tables. This leads to conversion overheads. Indeed, generally, MAPLE polynomials are represented by sparse data-structures whereas those used by fast arithmetic operations are dense.

This situation implies a second downside factor: conversions between C and MAPLE must be performed on the MAPLE side or both sides, as interpreted code. Clearly, one would like to implement them on the C side, as compiled and optimized code.

The fact that the MAPLE language does not enforce "modular programming" or "generic programming" is a third disadvantage compared to AXIOM integration. Providing a MAPLE *connection-package* capable of calling our efficient C routines will not be sufficient to speed-up all MAPLE libraries using polynomial arithmetic. Clearly, high-level MAPLE code needs to be rewritten to call this connection-package and obtain improved performances.

These constraints being raised, bearing in mind that we aim at achieving high-performance, we can now state the questions which have motivated the design of a framework in this compiled-interpreted programming environment, together with the experimental evaluation of this framework.

($Q1$) To which extent triangular decomposition algorithms (implemented in the `RegularChains` library in MAPLE) can take advantage of fast polynomial arithmetic (implemented in C)?

($Q2$) What is a good design for such hybrid applications?

($Q3$) Can an implementation based on this strategy outperform other highly efficient computer algebra packages performing similar computations?

($Q4$) Does the observed performance of this hybrid C-MAPLE application comply to its estimated performance by complexity analysis?

This paper attempts to provide elements of answers to these questions. In Section 2, we start by describing the framework that we have designed in this compiled-interpreted programming environment. In Sections 3 and 4 we present the three applications that we have implemented in this framework. We introduce them hereafter (definitions are given in the latter sections).

Bivariate solver. This application takes as input a polynomial system of two equations F_1, F_2 in two variables $X_1 < X_2$ and with coefficients in a prime field $\mathbb{K}$ (whose size is a machine word size Fourier prime). It returns a triangular decomposition of the common roots of F_1 and F_2.

Two-equation solver. This application takes as input two polynomials F_1, F_2 in several variables $X_1 < \cdots < X_n$ and with coefficients in $\mathbb{K}$. It returns the resultant R_1 of F_1, F_2 w.r.t. X_n and a regular GCD of F_1, F_2 modulo (the primitive part of) R_1. This application is an extension of the previous one to the case of two equations with an arbitrary number of variables.

Invertibility test. This application takes as input a zero-dimensional regular chain T and a polynomial p. It separates the points of the zero set $V(T)$ of T that cancel p from those which do not. More precisely, this application computes two triangular decompositions: one for $V(T) \cap V(p)$ and one for $V(T) \setminus V(p)$. This is a fundamental operation when computing modulo a regular chain. It is used, actually, by our two other applications.

In each case, the "top-level" algorithm is written in MAPLE and relies on our C routines for different tasks such as the computation of subresultant chain, normal form of a polynomial w.r.t. a zero-dimensional regular chain, etc.

These three applications perform triangular decompositions of a polynomial system of different types. They are therefore well representative of the high-level MAPLE code that we aim at improving with our C routines, while also simple enough such that their performance can be sharply evaluated. Moreover, these applications put to challenge our framework in different ways, revealing its strengths and weaknesses. Our experimental results are reported and analyzed in Section 5.

2. A Compiled-Interpreted Programming Environment

Our library contains two levels of implementation: MAPLE code (interpreted) and C code (compiled); our purpose is to reach high-performance while spending reasonable amount of development time.

Relying on asymptotically fast algorithms, the C level routines are highly optimized. The core operations are fast operations modulo triangular sets (multiplication / inversion as in [14]), gcd's, resultants, lifting techniques [21] and fast interpolation. This library of functions is called `modpn` and is introduced in more detail in [12]. At the MAPLE level, we write more abstract algorithms; typically, these are higher level polynomial solvers. The major trade-off between two levels are language abstraction and high-performance.

We use multiple polynomial data encodings at each level: some encodings are specifically devoted to some algorithms; others "intermediate" encodings are written to speed-up data conversions.

There are multiple issues to take care of: what operations should be written in C, how to map the MAPLE data to C ones and vice versa, to what extent we should rely

on existing packages or develop our own ones, etc. Of the questions mentioned in the introduction, we discuss the following ones here: to which extent triangular decomposition algorithms can take advantage of fast polynomial arithmetic implemented in C, and what is a good design for a hybrid C-MAPLE application.

2.1 The C Level

Primarily, our C code targets on the best performance. All operations are based on asymptotically fast algorithms rooted at Fast Fourier Transform (FFT) and its variant Truncated Fourier Transform (TFT) [7]. These operations are optimized with respect to crucial features of hardware architecture: memory hierarchy, instruction pipe-lining, and vector instructions. As reported in [14, 11, 6], our C library often outperforms the best known implementations such as Magma and NTL [23, 22].

The C code is dedicated to triangular set operations modulo a machine size prime number. Such computations typically generate dense polynomials; thus, we use multidimensional arrays as the canonical encoding, and we call them CUBEs (since all partial degrees are bounded in advance). This encoding is the most appropriate for FFT based multiplication, inversion, resultant modulo a triangular set, interpolation, ... Besides, we can pre-allocate the working buffer and use in-place operations whenever applicable. Tracing coefficients and degrees also becomes trivial. Locality of reference is easily preserved by transposing or permitting the data inside these CUBEs [1].

Besides the CUBE encoding, we used another polynomial encoding at C level. With a view to apply triangular lifting algorithms [21, 4], we designed a Directed Acyclic Graph representation (DAG). By setting "smart" flags in the nodes of these DAGs, we can track the information of visibility, liveness, and reducibility information in constant time. We do not report on such operations here; see [13].

We implemented a third data structure, the 2VECTOR encoding, which is dedicated to facilitate the conversion between CUBE and MAPLE's `RecDen`(recursive dense) encoding, and is described below.

2.2 The MAPLE Level

Our algorithms for triangular decompositions are of a higher level, so it seems sensible to implement them in a well equipped interpreted environment like MAPLE. First, the implementation effort is much less intensive than in C or C++; besides, MAPLE has comprehensive mathematical libraries, so it is possible to use different implementations of the same algorithm to verify our results. In our case, we checked our results using the `Triade` and `RegularChains` packages [8, 9].

At the MAPLE level, we use two types of polynomials: MAPLE DAGs and `RecDen` (recursive dense) polynomials. DAGs are the canonical data representation for MAPLE polynomials; `Triade` and `RegularChains` use them uniformly. Thus, to access functionalities from these packages, we need to use MAPLE DAGS. In addition, we used `RecDen` when implementing dense polynomial algorithms in MAPLE: operations modulo a triangular set are essentially dense methods, so that `RecDen` is one of the best candidate at the MAPLE level.

When designing our algorithms, we tried to rely on our C library's fast arithmetic for the efficiency critical operations. Recall our first question: is this an effective approach? Our answer is a conditional yes: if the integration process is careful, our C level fast arithmetic provides a large speed-up for the MAPLE code; this is reported in Section 5.

2.3 MAPLE and C Cooperation

For general MAPLE users (as we are), the use of the `ExternalCalling` package is the standard way to link in externally defined C functions. The action of linking is not very complicated: the user just needs to carefully map MAPLE level data onto C. For example, a MAPLE `rtable` type can be directly mapped to a C array. However, if the MAPLE data encoding is different from the C one, an important issue arises, data conversion.

This a difficult problem in our design. Only a small group of simple MAPLE data structures, such as integers, arrays or tables, can be automatically converted. When the data structure are DAGs, we have to manually pack the data into a buffer, and unpack it at the target level. Especially when the conversions mostly happen at the MAPLE level, the overhead may be significant.

There are two major ways to reduce this overhead: carefully designing the algorithm to reduce the total number of conversions, and implementing efficient converters to minimize the time of each unit conversion.

The frequency of encoding conversions is application dependent; it turns out that it can happen quite often in our algorithms for triangular decomposition. Hence, we try to reuse C objects as much as possible. Many conversions are "voluntary": we are willing to conduct them, expecting that better algorithms or better implementations can then be used in C. However, some conversions are "involuntary". Indeed, even if we would like all computational intensive operations be carried out at the C level, our algorithms are complex, so that it becomes unrealistic to implement everything in C. Thus, there are cases where we have to convert polynomials from C to MAPLE and use its library operations.

The second direction – minimizing the cost of each unit conversion – is crucial as well. As mentioned above, we

designed a so-called 2VECTOR polynomial representation: one vector recursively encodes the degrees of all polynomial coefficients, and another vector all the coefficients, in the same traversal order. This data representation in our library does not participate to any real computation: it is specifically designed for facilitating the data conversion from CUBE to `RecDen` encoding. The 2VECTOR encoding has the same recursive structure as `RecDen`, so the mapping is easy between these two. Besides, the 2VECTOR encoding use flattened polynomial tree structures, which are convenient to pass from C to MAPLE.

It remains to estimate if the benefits from the conversion outweighs the overhead of the conversion itself. As reported in Section 5, for certain cases the data conversion time become dominant, thus the corresponding algorithm needs to be adjusted to reduce number of conversions.

3 Bivariate Solver

The first application that we use to evaluate our software framework is the solving of bivariate polynomial systems by means of triangular decompositions. We consider two bivariate polynomials F_1 and F_2, with ordered variables $X_1 < X_2$ and with coefficients in a field $\mathbb{K}$. We assume that $\mathbb{K}$ is perfect; in our experimentation $\mathbb{K}$ is a prime field whose characteristic is a machine word size prime.

We rely on an algorithm introduced in [19, 12] and based on the following well-known fact [2]. The common roots of F_1 and F_2 over an algebraic closure $\overline{\mathbb{K}}$ of $\mathbb{K}$ are "likely" to be described by the common roots of a system with a triangular shape:

$$\left\{ \begin{array}{rcl} T_1(X_1) & = & 0 \\ T_2(X_1, X_2) & = & 0 \end{array} \right.$$

such that the leading coefficient of T_2 w.r.t. X_2 is invertible modulo T_1; moreover the degree of T_2 w.r.t. X_2 is "likely" to be 1. For instance, the system

$$\left\{ \begin{array}{rcl} X_1^2 + X_2 + 1 & = & 0 \\ X_1 + X_2^2 + 1 & = & 0 \end{array} \right.$$

is *solved* by the triangular system

$$\left\{ \begin{array}{rcl} X_1^4 + 2X_1^2 + X_1 + 2 & = & 0 \\ X_2 + X_1^2 + 1 & = & 0 \end{array} \right.$$

The goal of this section is to show that this algorithm can easily be implemented in our software framework while providing high-performance. In Section 3.1 we review briefly the necessary mathematical concepts. Sections 3.2 and 3.3 contain the algorithm and the corresponding code, respectively.

3.1 Theoretical Background

The main theoretical tools of our bivariate solver algorithm are subresultant theory and polynomial GCDs modulo regular chains. Classical textbooks for the former are [24, 16] whereas the latter was introduced in [18].

Subresultant theory. In Euclidean domains such as $\mathbb{K}[X_1]$, polynomial GCDs can be computed by the Euclidean Algorithm and by the subresultant algorithm (we refer here to the algorithm presented in [5]) .

Consider next more general rings, such as $\mathbb{K}[X_1, X_2]$. Assume F_1, F_2 are non-constant polynomials with $\deg(F_1, X_2) \geq q := \deg(F_2, X_2)$. The polynomials computed by the subresultant algorithm form a sequence, called the *subresultant chain* of F_1 and F_2 and denoted by $\mathrm{src}(F_1, F_2)$. This sequence consists of $q + 1$ polynomials, starting at $\mathrm{lc}(F_2, X_2)^\delta F_2$, with $\delta = \deg(F_1, X_2) - \deg(F_2, X_2)$, and ending at $R_1 := \mathsf{res}(F_1, F_2)$, the resultant of F_1 by F_2 w.r.t. X_2. We write this sequence $S_q, \ldots, S_0$ where the polynomial $S_j := S_j(F_1, F_2)$ is called the *subresultant* (*of* F_1, F_2) *of index* j. Let j be an index such that $0 \leq j \leq q$. If S_j is not zero, it turns out that its degree is at most j and S_j is said *regular* when $\deg(S_j, X_2) = j$ holds.

The subresultant chain of F_1 and F_2 satisfies a fundamental property, called the *block structure*, which implies the following fact: if the subresultant S_j of index j with $j < \deg(F_2, X_2) - 1$, is not zero and not regular, then there exists a non-zero subresultant S_i with index $i < j$ such that S_i is regular, has the same degree as S_j and for all $i < \ell < j$ the subresultant S_ℓ is null.

The subresultant chain of F_1 and F_2 satisfies another fundamental property, called the *specialization property*, which plays a central in our algorithm. Let Φ be a homomorphism from $\mathbb{K}[X_1, X_2]$ to $\overline{\mathbb{K}}[X_2]$, with $\Phi(X_1) \in \overline{\mathbb{K}}$. Assume $\Phi(a) \neq 0$ where $a = \mathrm{lc}(f_1, X_2)$. Then we have:

$$\Phi(S_j(F_1, F_2)) = \Phi(a)^{q-k} S_j(\Phi(F_1), \Phi(F_2)) \qquad (1)$$

where $q = \deg(F_2, X_2)$ and $k = \deg(\Phi(F_2), X_2)$.

Regular GCDs modulo regular chains. Let $T_1 \in \mathbb{K}[X_1] \setminus \mathbb{K}$ and $T_2 \in \mathbb{K}[X_1, X_2] \setminus \mathbb{K}[X_1]$ be two polynomials. Note that T_i has a positive degree w.r.t. X_i, for $i = 1, 2$. The pair $\{T_1, T_2\}$ is a *regular chain* if $\mathrm{lc}(T_2, X_2)$, the leading coefficient of T_2 w.r.t. X_2, is invertible modulo T_1. By definition, the set $\{T_1\}$ is also a regular chain.

For simplicity, we will require T_1 to be squarefree, which has the following benefit: the residue class ring $\mathbb{L} = \mathbb{K}[X_1]/\langle T_1 \rangle$ is a direct product of fields. For instance, with $T_1 = X_1(X_1 + 1)$, we have:

$$\begin{array}{rcl} \mathbb{K}[X_1]/\langle T_1 \rangle & \simeq & \mathbb{K}[X_1]/\langle X_1 \rangle \oplus \mathbb{K}[X_1]/\langle X_1 + 1 \rangle \\ & \simeq & \mathbb{K} \oplus \mathbb{K}. \end{array}$$

Let $F_1, F_2, G \in \mathbb{K}[X_1X_2]$. We say G is a *regular GCD* of F_1, F_2 modulo T_1 if the following conditions hold:

(i) $\mathrm{lc}(G, X_2)$ is invertible modulo T_1,

(ii) there exist $A_1, A_2 \in \mathbb{K}[X_1, X_2]$ such that $G \equiv A_1 f_1 + A_2 f_2 \mod T_1$,

(iii) if $\deg(G, X_2) > 0$ then G divides F_1 and F_2 in $\mathbb{L}[X_2]$.

The polynomial F_1, F_2 may not have a regular GCD in the previous sense. However the following holds.

Proposition 1 *There exists polynomials $A_1, \ldots, A_e$ in $\mathbb{K}[X_1]$ and polynomials $B_1, \ldots, B_e$ in $\mathbb{K}[X_1, X_2]$ such that the following properties hold:*

- *the product $A_1 \cdots A_e$ equals T_1,*
- *for all $1 \leq i \leq e$, the polynomials B_i is a regular GCD of F_1, F_2 modulo A_i.*

The sequence $(A_1, B_1), \ldots, (A_e, B_e)$ is called a GCD sequence *of F_1 and F_2 modulo T_1.*

Consider for instance $T_1 = X_1(X_1 + 1)$, $F_1 = X_1X_2 + (X_1+1)(X_2+1)$ and $F_2 = X_1(X_2+1)+(X_1+1)(X_2+1)$. Then $(X_1, X_2 + 1), (X_1 + 1, 1)$ is a GCD sequence of F_1 and F_2 modulo T_1.

3.2 Algorithm

Recall that we aim at computing the set $V(F_1, F_2)$ of the common roots of F_1 and F_2 over $\overline{\mathbb{K}}$. For simplicity, we assume that both F_1 and F_2 have a positive degree w.r.t. X_2; we define $h_1 = \mathrm{lc}(f_1, X_2)$, $h_2 = \mathrm{lc}(f_2, X_2)$ and $h = \gcd(h_1, h2)$. Recall that R_1 denotes the resultant of F_1 and F_2 w.r.t. X_2. It is well-known that h divides R_1. Thus, we define $R_1{}'$ to be the quotient of the squarefree part of R_1 by the squarefree part of h. Our algorithm relies on the following observation.

Theorem 1 *Assume that $V(F_1, F_2)$ is finite and not empty. Then $R_1{}'$ is not constant. Moreover, for any any GCD sequence $(A_1, B_1), \ldots, (A_e, B_e)$ of F_1 and F_2 modulo $R_1{}'$, we have*

$$V(F_1, F_2) = \bigcup_{i=1}^{i=e} V(A_i, B_i) \cup V(h, F_1, F_2). \qquad (2)$$

and for all $1 \leq i \leq e$ the polynomial B_i has a positive degree w.r.t. X_2 and thus $V(A_i, B_i)$ is not empty.

This theorem implies that the points of $V(F_1, F_2)$ which do not cancel h can be computed by means of one GCD sequence computation. This is the purpose of Algorithm 1. The entire set $V(F_1, F_2)$ is computed by Algorithm 2.

Algorithm 1

Input: *F_1, F_2 as in Theorem 1.*

Output: *$(A_1, B_1), \ldots, (A_e, B_e)$ as in Theorem 1.*

ModularGenericSolve2(F_1, F_2, h) ==
(1) **Compute** $\mathrm{src}(F_1, F_2)$
(2) **Let** $R_1{}'$ *be as in Theorem 1*
(3) i *:= 1*
(4) **while** $R_1{}' \notin \mathbb{K}$ **repeat**
(5) **Let** $S_j \in \mathrm{src}(F_1, F_2)$ *regular with $j \geq i$ minimum*
(6) **if** $\mathrm{lc}(S_j, X_2) \equiv 0 \mod R_1{}'$
then $i := i + 1$; **goto** *(5)*
(7) $G :=$ $\gcd(R_1{}', \mathrm{lc}(S_j, X_2))$
(8) **if** $G \in \mathbb{K}$
then output $(R_1{}', S_j)$; **exit**
(9) **output** $(R_1{}' \text{ quo } G, S_j)$
(10) $R_1{}' := G$; $i := i + 1$

The following comments justify Algorithm 1 and are essential in view of our implementation. In Step (1) we compute the subresultant chain of F_1, F_2 in the following *lazy fashion*:

1. $B := 2d_1d_2$ is a bound for the degree of R_1, where $d_1 = \max(\deg(F_i, X_1))$ and $d_2 = \max(\deg(F_i, X_2))$. We evaluate F_1 and F_2 at $B + 1$ different values of X_1, say $x_0, \ldots, x_B$, such that none of these specializations cancels $\mathrm{lc}(F_1, X_2)$ or $\mathrm{lc}(F_2, X_2)$.
2. For each $i = 0, \ldots, B$, we compute the subresultant chain of $F_1(X_1 = x_i, X_2)$ and $F_2(X_1 = x_i, X_2)$.
3. We interpolate the resultant R_1 and do not interpolate any other subresultants in $\mathrm{src}(F_1, F_2)$.

In Step (5) we consider S_j the regular subresultant of F_1, F_2 with minimum index j greater or equal to i. We view S_j as a "candidate GCD" of F_1, F_2 modulo $R_1{}'$ and we interpolate its leading coefficient w.r.t. X_2 only. In Step (6) we test whether $\mathrm{lc}(S, X_2)$ is null modulo $R_1{}'$; if this is the case, then it follows from the block structure property that S_j is null modulo $R_1{}'$ and we go to the next candidate. In Step (8), if $G \in \mathbb{K}$ then we have proved that S_j is a GCD of F_1, F_2 modulo $R_1{}'$; in this case we interpolate S_j completely and return the pair $(R_1{}', S_j)$. In Steps (9)-(10) $\mathrm{lc}(S_j, X_2)$ has been proved to be a zero-divisor. Since $R_1{}'$ is squarefree, we apply the *D5 Principle* and the computation splits into two branches:

1. $\mathrm{lc}(S_j, X_2)$ is invertible modulo $R_1{}' \text{ quo } G$, so we output the pair $(R_1{}' \text{ quo } G, S_j)$
2. $\mathrm{lc}(S, X_2) = 0 \mod G$; we go to the next candidate.

Algorithm 2

Input: F_1, F_2 *as in Theorem 1.*

Output: *regular chains* $(A_1, B_1), \ldots, (A_e, B_e)$ *such that* $V(F_1, F_2) = \bigcup_{i=1}^{i=e} V(A_i, B_i)$.

ModularSolve2(F_1, F_2) ==
(1) **if** $F_1 \in \mathbb{K}[X_1]$ **then** ModularSolve2$(F_1 + F_2, F_2)$
(2) **if** $F_2 \in \mathbb{K}[X_1]$ **then** ModularSolve2$(F_1, F_2 + F_1)$
(3) $h := \gcd(\mathrm{lc}(F_1, X_2), \mathrm{lc}(F_2, X_2))$
(4) $G :=$ ModularGenericSolve2(F_1, F_2, h)
(5) **if** $h = 1$ **return** G
(6) $(F_1, F_2) := (\mathrm{reductum}(F_1, X_2), \mathrm{reductum}(F_2, X_2))$
(7) $D :=$ ModularSolve2(F_1, F_2)
(8) **for** $(A(X_1), B(X_1, X_2)) \in D$ **repeat**
(9) $g := \gcd(A, h)$
(10) **if** $\deg(g, X_1) > 0$ **then** $G := G \cup \{(g, B)\}$
(11) **return** G

The following comments justify Algorithm 2. Recall that $V(F_1, F_2)$ is assumed to be non-empty and finite. Steps (1)-(2) handle the case where one input polynomial is univariate in X_1; the only motivation of the trick used here is to keep pseudo-code simple. Step (4) computes the points of $V(F_1, F_2)$ which do not cancel h. From Step (6) one computes the points of $V(F_1, F_2)$ which do cancel h, so we replace F_1, F_2 by their reductums w.r.t. X_2. In Steps (8)-(10) we filter out the solutions computed at Step (7), discarding those which do not cancel h.

3.3 Implementation

We explain in the section how Algorithms 1 and 2 are implemented in MAPLE interpreted code and based on the functions of the `modpn` library.

We start with Algorithm 1. The dominant cost is at Step (1) and it is desirable to perform this step entirely at the C level in one "function call". On the other hand the data computed at Step (1) must be accessible on the MAPLE side, in particular at Step (5). Recall that the only structured data that the C and MAPLE levels can share are arrays. Fortunately, there is a natural efficient method for implementing Step (1) under these constraints:

- We represent F_1 (resp. F_2) by a $(B+1) \times d_2$ array (or "cube") C_1 (resp. C_2) where $C_1[i, j]$ (resp. $C_2[i, j]$) is the coefficient of F_1 (resp. F_2) of X_2^i evaluated at x_j; if F_1 (resp. F_2) is given over the monomial basis of $\mathbb{K}[X_1, X_2]$, then the "cube" C_1 (resp. C_2) is obtained by fast evaluation techniques.

- For each $i = 0, \ldots, B$, the subresultant chain of $F_1(X_1 = x_i, X_2)$ and $F_2(X_1 = x_i, X_2)$ is computed and stored in an $(B+1) \times d_2 \times d_2$ array, that we call "Scube"; this array is allocated on the MAPLE side and is available at the C level without any data conversions.

- The resultant R_1 (of F_1 and F_2 w.r.t. X_2) is obtained from the "Scube" by fast interpolation techniques.

In Step (5) the "Scube" is passed to a C function which computes the index j and interpolates the leading coefficient $\mathrm{lc}(S_j, X_2)$ of S_j, the candidate GCD. Testing whether $\mathrm{lc}(S_j, X_2)$ is zero or invertible modulo $R_1{}'$ is done at the MAPLE level using the `RecDen` module. Finally, in Step (8), when $\mathrm{lc}(S_j, X_2)$ has been proved to be invertible modulo $R_1{}'$, the "Scube" is passed to a C function in order to interpolate S_j.

The implementation of Algorithm 2 is much more straightforward, since the operation ModularSolve2 consists mainly of recursive calls and calls to ModularGenericSolve2. The only place where computations take place "locally" is at Step (9) where the `RecDen` module is called for performing GCD computations.

4 Two-equation Solver and Invertibility Test

In this section, we present the two other applications used to evaluate the framework presented in Section 2. The top-level algorithms are presented in Sections 4.2 and 4.3. In Section 4.1, we specify the main subroutines on which these algorithms rely; we also include there the specifications of Algorithm 4, for convenience. As we shall see in Section 5, under certain circumstances, the data-conversions implied by the calling of these subroutines can become a bottleneck. It is, thus, important to have a good picture not only of these top-level algorithms but also of their subroutines.

In this paper, however, we aim at presenting our implementation framework and its experimental evaluation without assuming that the reader has a preliminary knowledge on triangular decomposition algorithms. To this end, the presentation of our bivariate solver in Section 3 was relatively self-contained while omitting proofs. This was made easy by the bivariate nature of this application, which allowed us to hide some abstract concepts.

Our other two applications, the *two-equation solver* and the *invertibility test* involve polynomials with an arbitrary number of variables, leading to additional algebraic difficulties. Nevertheless, we hope that the detailed and elementary discussion of Section 3 could have prepared the reader.

In Section 3.1 we have introduced the notion of a *regular chain* and that of a *regular GCD (modulo a regular chain)* for bivariate polynomials. In the sequel, we rely on "natural" generalizations of these notions: we recall them briefly and refer to [12, 3] for introductory presentation.

4.1 Subroutines

From now on, our polynomials are multivariate in the ordered variables $X_1 < \cdots < X_n$ and with coefficients in a prime field $\mathbb{K}$. Let $T = T_1(X_1), \ldots, T_n(X_1, \ldots, X_n)$ be a set of n non-constant polynomials such that, for all $i = 1 \cdots n$, the largest variable in T_i is X_i. (Such a set is called a triangular set.) The set T is a *regular chain* if, for all $i = 2 \cdots n$, the leading coefficient of T_i w.r.t. X_i is invertible modulo the ideal generated by $T_1, \ldots, T_{i-1}$; moreover, it is a *normalized regular chain* if for all $i = 1 \cdots n$, the leading coefficient of T_i w.r.t. X_i is a constant polynomial, that is, belongs to $\mathbb{K}$. Observe that a normalized regular chain is a lexicographical Gröbner basis.

In the specification of our subroutines below, we denote by T a normalized regular chain and p, q polynomials in $\mathbb{K}[X_1, \ldots, X_n]$. More details about these operation can be found in the `RegularChains` library [10] where they appear with the same names and specifications.

MainVariable(p)**:** assumes that p is non-constant and returns its largest (or main) variable.

Initial(p)**:** assumes that p is non-constant and returns its leading coefficient w.r.t. MainVariable(p).

NormalForm(p, T)**:** returns the *normal form* of p w.r.t. T (in the sense of Gröbner bases). This operation is performed at the C level of our framework; it uses the fast algorithm of [14].

Normalize(p, T)**:** returns p if $p \in \mathbb{K}$; otherwise assumes that $h :=$ Initial(p) is invertible modulo the ideal generated by T and returns NormalForm$(h^{-1}p, T)$ where h^{-1} is the inverse of h modulo T. This operation is also performed at the C level of our framework and based on [14].

RegularGcd(p, q, T)**:** assumes p, q non-constant, with same main variable v and such that either Initial(p) or Initial(q) is invertible modulo T; then returns pairs $(g_1, T^1), \ldots, (g_e, T^e)$ where $g_1, \ldots, g_e$ are non-constant polynomials and $T^1, \ldots, T^e$ are normalized regular chains, such that $V(T) = V(T^1) \cup \cdots \cup V(T^e)$ holds and such that for all $i = 1 \cdots e$ g_i is a regular GCD of p, q modulo T^i, that is, satisfies the following three properties:

- (i) the leading coefficient of g_i w.r.t. v is invertible modulo T^i,
- (ii) there exist $A_1, A_2 \in \mathbb{K}[X_1, \ldots, X_n]$ such that $g_i \equiv A_1 p + A_2 q \mod T^i$,
- (ii) if $\deg(g_i, v) > 0$ then g_i divides p and q modulo T^i.

This operation is implemented on the MAPLE side with calls to our C routines; the algorithm is very similar to Algorithm 3.

IsInvertible(p, T)**:** returns pairs $(p_1, T^1), \ldots, (p_e, T^e)$ where $p_1, \ldots, p_e$ are polynomials and $T^1, \ldots, T^e$ are normalized regular chains, such that $V(T) = V(T^1) \cup \cdots \cup V(T^e)$ holds and such that for all $i = 1 \cdots e$ the polynomial p_i is either null or invertible modulo T^i and $p \equiv p_i \mod T^i$. The algorithm and implementation of this operation are described in Section 4.3.

4.2 Two-equation Solver

Let $F_1, F_2 \in \mathbb{K}[X_1, \ldots, X_n]$ be non-constant polynomials with MainVariable$(F_1) =$ MainVariable$(F_2) = X_n$. We assume that $R_1 = \mathsf{res}(F_1, F_2, X_n)$ is non-constant. Algorithm 3 below is simply the adaptation of Algorithm 1 to the case where F_1, F_2 are n-variate polynomials instead of bivariate polynomials. The relevance of Algorithm 3 to our study is based on the following observation.

As we shall see in Section 5, the implementation of Algorithm 1 in our framework is quite successful. It is, therefore, natural to check how these results are affected when some of the parameters of Algorithm 1 are modified. A natural parameter is the number of variables. Increasing it makes some routine calls more expensive and could raise some overheads.

In broad terms, Algorithm 3 computes the "generic solutions" of F_1, F_2. Formally speaking, it computes regular chains $T, \ldots, T^e$ such that we have

$$V(F_1, F_2) = \overline{W(T^1)} \cup \cdots \cup \overline{W(T^e)} \cup V(F_1, F_2, h_1 h_2) \quad (3)$$

where $h_1 h_2$ is the product Initial(F_1)Initial(F_2) and where $\overline{W(T^i)}$ denotes the Zariski closure of the quasi-component of T^i. It is out of the scope of this paper to expand on the theoretical background of Algorithm 3; this can be found in [17, 12]. Instead, as mentioned above, our goal is to measure how Algorithm 1 scales when the number of variable increases.

Algorithm 3

Input: $F_1, F_2 \in \mathbb{K}[X_1, \ldots, X_n]$ *with* $\deg(F_1, X_n) > 0, \deg(F_2, X_n) > 0$ *and* $\mathsf{res}(F_1, F_2, X_n) \notin \mathbb{K}$.

Output: $T^1 = (A_1, B_1), \ldots, T^e = (A_e, B_e)$ *as in (3).*

ModularGenericSolveN(F_1, F_2) ==
(1) **Compute** src(F_1, F_2)*;* R_1 := $\mathsf{res}(F_1, F_2, X_n)$
h := gcd(Initial(F_1), Initial(F_2))
(2) ${R_1}'$:= squarefreePart(R_1) quo squarefreePart(h)
v := MainVariable(R_1)*;*

$R_1{}'$:= primitivePart(R_1, v)
(3) i := *1*
(4) **while** $\deg(R_1{}', v) > 0$ **repeat**
(5) **Let** $S_j \in \text{src}(F_1, F_2)$ *regular with* $j \geq i$ *minimum*
(6) **if** $\text{lc}(S_j, X_n) \equiv 0 \mod R_1{}'$
then $i := i + 1$*;* **goto** *(5)*
(7) $G := \gcd(R_1{}', \text{lc}(S_j, X_2))$
(8) **if** $\deg(G, v) = 0$
then output $(R_1{}', S_j)$*;* **exit**
(9) **output** $(R_1{}' \text{ quo } G, S_j)$
(10) $R_1{}' := G$*;* $i := i + 1$

The implementation plan of Algorithm 3 is exactly the same as that of Algorithm 1. In particular, the computations of squarefree parts, primitive parts and the GCDs at Steps (1) and (7) are performed on the MAPLE side, whereas the subresultant chain $\text{src}(F_1, F_2)$ is computed on the C side. In the complexity analysis of Algorithm 3 (see [12]) the dominant cost is given by $\text{src}(F_1, F_2)$ and a natural question is whether this is verified experimentally. If this is the case, this will be a positive point for our implementation framework.

4.3 Invertibility Test

Invertibility test modulo a regular chain is a fundamental operation in algorithms computing triangular decompositions. The precise specification of this operation has been given in Section 4.1. In broad terms, for a regular chain $T = T_1(X_1), \ldots T_n(X_1, \ldots, X_n)$ and a polynomial p the call IsInvertible(p, T) "separates" the points of $V(T)$ that cancel p from those which do not. The output is a list of pairs $(p_1, , T^1), \ldots, (p_e, T^e)$ where $p_1, \ldots, p_e$ are polynomials and $T^1, \ldots, T^e$ are normalized regular chains: the points of $V(T)$ which cancel p are given by the T^i's such that p_i is null.

Algorithm 4 is in the spirit of those in [18, 17] implementing this invertibility test. However, it offers more opportunities for using modular methods and fast polynomial arithmetic. The trick is based on the following result (Theorem 1 in [3]): the polynomial p is invertible modulo T if and only if the iterated resultant of p w.r.t. T is non-zero.

Iterated resultants can be computed efficiently by evaluation and interpolation, following the same implementation techniques as those of Algorithm 1. Our implementation of Algorithm 4 employs this strategy. In particular the resultant r (computed at Step (4)) and the regular GCDs (g, D) (computed at Step (7)) are obtained from the same "Scube".

The calls NormalForm(p, T) (Step (1)) Normalize(g, D) (Step (8)) and NormalForm$(\text{quo}(T_v, g), D)$ (Step (10)) are performed on the C side: they require the conversions of regular chains encoded by MAPLE polynomials to regular chains encoded by C "cube" polynomials.

If the call RegularGcd(p, T_v, C) (Step (7)) outputs many cases, that is, if computations split in many branches, these conversions could become a bottleneck as we shall see in Section 5.

Algorithm 4

Input: T *a normalized regular chain and* p *a polynomial, both in* $\mathbb{K}[X_1, \ldots, X_n]$.

Output: *See specification in Section 4.1.*

IsInvertible(p, T) ==
(1) p := NormalForm(p, T)
(2) **if** $p \in \mathbb{K}$ **then return** $[p, T]$
(3) v := MainVariable(p)
(4) r := res(p, T_v, v)
(5) **for** $(q, C) \in$ IsInvertible(r, T_v) **repeat**
(6) *if* $q \neq 0$ **then output** $[p, C \cup T_v \cup T_{>v}]$
(7) **else for** $(g, D) \in$ RegularGcd(p, T_v, C) **repeat**
(8) g := Normalize(g, D)
(9) **output** $[0, D \cup g \cup T_{>v}]$
(10) q := NormalForm$(\text{quo}(T_v, g), D)$
(11) **if** $\deg(q, v) \neq 0$ **then**
output IsInvertible$(p, D \cup q \cup T_{>v})$

5. Experiments

We discuss here the last questions from Section 2: Can our implementation based on the above strategy outperform other highly efficient systems? Does the performance comply with the theoretical complexity?

Our answer for the first one is "yes, if the application is well suited to our framework". As shown below, we have improved the performance of triangular decompositions in MAPLE; on the example of the invertibility test, our code is competitive with MAGMA and often outperforms it. The answer to the second one is "yes" as well, even though there are interferences due to the data conversion and other overheads.

We give two kinds of data. First, we compare the operations we have implemented with their existing counterparts in MAPLE or MAGMA: we give details for the invertibility test. Second, we profile our algorithms to determine for which kind of computations our framework is best suited. Besides invertibility, we will then discuss our two other operations – the bivariate and two-equation solvers. In all examples, the base field is $\mathbb{Z}/p\mathbb{Z}$, where p is a large machine-word size FFT prime. In the following profiling samples, we just calculate the MAPLE conversion time. The converters operating at the C level are fairly efficient; their computation time is negligible.

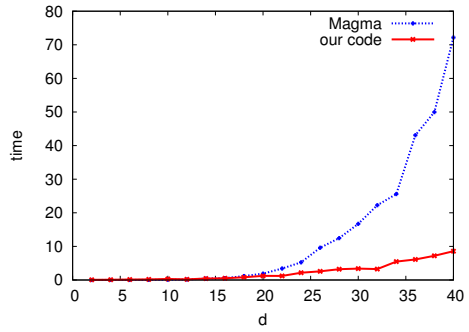

Figure 1. Bivariate case: timings, $p = 0.98$.

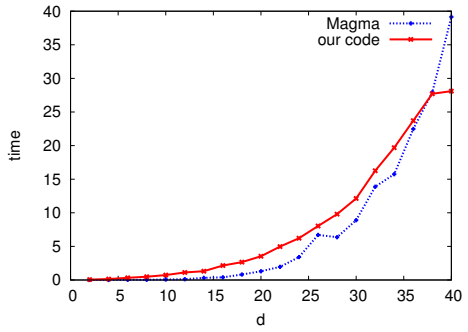

Figure 2. Bivariate case: timings, $p = 0.5$.

5.1 Invertibility Test

We start with the operation IsInvertible. Designing good test suites for this algorithms is not easy: one of the main reasons for the high technicality of these algorithms is that various kinds of degeneracies need to be handled. Using random systems, one typically does not meet such degeneracies: a random polynomial is invertible modulo a random regular chain. Hence, if we want our test suite to address more than the "generic" case of our algorithms, the examples must be constructed ad-hoc.

Here, we report on such examples for bivariate and trivariate systems. We construct our regular chain T by Chinese Remaindering, starting from smaller regular chains $T^{(i)}$ of degree 1 or 2. Then, we interpolate a function f from its values $f^{(i)} = f \bmod T^{(i)}$, these values being chosen at random. The probability p that $f^{(i)} \neq 0$ is a parameter of our construction. We generated families of examples with $p = 0.5$, for which we expect that the invertibility test of f will generate a large number of splittings. Other families have $p = 0.98$, for which few splittings should occur.

The bivariate case. Figure 1 gives results for bivariate systems with $p = 0.98$ and $d = d_1 = d_2$ in abscissa. We compare our implementation with MAGMA's counterpart, that relies on the functions `TriangularDecomposition` and `Saturation` (in general, when using MAGMA, we always choose the fastest available solution). We also tested the case $p = 0.5$ in Figure 2. Figure 3 profiles the percentage of the conversion time w.r.t. the total computation time, for the same set of samples. With $p = 0.98$, IsInvertible spends less time on conversions (around 60%) and has fewer calls to the MAPLE operations than with $p = 0.5$ (the conversion ratio with $p = 0.5$ reaches up to 83%).

The trivariate case. Table 5.1 uses trivariate polynomials as the input for IsInvertible, with $p = 0.98$; Table 5.1 has $p = 0.5$. Figure 4 profiles the conversion time spent on these samples. The conversion time increases dramatically along the input size. For the largest example, the conversion time reaches 85% of the total computation time. More than

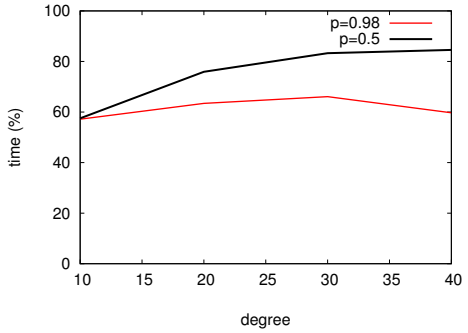

Figure 3. Bivariate case: profiling.

5% of the time is spent on other MAPLE computations, so that the real C computation costs less than 5%. We also provide the timing of the operation REGULARIZE from the MAPLE `RegularChains` library. The pure MAPLE code, with no fast arithmetic, is several hundred times slower than our implementation.

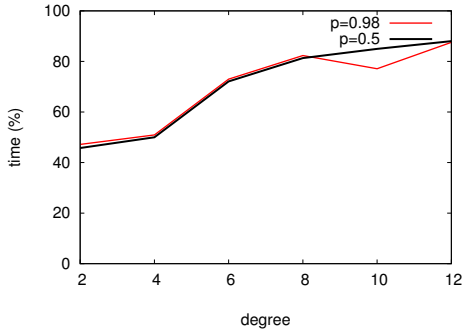

Figure 4. Trivariate case: profiling.

The 5 variable case. We performed further tests between the MAPLE REGULARIZE operation and our IsInvertible function, using random dense polynomials in 5 variables. IsInvertible is significantly faster than REGULARIZE; the speedup reaches a factor of 300. Similar experiments with sparse polynomials give a speed-up of 100.

Table 1. Trivariate case: timings, $p = 0.98$.

$d_1 d_2$	d_3	MAGMA	MAPLE	
			REGULARIZE	IsInvertible
4	3	0.000	1.199	0.091
12	6	0.020	6.569	0.281
24	9	0.050	24.312	0.509
40	12	0.170	73.905	1.293
60	15	0.550	172.931	1.637
84	18	1.990	450.377	5.581
112	21	5.130	871.280	9.490
144	24	12.830	1956.728	12.624
180	27	30.510	3621.394	23.564
220	30	62.180	6457.538	32.675
264	33	129.900	7980.241	89.184

Table 2. Trivariate case: timings, $p = 0.5$.

$d_1 d_2$	d_3	MAGMA	MAPLE	
			REGULARIZE	IsInvertible
4	3	0.010	0.773	0.199
12	6	0.020	4.568	0.531
24	9	0.040	17.663	1.082
40	12	0.150	47.767	2.410
60	15	0.480	126.629	5.023
84	18	1.690	284.697	10.405
112	21	4.460	632.539	19.783
144	24	10.960	1255.980	42.487
180	27	26.070	2328.012	69.736
220	30	58.700	4170.468	109.667
264	33	106.140	7605.915	191.514

5.2 Other Operations

We conclude with profiling information for our other applications. The differences between these algorithms have noticeable consequences regarding profiling time.

Bivariate solver. For this algorithm, there is no risk of data duplication. The amount of data conversion is bounded by the size of the input plus the size of the output; hence we expect that data conversions cannot be a bottleneck. Third, the calls to MAPLE interpreted code simply perform univariate operations, thus we do not expect them to become a bottleneck either.

Table 5.2 confirms this expectation, by giving the profiling information for this algorithm. The input system is dense and contains 400 solutions. The computation using the `RecDen` package costs 49% of the total computation time. The C level subresultant chain computation spends around 34%, and the conversion time is less than 11%. With larger input systems, the conversion time reduces. For systems with 2,500 and 10,000 solutions, the C computation takes about 40% of the time; `RecDen` computations takes roughly 50%; other MAPLE functions take 5% and the conversion time is less than 5%.

Table 3. Bivariate solver: profiling, $p = 0.98$.

Operation	calls	time	time (%)
Subresultant chain	1	0.238	33.85
`Recden`	41	0.344	48.93
Conversions	17	0.076	10.81

The profiling information in Figure 5 also concerns the Bivariate solver; there, the sample input intends to generate many splittings (we take $p = 0.5$, as in the examples in the previous subsection). The conversion time slowly increases but does not become the bottleneck (28% to 38%).

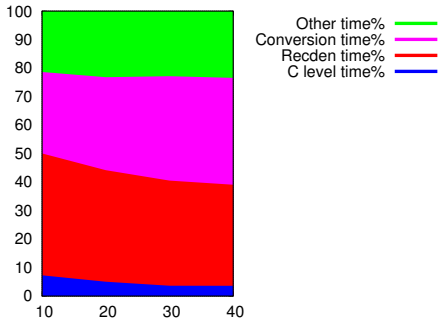

Figure 5. Bivariate solver: profiling, $p = 0.5$.

Two-equation solver. This algorithm has properties similar to the Bivariate Solver, except that the calls to interpreted code can be expensive since it involves multivariate

arithmetic. Hence, we expect that the overhead of conversion is quite limited. Indeed, in Table 5.2, N is the number of variables and d_1, d_2 are the degrees of T_1, T_2 respectively 3. The C level computation is the major factor of the total computation time; it reaches 91% in case $N = 4,\ d_1 = 5,\ d_2 = 5$.

Table 4. Two-equation solver: profiling.

N	d_1	d_2	C (%)	MAPLE (%)	Conversion (%)
3	5	5	56.47	12.96	30.57
4	5	5	91.54	2.64	5.82
8	2	2	83.67	8.02	8.31

6 Conclusion

The answers to our main questions are mostly positive: we obtained large performance improvements over existing MAPLE implementations, and often perform better than MAGMA, a reference regarding high performace.

Still, some triangular decomposition algorithms are not perfectly suited to our framework. For instance, we implemented the efficiency-critical operations of ISINVERTIBLE in C, but the main algorithm itself in MAPLE. Still, this algorithm may generate large amount of "external" calls to the C functions, so the data conversion between MAPLE and becomes dominant in timings. For this kind of algorithms, we suggest either to implement them in C or tune the algorithmic structure to avoid intensive data conversion at the MAPLE level; we are working on both directions.

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools (2nd Edition)*. Addison-Wesley, 2006.

[2] E. Becker, T. Mora, M. G. Marinari, and C. Traverso. The shape of the Shape Lemma. In *ISSAC'94*, pages 129–133. ACM, 1994.

[3] C. Chen, F. Lemaire, O. Golubitsky, M. Moreno Maza, and W. Pan. *Comprehensive Triangular Decomposition*, volume 4770 of *Lecture Notes in Computer Science*, pages 73–101. Springer Verlag, 2007.

[4] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In *ISSAC'05*, pages 108–115. ACM Press, 2005.

[5] L. Ducos. Optimizations of the subresultant algorithm. *Journal of Pure and Applied Algebra*, 145:149–163, 2000.

[6] 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.

[7] J. van der Hoeven. The Truncated Fourier Transform and applications. In *ISSAC'04*, pages 290–296. ACM, 2004.

[8] F. Lemaire, M. Moreno Maza, and Y. Xie. The TRIADE library in MAPLE, 2004. With contributions of W. Wu and É. Schost.

[9] F. Lemaire, M. Moreno Maza, and Y. Xie. The `RegularChains` library. In *Maple 10*, Maplesoft, Canada, 2005. Refereed software.

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

[11] 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.

[12] X. Li, M. Moreno Maza, and R. Rasheed. Fast arithmetic and modular techniques for triangular decompositions, 2008.

[13] X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. The `modpn` library: Bringing fast polynomial arithmetic into MAPLE. In *MICA'08*, 2008.

[14] 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.

[15] X. Li, M. Moreno Maza, and É. Schost. On the virtues of generic programming for symbolic computation. In *ICCS'07*, volume 4488 of *Lecture Notes in Computer Science*, pages 251–258. Springer, 2007.

[16] B. Mishra. *Algorithmic Algebra*. Springer-Verlag, New Yor, 1993.

[17] 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`.

[18] M. Moreno Maza and R. Rioboo. Polynomial gcd computations over towers of algebraic extensions. In *Proc. AAECC-11*, pages 365–382. Springer, 1995.

[19] R. Rasheed. Modular methods for solving polynomial systems, 2007. University of Western Ontario.

[20] É. Schost. Complexity results for triangular sets. *J. Symb. Comp.*, 36(3-4):555–594, 2003.

[21] É. Schost. Computing parametric geometric resolutions. *Appl. Algebra Engrg. Comm. Comput.*, 13(5):349–393, 2003.

[22] V. Shoup. *The Number Theory Library*. 1996–2008. `http://www.shoup.net/ntl`.

[23] The Computational Algebra Group in the School of Mathematics and Statistics at the University of Sydney. *The MAGMA Computational Algebra System for Algebra, Number Theory and Geometry*. `http://magma.maths.usyd.edu.au/magma/`.

[24] C. Yap. *Fundamental Problems in Algorithmic Algebra*. Princeton University Press, 1993.