The \texttt{modpn} library: Bringing fast polynomial arithmetic into \textsc{Maple}

Xin Li, Marc Moreno Maza, Raqeeb Rasheed, Éric Schost

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

\textbf{A R T I C L E \ I N F O}

\textbf{Article history:}
Available online 1 March 2011

In honour of Keith Geddes on his 60th birthday

\textbf{Keywords:}
Fast polynomial arithmetic
Interoperability between C and Maple
High-performance computing
Polynomial system solving

\textbf{A B S T R A C T}

We investigate the integration of C implementation of fast arithmetic operations into \textsc{Maple}, focusing on triangular decomposition algorithms. We show substantial improvements over existing \textsc{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.

© 2011 Published by Elsevier Ltd

doi:10.1016/j.jsc.2010.08.016

\section{1. Introduction}

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

Computer algebra systems and libraries born in the 90s, such as \textsc{Magma} and \texttt{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 algorithm. This increased in a spectacular manner the range of solvable problems. Meanwhile, \textsc{Axiom} and \textsc{Maple} remain highly attractive, none the least for their programming environments and because of their user communities.
In previous work (Filat et al., 2006; Li and Moreno Maza, 2006; Li et al., 2007b) we have investigated the integration of asymptotically fast arithmetic operations into AXIOM. Since AXIOM is based today on GNU Common Lisp (GCL), we realized 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-ups, it was sufficient to extend the existing AXIOM polynomial domain constructors with our fast routines (for univariate multiplication, division, GCD, etc.) and call them in the existing generic packages (for instance, for univariate squarefree factorization); see Li et al. (2007b) 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; within MAGMA, users can define packages, where functions are compiled in MAGMA internal code, but not in C.

In the present paper, we investigate the integration of fast arithmetic operations implemented in C into MAPLE. Most of the 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 (Li et al., 2007a). This question is made more difficult by the following factors.

First, 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 objects must be performed on the MAPLE side, 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 only a MAPLE connection mechanism capable of calling our C routines will not be sufficient to speed up all MAPLE libraries using polynomial arithmetic: clearly, a high-level MAPLE code needs to be rewritten to rely on this connection mechanism.

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

(Q1) To what extent can triangular decomposition algorithms (implemented in the MAPLE RegularChains library) 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 performance estimated by complexity analysis?

This paper attempts to provide elements of answers to these questions. In Section 2, we describe the framework that we designed in this programming environment: modpn, a C-MAPLE library dedicated to fast arithmetic for multivariate polynomials over finite fields. In Sections 3 and 4 we present the following applications, that were implemented in this framework (definitions are given in the later sections):

**Bivariate solver.** This application takes as input two polynomials $F_1, F_2$ in two variables $X_1 < X_2$, 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 $F_1, F_2$ with respect to $X_n$ and a regular GCD of $F_1, F_2$ modulo (the primitive part of) $R_1$. This is an extension of the previous question to the case of 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 $V(T)$ that cancel $p$ from those which do not, and outputs 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 with respect to a zero-dimensional regular chain, etc. These three applications are actually part of a new module of the RegularChains library, called FastArithmeticTools, which provides operations on regular chains (in prime characteristic and mainly in dimensions zero or one) based on modular methods and fast polynomial arithmetic. Therefore, these three applications are 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, they challenge our framework in different ways. Our experimental results are reported and analyzed in Section 5.

2. A compiled–interpreted programming environment

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

Relying on asymptotically fast algorithms, the C level routines are highly optimized. The core operations are fast operations with triangular sets (multiplication/inversion as in Li et al. (2007a) and lifting techniques (Schost, 2003)), GCDs, resultants and fast interpolation. 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 (the speed ratio may reach several orders of magnitude, as reported later).

We use multiple polynomial data encodings at each level, showed in Fig. 1. The Maple-Dag and Maple-Recursive-Dense polynomials are MAPLE built-in types; the C-Dag, C-Cube and C-2-Vector polynomials were written by us in C. Each encoding is adapted to certain applications; we switch between different representations at run-time. Thus, this section discusses some of the 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 the existing packages or develop our own ones, etc.

Of the four questions mentioned in the Introduction, we discuss the first two here: to what 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 the best performance. All operations are based on asymptotically fast algorithms rooted at Fast Fourier Transform (FFT) and its variant Truncated Fourier Transform (TFT) (van der Hoeven, 2004). These operations are optimized with respect to crucial features of hardware architecture: memory hierarchy, instruction pipe-lining, and vector instructions. As reported in Li et al. (2007a), Li and Moreno Maza (2006) and Filatei et al. (2006), our C library often outperforms the best known implementations such as Magma and NTL (Bosma et al., 1997; Shoup, 1996–2008).

The C code is dedicated to regular chain operations modulo a machine size prime number, mainly in dimension zero. Such computations typically generate dense polynomials; thus, we use multidimensional arrays as the canonical encoding, and we call them C-Cubes (since all partial degrees are bounded in advance). This encoding is the most appropriate for FFT-based multiplication, inversion modulo a zero-dimensional regular chain, interpolation, etc. Besides, we can pre-allocate the working buffer and use in-place operations whenever applicable. Tracing coefficients and degrees also becomes trivial.
In addition to the C-Cube encoding, we use another polynomial encoding at the C level: to apply triangular lifting algorithms (Schost, 2003; Dahan et al., 2005), we use a Directed Acyclic Graph representation, that we call C-Dag. By setting “smart” flags in the nodes of these Dags, we can track the information of visibility and liveness in constant time. We implemented a third data structure at C level, the C-2-Vector encoding, which is meant to facilitate the conversion between C-Cubes and MAPLE's RecDen (recursive dense) polynomials.

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 RegularChains library (Lemaire et al., 2005).

At the MAPLE level, we use two types of polynomials: MAPLE Dags and RecDen polynomials. Dags are the default data representation for MAPLE polynomials; the RegularChains library uses them uniformly. Thus, in order to access the functionalities of this library and support the development of more efficient algorithms there, we need to use MAPLE Dags. In addition, we rely on RecDen when implementing dense polynomial algorithms in MAPLE: RecDen allows one to handle dense polynomials in a recursive manner (using one recursion level per variable); operations modulo a zero-dimensional regular chain are essentially dense methods, so that using RecDen is appropriate 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 code provides a large speed-up to the MAPLE code; this is reported in Section 5.

2.3. MAPLE and C cooperation

For 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, we face the issue of data conversion.

This is 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 structures 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: designing the algorithms with the objective to reduce the number of conversions, and implementing efficient converters to minimize the time of each unit conversion.

The frequency of 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 to 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 C-2-Vector 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 is not used in any “algebraic” algorithm: it is specifically designed for facilitating the data conversion from C-Cube to RecDen encoding. The C-2-Vector encoding has the same recursive structure as RecDen, so the mapping is easy between these two. Besides, the C-2-Vector encoding uses flattened polynomial tree structures, which are convenient to pass from C to Maple.

3. Bivariate solver

The first application we used to evaluate our 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 \( K \). We assume that \( K \) is perfect; in our experimentation \( K \) is a prime field whose characteristic is a machine word size prime.

We rely on an algorithm introduced in Rasheed (2007) and based on the following well-known fact (Becker et al., 1994). The common roots of \( F_1 \) and \( F_2 \) over an algebraic closure \( \bar{K} \) of \( K \) are “likely” to be described by the common roots of a system with a triangular shape:

\[
\begin{align*}
T_1(X_1) &= 0 \\
T_2(X_1, X_2) &= 0,
\end{align*}
\]

such that the leading coefficient of \( T_2 \) with respect to \( X_2 \) is invertible modulo \( T_1 \); moreover, the degree of \( T_2 \) with respect to \( X_2 \) is “likely” to be 1. For instance, the system

\[
\begin{align*}
X_1^2 + X_2 + 1 &= 0 \\
X_1 + X_2^2 + 1 &= 0,
\end{align*}
\]

is solved by the triangular system

\[
\begin{align*}
X_1^4 + 2X_2^2 + X_1 + 2 &= 0 \\
X_2 + X_1^2 + 1 &= 0.
\end{align*}
\]

In general, though, more complex situations can arise, where more than one triangular system is needed. The goal of this section is to show that this algorithm can easily be implemented in our 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 GCD’s modulo regular chains. Classical textbooks for the former are Yap (1993) and Mishra (1993) whereas the latter was introduced in Moreno Maza and Rioboo (1995).

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 Ducos (2000)).

Consider more general rings, such as \( \mathbb{K}[X_1, X_2] \). Assume that \( F_1, F_2 \) are non-constant polynomials with \( \text{deg}(F_1, X_2) \geq \text{deg}(F_2, X_2) \), and \( \text{deg}(F_2, X_2) = q \). The polynomials computed by the subresultant algorithm form a sequence, called the subresultant chain of \( F_1 \) and \( F_2 \) and denoted by \( \text{src}(F_1, F_2) \). This sequence consists of \( q + 1 \) polynomials, starting at \( \text{lct}(F_2, X_2)^j F_2 \), with \( \delta = \text{deg}(F_1, X_2) - \text{deg}(F_2, X_2) \), and ending at \( R_1 := \text{res}(F_1, F_2) \), the resultant of \( F_1 \) by \( F_2 \) with respect to \( X_2 \). We write this sequence \( S_0, \ldots, S_q \) 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 to be regular when \( \text{deg}(S_j, X_2) = j \) 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 \( j < \text{deg}(F_2, X_2) - 1 \), 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 \( S_i \) is null.

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

\[
\Phi(S_j(F_1, F_2)) = \Phi(a)^{\delta - k} \Phi(F_1, \Phi(F_2)),
\]

where \( q = \text{deg}(F_2, X_2) \) and \( k = \text{deg}(\Phi(F_2), X_2) \).

Regular GCD’s 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 \( T_1 \) has a positive degree in \( X_i \), for \( i = 1, 2 \). The pair \( \{T_1, T_2\} \) is a regular chain if the leading coefficient \( \text{lct}(T_2, X_2) \) of \( T_2 \) in \( X_2 \) is invertible modulo \( T_1 \). By definition, the set \( \{T_1\} \) is also a regular chain. For simplicity, we will require \( T_2 \) to be squarefree, which has the following benefit: the residue class ring \( \mathbb{L} = \mathbb{K}[X_1] / (T_1) \) is a direct product of fields.

Let \( F_1, F_2, G \in \mathbb{K}[X_1, X_2] \) be non-zero. We say \( G \) is a regular GCD of \( F_1, F_2 \) modulo \( T_1 \) if the following conditions hold:

1. \( \text{lct}(G, X_2) \) is invertible modulo \( T_1 \),
2. there exist \( A_1, A_2 \in \mathbb{K}[X_1, X_2] \) such that \( G \equiv A_1 f_1 + A_2 f_2 \mod T_1 \),
3. if \( \text{deg}(G, X_2) > 0 \) then \( G \) divides \( F_1 \) and \( F_2 \) in \( \mathbb{L}[X_2] \).

The polynomials \( F_1, F_2 \) may not have a regular GCD in the previous sense. However, the following holds.

Proposition 1. There exist 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 polynomial \( B_i \) 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 \( T_1 = X_1(X_1 + 1) \),

\[
F_1 = X_1 X_2 + (X_1 + 1)(X_2 + 1) \quad \text{and} \quad F_2 = X_1(X_2 + 1) + (X_1 + 1)(X_2 + 1).
\]

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 in $X_2$; we define $h_1 = \text{lcf}(f_1, X_2)$, $h_2 = \text{lcf}(f_2, X_2)$ and $h = \gcd(h_1, h_2)$. Recall also that $R_1$ denotes the resultant of $F_1$ and $F_2$ in $X_2$. Since $h$ divides $R_1$, 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, which proof is routine with the framework of regular chains, but for which we are not aware of a reference.

**Theorem 1.** Assume that $V(F_1, F_2)$ is finite and not empty. Then $R'_1$ is not constant. Moreover, for any GCD sequence $(A_1, B_1), \ldots, (A_e, B_e)$ of $F_1$ and $F_2$ modulo $R'_1$, we have

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

and for all $1 \leq i \leq e$ the polynomial $B_i$ has a positive degree in $X_2$ 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($F_1, F_2, h$) $\Rightarrow$

1. **Compute** 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** $S_j \in \text{src}(F_1, F_2)$ with $j \geq i$ minimum
6. **if** $\text{lcf}(S_j, X_2) \equiv 0 \mod R'_1$
7. **then** $i := i + 1$; **goto** (5)
8. **if** $G \in \mathbb{K}$
9. **then output** $(R'_1, S_j)$; **exit**
10. **output** $(R'_1 \, \text{quo} \, G, S_j)$
11. $R'_1 := G$; $i := i + 1$

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 $d_1 = \max(\deg(F_1, X_1))$ and $d_2 = \max(\deg(F_1, X_2))$. 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 $\text{lcf}(F_1, X_2)$ or $\text{lcf}(F_2, X_2)$.
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 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 than 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 with respect to $X_2$ only. In Step (6) we test whether $\text{lcf}(S_j, X_2)$ 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) $\text{lcf}(S_j, X_2)$ 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:
Theresultant
F
\text{for each}\\\\\text{We}\\\\\text{Algorithm}
\text{Algorithm 2.}
\text{Input: } F_1, F_2 \text{ as in Theorem 1.}
\text{Output: } \text{regular chains } (A_1, B_1), \ldots, (A_e, B_e) \text{ such that } V(F_1, F_2) = \bigcup_{i=1}^e V(A_i, B_i).
\text{ModularSolve2}(F_1, F_2) ==
(1) \text{if } F_1 \in \mathbb{K}[X_1] \text{ then return } \text{ModularSolve2}(F_1 + F_2, F_2)
(2) \text{if } F_2 \in \mathbb{K}[X_1] \text{ then return } \text{ModularSolve2}(F_1, F_1 + F_2)
(3) h := \gcd(\text{lc}(F_1, X_2), \text{lc}(F_2, X_2))
(4) G := \text{ModularGenericSolve2}(F_1, F_2, h)
(5) \text{if } h = 1 \text{ return } G
(6) (F_1, F_2) := (\text{reductum}(F_1, X_2), \text{reductum}(F_2, X_2))
(7) D := \text{ModularSolve2}(F_1, F_2)
(8) \text{for } (A(X_1), B(X_1, X_2)) \in D \text{ repeat}
(9) \quad g := \gcd(A, h)
(10) \quad \text{if } \deg(g, X_1) > 0 \text{ then } G := G \cup \{(g, B)\}
(11) \text{return } G
The following comments justify \textbf{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 with respect to \(X_2\). 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 now how \textbf{Algorithms 1} and \textbf{2} are implemented in \textsc{Maple} interpreted code, using the functions of the \texttt{modpn} library.

We start with \textbf{Algorithm 1}. The dominant cost is at Step (1) and it is desirable to perform this step entirely at the \textsc{C} level in one “function call”. On the other hand, the data computed at Step (1) must be accessible on the \textsc{Maple} side, in particular at Step (5). Recall that the only structured data that the \textsc{C} and \textsc{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 \((B+1) \times d_x\) 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 \(X_1^j\) evaluated at \(X_1^i\); 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 a \((B+1) \times d_x \times d_x\) array, that we call “Scube”; this array is allocated on the \textsc{Maple} side and is available at the \textsc{C} level without any data conversions.
- The resultant \(R_1\) (of \(F_1\) and \(F_2\) in \(X_2\)) is obtained from the “Scube” by fast interpolation techniques.

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

The implementation of \textbf{Algorithm 2} is much more straightforward, since the operation \texttt{ModularSolve2} consists mainly of recursive calls and calls to \texttt{ModularGenericSolve2}. The only place where computations take place “locally” is at Step (9) where the \textsc{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 of Section 2. In Section 4.1, we specify the main subroutines on which these algorithms rely; we also include there the specifications of the invertibility test for convenience. The top-level algorithms are presented in Sections 4.2 and 4.3.

As we shall see in Section 5, under certain circumstances, the data conversions implied by the calling of subroutines can become a bottleneck. It is thus useful to have a clear picture of these subroutines.

In this paper, however, we do not assume 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. In this section, we deal with polynomials with an arbitrary number of variables. 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 what follows, we rely on “natural” generalizations of these notions: we recall them briefly and refer to Aubry et al. (1999) and Chen et al. (2007) for introductory presentations.

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 $i = 1 \ldots n$, 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 $i = 2, \ldots, n$, the leading coefficient of $T_i$ with respect to $X_1$ is invertible modulo the ideal generated by $T_1, \ldots, T_{i-1}$; moreover, it is a normalized regular chain if for all $i = 1, \ldots, n$, the leading coefficient of $T_i$ with respect to $X_i$ belongs to $\mathbb{K}$. Note that we restrict ourselves here to zero-dimensional regular chains. In this setting, 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 by $p, q$ two polynomials in $\mathbb{K}[X_1, \ldots, X_n]$. More details about these operations can be found in the RegularChains library (Lemaire et al., 2005) 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 with respect to MainVariable($p$).

NormalForm($p, T$): returns the normal form of $p$ with respect to $T$ (in the sense of Gröbner bases). This operation is performed at the C level of our framework; it uses the fast algorithm of Li et al. (2007a).

Normalize($p, T$): returns $p$ if $p \in \mathbb{K}$; otherwise assumes that $h := \text{Initial}(p)$ is invertible modulo the ideal generated by $T$ and returns $\text{NormalForm}(h^{-1}p, T)$ 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 Li et al. (2007a).

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 $T$. 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 $i = 1, \ldots, e$, $g_i$ is a regular GCD of $p, q$ modulo $T^{(i)}$, that is, satisfies the following three properties:

(i) the leading coefficient of $g_i$ with respect to $v$ is invertible modulo $T^{(i)}$,
(ii) there exist $A_1, A_2 \in \mathbb{K}[X_1, \ldots, X_n]$ such that $g_i \equiv A_1p + A_2q \mod T^{(i)}$.
(iii) if $\text{deg}(g_i, v) > 0$ 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 \(i = 1, \ldots, e\), 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.

\(T_{<v}, T_v, T_{>v}\): these denote respectively the polynomials in \(T\) with main variable less than \(v\), the polynomial in \(T\) with main variable \(v\), and the polynomials in \(T\) with main variable greater than \(v\), where \(v \in \{X_1, \ldots, X_n\}\).

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 \(R_1 = \text{res}(F_1, F_2, X_n)\) 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 its parameters 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 \(T^{(1)}, \ldots, T^{(e)}\) such that we have

\[
V(F_1, F_2) = \overline{W(T^{(1)})} \cup \cdots \cup \overline{W(T^{(e)})} \cup V(F_1, F_2, h_1h_2),
\]

where \(h_1h_2\) is the product \(\text{Initial}(F_1)\text{Initial}(F_2)\) and where \(\overline{W(T^{(0)})}\) denotes the Zariski closure of the quasi-component of \(T^{(0)}\). It is out of the scope of this paper to expand on the theoretical background of Algorithm 3; this can be found in Moreno Maza (1999). Instead, as mentioned above, our goal is to measure how Algorithm 1 scales when the number of variables 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 \(\text{res}(F_1, F_2, X_n) \notin \mathbb{K}\).

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

ModularGenericSolveN\((F_1, F_2)\) ==

1. **Compute** \(\text{src}(F_1, F_2); R_1 := \text{res}(F_1, F_2, X_n)\)
   \(h := \gcd(\text{Initial}(F_1), \text{Initial}(F_2))\)
2. \(R'_1 := \text{squarefreePart}(R_1) \mod \text{squarefreePart}(h)\)
   \(v := \text{MainVariable}(R_1)\)
   \(R'_1 := \text{primitivePart}(R_1, v)\)
3. \(i := 1\)
4. **while** \(\deg(R'_1, v) > 0\) **repeat**
5. \(\text{Let } S_j \in \text{src}(F_1, F_2)\) regular with \(j \geq i\) minimum
6. \(\text{if } \text{lc}(S_j, X_n) \equiv 0 \mod R'_j\)
   **then** \(i := i + 1; \text{goto (5)}\)
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 (to be reported in another article) 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 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 \( \text{IsInvertible}(p, T) \) “separates” the points of \( V(T) \) that cancel \( p \) from those which do not. The output is a list of pairs \((p_1, T^{(1)})\), . . . , \((p_e, T^{(e)})\) 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 Moreno Maza and Rioboo (1995) and Moreno Maza (1999) 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 Chen et al., 2007): the polynomial \( p \) is invertible modulo \( T \) if and only if the iterated resultant of \( p \) with respect to \( T \) 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)), NormalForm\((\text{quo}(T_v, g), D)\) (Step (10)) and Normalize\((g, D)\) (Step (8)) are performed on the C side: they require the conversions of regular chains encoded by \textsc{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. Finally, for simplicity, we restrict Algorithm 4 to the case of (zero-dimensional) regular chains generating radical ideals.

Algorithm 4.

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

**Output:** See specification in Section 4.1.

\[ \text{IsInvertible}(p, T) = \]

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

5. Experiments

We discuss here the last two questions mentioned in the Introduction: 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 \textsc{Maple}; on the example of the invertibility test, our code is competitive with \textsc{Magma} and often outperforms it. The answer to the last question is “yes” as well, even though there are interferences due to the data conversion and other overheads.
Table 1
Generic bivariate systems: all solvers.

<table>
<thead>
<tr>
<th>$d_1$</th>
<th>$d_2$</th>
<th>MAPLE Basis Solve Triangularize ModularSolve2</th>
<th>MAGMA GB Trig</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>2</td>
<td>0.3 37 12</td>
<td>0.1 0.03 0.03</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>3 306 62</td>
<td>0.13 0.11 0.12</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>18 1028 122</td>
<td>0.16 0.32 0.32</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>27 2525 256</td>
<td>0.2 0.61 0.66</td>
</tr>
</tbody>
</table>

We give two kinds of data. First, we compare the operations we have implemented with their existing counterparts in MAPLE or MAGMA; see Sections 5.1–5.3. Second, we profile our algorithms to determine for which kind of computations our framework is best suited. For the invertibility test, this profiling information is located in Section 5.3. For the solvers, it is reported in Section 5.4. 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.

5.1. Bivariate solver

We start our comparison of bivariate system solvers using random dense, thus generic, systems. We choose partial degrees $d_1$ (in $X_1$) and $d_2$ (in $X_2$); the input polynomials have support $X^i_1X^j_2$, with $i \leq d_1$ and $j \leq d_2$, and random coefficients. Such random systems are in Shape Lemma position: no splitting occurs, and the output has the form $T_1(X_1)$, $T_2(X_1, X_2)$, where $\deg(T_1, X_1) = d_1d_2$ and $\deg(T_2, X_2) = 1$.

Table 1 is an overview of the running time of many solvers. In MAPLE, we compare the Basis and Solve commands of the Groebner package to the Triangularize command of the RegularChains package and our code, referred to as ModularSolve2. In MAGMA, we use the GroebnerBasis and TriangularDecomposition commands; the columns in the table follow this order. Gröbner bases are computed for lexicographic orders.

MAPLE uses the FGB software for Gröbner basis computations over some finite fields, written in C (Faugère, in press). However, our large Fourier base field is not handled by FGB; hence, our Basis experiments are done modulo $p' = 65521$, for which FGB can be used. This limited set of experiment already shows that our code performs quite well. To be fair, we add that for MAPLE’s Basis computation, most of the time is spent in basis conversion, which is interpreted MAPLE code: for the largest example, the FGB time was 0.97 s.

We refine these first results by comparing in Fig. 2 our solver with MAGMA’s triangular decomposition for larger degrees. It quickly appears that our code performs better; for the largest examples (having about 5700 solutions), the ratio is about 460/7.
Experimentation with non-generic, and in particular non-equiprojectable systems are reported in Fig. 3. Those examples are generated in order to enforce many splittings during the computations. (Details on the generation of our examples are actually given in the next section.) For those non-equiprojectable systems, our tests compare our solver and Magma: for the largest examples, the ratio is about 5260/80, in our favor.

5.2. Two-equation solver

We consider now the solver of Algorithm 3. For a machine word size FFT prime $p$, we consider a pair of trivariate polynomials $F_1, F_2 \in \mathbb{Z}/p\mathbb{Z}[X_1, X_2, X_3]$ of total degrees $d_1, d_2$. We compare our code for ModularGenericSolveN (Algorithm 3) to the Triangularize function of RegularChains library. In Magma there are several ways to obtain similar outputs: either by a triangular decomposition in $\mathbb{K}(X_1)[X_2, X_3]$ (triangular decompositions in MAGMA require the ideal to have dimension zero) or by computing the GCD of the input polynomials modulo their resultant (assuming that this resultant is irreducible).

Table 2 summarizes the timings (in seconds) obtained on random dense polynomials by the approaches above (in the same order). Our new code performs significantly faster than all other ones. For completeness, we add that on these examples, computing a lexicographic Gröbner basis in $\mathbb{K}[X_1, X_2, X_3]$ in Magma takes time similar to that of the triangular decomposition.

5.3. Invertibility test

We continue with the operation IsInvertible. Designing good test suites for this algorithm 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 $f^{(i)} = f \mod T^{(i)}$, these values being chosen at random. The probability $\text{prob}$ that $f^{(i)} \neq 0$ is a parameter of our construction. We generated families of examples with $\text{prob} = 0.5$, for which we expect that the invertibility test of $f$ will generate a large number of splittings. Other families have $\text{prob} = 0.98$, for which few splittings should occur.

**The bivariate case.** Fig. 4 gives results for bivariate systems with $\text{prob} = 0.98$ 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 $\text{prob} = 0.5$ in Fig. 5. Fig. 6 profiles the percentage of the conversion time with respect to the total computation time, for the same set of samples. With $\text{prob} = 0.98$, IsInvertible spends less time on conversions (around 60%) and has fewer calls to the MAPLE operations than with $\text{prob} = 0.5$ (the conversion ratio with $\text{prob} = 0.5$ reaches 83%).

**The trivariate case.** Table 3 uses trivariate polynomials as the input for IsInvertible, with $\text{prob} = 0.98$: Table 4 has $\text{prob} = 0.5$. Fig. 7 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 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.
Table 3
Trivariate case: timings, prob = 0.98.

<table>
<thead>
<tr>
<th>$d_1$</th>
<th>$d_2$</th>
<th>$d_3$</th>
<th>Magma</th>
<th>Maple Regularize</th>
<th>IsInvertible</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>0.000</td>
<td>1.199</td>
<td>0.091</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>0.020</td>
<td>6.669</td>
<td>0.281</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>9</td>
<td>0.050</td>
<td>24.312</td>
<td>0.509</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>12</td>
<td>0.170</td>
<td>73.905</td>
<td>1.293</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>15</td>
<td>0.550</td>
<td>172.931</td>
<td>1.637</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>18</td>
<td>1.990</td>
<td>450.377</td>
<td>5.581</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>21</td>
<td>5.130</td>
<td>871.280</td>
<td>9.490</td>
<td></td>
</tr>
<tr>
<td>144</td>
<td>24</td>
<td>12.830</td>
<td>1956.728</td>
<td>12.624</td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>27</td>
<td>30.510</td>
<td>3621.394</td>
<td>23.564</td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>30</td>
<td>62.180</td>
<td>6457.538</td>
<td>32.675</td>
<td></td>
</tr>
<tr>
<td>264</td>
<td>33</td>
<td>129.900</td>
<td>7980.241</td>
<td>89.184</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 7. Trivariate case: time spent in conversions.

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 4
Trivariate case: timings, prob = 0.5.

<table>
<thead>
<tr>
<th>(d_1)</th>
<th>(d_2)</th>
<th>(d_3)</th>
<th>Magma</th>
<th>Maple</th>
<th>Regularize</th>
<th>IsInvertible</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td></td>
<td>0.010</td>
<td>0.773</td>
<td>0.199</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td></td>
<td>0.020</td>
<td>4.568</td>
<td>0.531</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>9</td>
<td></td>
<td>0.040</td>
<td>17.663</td>
<td>1.082</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>12</td>
<td></td>
<td>0.150</td>
<td>47.767</td>
<td>2.410</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>15</td>
<td></td>
<td>0.480</td>
<td>126.629</td>
<td>5.023</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>18</td>
<td></td>
<td>1.690</td>
<td>284.697</td>
<td>10.405</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>21</td>
<td></td>
<td>4.460</td>
<td>632.539</td>
<td>19.783</td>
<td></td>
</tr>
<tr>
<td>144</td>
<td>24</td>
<td></td>
<td>10.960</td>
<td>1255.980</td>
<td>42.487</td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>27</td>
<td></td>
<td>26.070</td>
<td>2328.012</td>
<td>69.736</td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>30</td>
<td></td>
<td>58.700</td>
<td>4170.468</td>
<td>109.667</td>
<td></td>
</tr>
<tr>
<td>264</td>
<td>33</td>
<td></td>
<td>106.140</td>
<td>7605.915</td>
<td>191.514</td>
<td></td>
</tr>
</tbody>
</table>

Table 5
Bivariate solver: profiling, prob = 0.98.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Calls</th>
<th>Time</th>
<th>Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subresultant chain</td>
<td>1</td>
<td>0.238</td>
<td>33.85</td>
</tr>
<tr>
<td>RecDen</td>
<td>41</td>
<td>0.344</td>
<td>48.93</td>
</tr>
<tr>
<td>Conversions</td>
<td>17</td>
<td>0.076</td>
<td>10.81</td>
</tr>
</tbody>
</table>

5.4. Profiling information for the solvers

We conclude this section with profiling information for the bivariate solver and the two-equation solver. 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 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 2500 and 10,000 solutions, the C computation takes about 40% of the time; RecDen computations take roughly 50%; other MAPLE functions take 5% and the conversion time is less than 5%.

The profiling information in Fig. 8 also concerns the Bivariate solver; there, the sample input intends to generate many splittings (we take prob = 0.5, as in the examples in the previous subsection). The conversion time slowly increases but does not become the bottleneck (28%–38%).

**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 6, \(N\) is the number of variables and \(d_1, d_2\) are the degrees of \(T_1, T_2\) respectively. The C level computation is the major factor of the total computation time; it reaches 91% in case \(N = 4, d_1 = 5, d_2 = 5\).

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 performance. 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. This algorithm may generate large amounts of “external” calls to the C functions, so the data conversion between MAPLE and C becomes dominant in timings. For these kinds 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.

Acknowledgements

We wish to thank two referees, whose careful reading helped us to improve the presentation of this paper. This research was partly supported by NSERC, MITACS, and the Canada Research Chair program.

References