The `modpn` library: Bringing Fast Polynomial Arithmetic into MAPLE

X. Li, M. Moreno Maza, R. Rasheed, É. Schost
Ontario Research Center for Computer Algebra
University of Western Ontario, London, Ontario, Canada.
{xli96,moreno,rrasheed,eschost}@csd.uwo.ca

One of the main successes of the computer algebra community in the last 30 years has been the discovery of algorithms, called modular methods, that allow to keep the swell of the intermediate expressions under control. Without these methods, many applications of computer algebra would not be possible and the impact of computer algebra in scientific computing would be severely limited. Amongst the computer algebra systems which have emerged in the 70's and 80's, MAPLE and its developers have played an essential role in this area.

Another major advance in symbolic computation is the development of implementation techniques for asymptotically fast (FFT-based) polynomial arithmetic. Computer algebra systems and libraries initiated in the 90's, such as MAGMA and NTL, have been key actors in this effort.

In this extended abstract, we present `modpn`, a MAPLE library dedicated to fast arithmetic for multivariate polynomials over finite fields. The main objective of `modpn` is to provide highly efficient routines for supporting the implementation of modular methods in MAPLE.

We start by illustrating the impact of fast polynomial arithmetic on a simple modular method by comparing its implementations in MAPLE with classical arithmetic and with the `modpn` library. Then, we discuss the design of `modpn`. Finally, we provide an experimental comparison.

1 The impact of fast polynomial arithmetic

To illustrate the speed-up that fast polynomial arithmetic can provide we use a basic example: the solving of a bivariate polynomial system. We give a brief sketch of the algorithm of [LMR08].

Let $F_1, F_2 \in \mathbb{K}[X_1, X_2]$ be two bivariate polynomials over a prime field $\mathbb{K}$. For simplicity, we make three *genericity assumptions* which are easy to relax:

1. F_1 and F_2 have positive degree with respect to X_2,
2. the zero set $V(F_1, F_2) \subset \overline{\mathbb{K}}^2$ is non-empty and finite (where $\overline{\mathbb{K}}$ is an algebraic closure of $\mathbb{K}$),
3. no point in $V(F_1, F_2)$ cancels the GCD of the leading coefficients of F_1 and F_2 with respect to X_2.

Then the algorithm below, ModularGenericSolve2(F_1, F_2), computes a triangular decomposition of $V(F_1, F_2)$.

Input: F_1, F_2 as above

Output: regular chains $(A_1, B_1), \ldots, (A_e, B_e)$ in $\mathbb{K}[X_1, x_2]$ such that $V(F_1, F_2) = \bigcup_{i=1}^{e} V(A_i, B_i)$.

ModularGenericSolve2(F_1, F_2) ==
(1) **Compute** the subresultant chain of F_1, F_2
(2) **Let** R_1 be the resultant of F_1, F_2 with respect to X_2
(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 $j := i + 1$; **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) $R_1 := G$; $j := i + 1$

In Step (1) we compute the subresultant chain of F_1, F_2 in the following *lazy fashion*:

1. Let B be a bound for the degree of R_1, for instance $B = 2\,d_1 d_2$ 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 subresultant in $\mathrm{src}(F_1, F_2)$.

In Step (5) we consider the regular subresultant S_j 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. The correctness of this algorithm follows from the block structure theorem and the specialization property of subresultants [GCL92].

We have realized two implementations of this modular algorithm. One is based on classical polynomial arithmetic and is written entirely in MAPLE whereas the other one relies on fast polynomial arithmetic provided by our C low-level routines. Figure 1 below corresponds to experiments with the former implementation and Figure 2 with the latter. In each case, the comparison is made versus the `Triangularize` command of the `RegularChains` library [LMX05]. Note that, over finite fields, the `Triangularize` command does not use any modular algorithms or fast arithmetic.

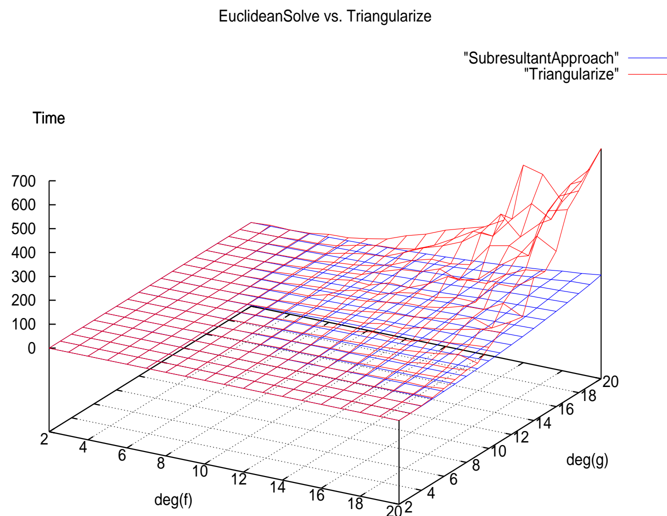

Figure 1: ModularGenericSolve2 vs. `Triangularize` in $\mathbb{Z}/p\mathbb{Z}[X_1, X_2]$, pure MAPLE code

The implementation of ModularGenericSolve2 compared in Figure 1 to `Triangularize` is written purely in MAPLE; both functions rely on MAPLE built-in DAG polynomials. The input systems are random and

d1	d2	Nsols	LexGB	FastTriade	Triangularize
10	10	50	0.280	0.044	1.276
15	15	100	1.892	0.104	16.181
20	20	150	6.224	0.208	54.183
25	25	200	15.041	4.936	115.479
15	15	100	1.868	0.100	7.492
20	20	200	14.544	0.308	47.683
25	25	300	49.763	1.268	282.249
30	30	400	123.932	1.152	907.649
20	20	150	6.176	0.188	17.105
25	25	300	50.631	1.852	117.195
30	30	450	171.746	1.341	575.647
35	35	600	445.040	7.260	2082.158
25	25	200	14.969	0.564	40.202
30	30	400	124.680	2.132	238.287
35	35	600	441.416	2.300	1164.244

Figure 2: ModularGenericSolve2 using `modpn` vs. `Triangularize` in $\mathbb{Z}/p\mathbb{Z}[X_1, X_2]$

dense; the horizontal axes correspond to the partial degrees d_1 and d_2. We observe that for input systems of about 400 solutions the speed-up is about 10.

In Figure 2, ModularGenericSolve2 is renamed FastTriade and relies on the `modpn` library. We also provide the timings for the command `Groebner:-Basis` using the `plex` terms order, since this produces the same output as ModularGenericSolve2, and `Triangularize` on our input systems. The invoked Gröbner basis computation consists of a degree basis (computed by the MAPLE code implementation of the F4 Algorithm) followed by a change basis (computed by the MAPLE code implementation of the FGLM Algorithm). We observe that for input systems of about 400 solutions the speed-up between ModularGenericSolve2 is now about 100.

2 The design of `modpn`

We designed and implemented a MAPLE library called `modpn`, which provides fast arithmetic for the polynomial ring $\mathbb{Z}/p\mathbb{Z}[X_1, \ldots, X_n]$, where p is a prime number; currently this library only supports machine word size prime.

Overview. `modpn` is a platform which supports general polynomial computations, and especially modular algorithms for triangular decomposition. The high performance of `modpn` relies on our C package reported in [Li05, FLMS06, LM06, LMS07], together with other new functionalities, such as fast interpolation, triangular Hensel lifting and subresultant chains computations. In addition, `modpn` also integrates MAPLE Recursive Dense (`RecDen`) polynomial arithmetic package for supporting dense polynomial operations. The calling to C and `RecDen` routines is transparent to the MAPLE users.

With the support of `modpn`, we have implemented in MAPLE a high level algorithm for polynomial system solving: regular GCD computations and their special case of bivariate systems, as presented in [LMR08]. The performance of our bivariate is satisfactory and reported in Section 3.

Challenges and solutions. Creating a highly efficient library is one of the most important and challenging components for this work. The difficulties result from the following aspects.

First, we mix MAPLE code with C code. MAPLE users may call external C routines using the `ExternalCalling` package. However, the developers need to implement efficient data type converters to transform the MAPLE level data representation into C level and vice versa. This task was all the more demanding as we used two MAPLE polynomial encodings and designed another three at C level.

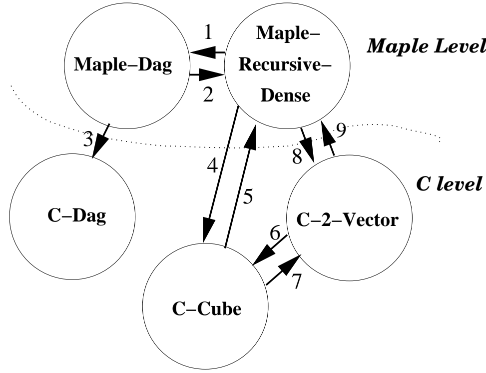

Figure 3: The polynomial data representations.

Second, the C level operations themselves form a complex setup: the top level functions, such as triangular Hensel lifting and subresultant-based methods, rely on interpolation and fast arithmetic modulo a triangular set, which themselves eventually rest on FFT/TFT-based polynomial arithmetic such as fast multiplication and division.

Finally, removing the bottlenecks in the mixed code and identifying cut-offs between different methods is a quite subtle and time-consuming task.

In the following subsections we will report on some technical aspects of the code integration and implementation methods for several core algorithms supported by `modpn`.

2.1 Code integration in Modpn

In [LMS07], we have described how to integrate our C package into AXIOM. Basically, we linked C code directly into the AXIOM kernel to make new functionalities available in the AXIOM interpreter. However, having no access to the MAPLE kernel, the only way to use external C code is to rely on the MAPLE `ExternalCalling` package. The MAPLE level data structures such as DAG's and trees have to be transformed and passed to C level, and vice versa. This step needs to be designed carefully to avoid producing bottlenecks. Moreover, the use of multiple data encodings in our library makes the code integration much harder.

Indeed, we use five polynomial encodings in our implementation, showed in Figure 3. The *Maple-Dag* and *Maple-Recursive-Dense* polynomials are MAPLE built-in types; the *C-Dag*, *C-Cube* and *C-2-Vector* polynomials are written in C. Each encoding is adapted to certain applications; we switch between different representations at run-time.

Maple polynomials. MAPLE polynomial objects by default are encoded as Directed Acyclic Graphs (DAG's). To use built-in MAPLE packages such as `RegularChains` we need such a *Maple-Dag* representation. On the other hand, we rely on the Maple `RecDen` package for several operations which are not implemented yet in our C package.

C polynomials. The *C-Cube* polynomials are our data representation for dense polynomials, already used in [LMS07]. Each polynomial is encoded by a multidimensional array holding all its coefficients (hence the name), whose dimensions are given in advance. This encoding is suitable for dense triangular set arithmetic and FFT/TFT based methods.

We also implemented a *C-Dag* polynomial representation, mainly to support Hensel lifting techniques. The *C-Dag* polynomials are encoded in the adjacency-list representation; each node contains a 32-bit header word for keeping the type, id, visiting history, and liveness information.

To make data conversion to `RecDen` more efficient, we finally designed a so-called *C-2-Vector* encoding, described below.

Conversions. In Figure 3, directed edges describe the conversions we use. Edges 1 and 2 are the conversions between *Maple-Dag* and `RecDen` polynomials; they are provided by the `RecDen` package. We implemented

the other conversion functions in MAPLE and C: conversions between C representations are of course written in C; conversions between the two languages involve two-sided operations (preparing the data on one side, decoding it on the other side).

Edge 3 stands for the conversion from *Maple-Dag* to *C-Dag*. *Maple-Dag* polynomials are traversed and packed into an array; this array is passed to and unpacked at the C-level, where common sub-expressions are identified using a hash table.

As mentioned before, the *C-Cube* is the canonical data representation in our fast polynomial arithmetic package; edges 4-9 serve the purpose of communicating between this format and `RecDen`. Edges 4 and 5 are in theory sufficient for this task. However, the *C-Cube* polynomials are in dense encoding, including all leading zeros up to some pre-fixed degree bound. This is the appropriate data structure for most operations modulo triangular sets; however, this raises the issue of converting all useless zeros back to MAPLE.

To make these conversions more efficient, we used our so-called *C-2-Vector* encoding, which essentially matches `RecDen` encoding, together with edges 6-9. Roughly, a multivariate polynomial is represented by a tree structure; each sub-tree is a coefficient polynomial. Precisely, in the *C-2-Vector* encoding, we use one vector to encode the degrees of all sub-trees in their main variables, and another vector to hold the coefficients, using the same traversal order. Thus, to locate a coefficient, we use the degree vector for finding indices. This encoding avoids to use any C pointer, so it can be directly passed back from C to MAPLE and decoded as a `RecDen` polynomial in a straightforward manner.

2.2 The Modpn Maple level

`modpn` appears to the users a pure MAPLE library. However, each `modpn` polynomial contains a `RecDen` encoding, a C encoding, or both. The philosophy of this design is still based on our long-term strategy: implementing the efficiency-critical operations in C and more abstract algorithms in higher level languages. When using `modpn` library, *Maple-Dag* polynomials will be converted into `modpn` polynomials. The computation of `modpn` will be selectively conducted by either `RecDen` or our C code up, depending on the application. Then, the output of `modpn` can be converted back to *Maple-Dag* by another function call.

2.3 The Modpn C level

The basic framework of our C implementation was described in [FLMS06, LMS07]: it consists of fast finite field arithmetic, FFT/TFT based univariate/multivariate polynomial arithmetic and computation modulo a triangular set, such as normal form, inversion and gcd. In this paper, we implemented higher-level algorithms on top of our previous code: interpolation, Hensel lifting and subresultant chains.

Triangular Hensel lifting. We implemented the *Hensel lifting of a regular chain* [Sch03] since this is a fundamental operation for polynomial system solving. The solver presented in [DJMS08] is based on this operation and is implemented in the `RegularChains` library. For simplicity, our recall of the specifications of the Hensel lifting operation is limited to regular chains consisting of two polynomials in three variables.

Let $X_1 < X_2 < X_3$ be ordered variables and let F_1, F_2 be in $\mathbb{K}[X_1, X_2, X_3]$. Let $\mathbb{K}(X_1)$ be the field of univariate rational functions with coefficients in $\mathbb{K}$. We denote by $\mathbb{K}(X_1)[X_2, X_3]$ the ring of bivariate polynomials in X_2 and X_3 with coefficients in $\mathbb{K}(X_1)$. Let π be the projection on the X_1-axis. For $x_1 \in \overline{\mathbb{K}}$, we denote by Φ_{x_1} the evaluation map from $\mathbb{K}[X_1, X_2, X_3]$ to $\overline{\mathbb{K}}[X_2, X_3]$ that replaces X_1 with x_1. We make two assumptions on F_1, F_2. First, the ideal $\langle F_1, F_2 \rangle$, generated by F_1 and F_2 in $\mathbb{K}(X_1)[X_2, X_3]$, is radical. Secondly, there exists a triangular set $\mathcal{T} = \{T_2, T_3\}$ in $\mathbb{K}(X_1)[X_2, X_3]$ such that $\mathcal{T}$ and F_1, F_2 generate the same ideal in $\mathbb{K}(X_1)[X_2, X_3]$. Under these assumptions, the following holds: for all $x_1 \in \overline{\mathbb{K}}$, if x_1 cancels no denominator in $\mathcal{T}$, then the fiber $V(F_1, F_2) \cap \pi^{-1}(x_1)$ satisfies

$$V(F_1, F_2) \cap \pi^{-1}(x_1) \;=\; V(\Phi_{x_1}(T_2), \Phi_{x_1}(T_3)).$$

We are ready to specify the Hensel lifting operation. Let x_1 be in $\mathbb{K}$. Let $N_2(X_2), N_3(X_2, X_3)$ be a triangular set in $\mathbb{K}[X_2, X_3]$, monic in its main variables, and with N_3 reduced with respect to N_2, such that we have

$$V(\Phi_{x_1}(F_1), \Phi_{x_1}(F_2)) = V(N_2, N_3).$$

We assume that the Jacobian matrix of $\Phi_{x_1}(F_1), \Phi_{x_1}(F_2)$ is invertible modulo the ideal $\langle N_2, N_3\rangle$. Then, the Hensel lifting operation applied to F_1, F_2, N_2, N_3, x_1 returns the triangular set $\mathcal{T}$. Using a translation if necessary, we assume that x_1 is zero. This simplifies the rest of the presentation.

The Hensel lifting algorithm progressively recovers the dependency of $\mathcal{T}$ on the variable X_1. At the beginning of the kth step, the coefficients of $\mathcal{T}$ are known modulo $\langle X_1^{\ell-1}\rangle$, with $\ell = 2^k$; at the end of this step, they are known modulo $\langle X_1^{\ell}\rangle$. Most of the work consists in reducing the input system and its Jacobian matrix modulo $\langle X_1^{\ell}, N_2, N_3\rangle$, followed by some linear algebra, still modulo $\langle X_1^{\ell}, N_2, N_3\rangle$.

To reduce F_1, F_2 modulo $\langle X_1^{\ell}, N_2, N_3\rangle$, we rely on our DAG representation. We start from X_1, X_2, X_3, which are known modulo $\langle X_1^{\ell}, N_2, N_3\rangle$; then, we follow step-by-step the DAG for F_1, F_2, and perform each operation (addition, multiplication) modulo $\langle X_1^{\ell}, N_2, N_3\rangle$. A "visiting history" bit is kept in the headword for each node to avoid multiple visits; a "liveness" bit is used for nullifying a dead node.

To reduce the Jacobian matrix of F_1, F_2, we proceed similarly. We used the plain (direct) automatic differentiation mode to obtain a DAG for this matrix, as using the reverse mode does not pay off for square systems such as ours. Then, this matrix is inverted using standard Gaussian elimination.

In this process, modular multiplication is by far the most expensive operation, justifying the need for the FFT / TFT based multiplication algorithms presented in [LMS07]. The other operations are relatively cheap: rational reconstruction uses extended Euclidean algorithm (we found that even a quadratic implementation did not create a bottleneck); the stop criterion is a reduction in dimension zero, much cheaper than all other operations.

Evaluation and interpolation. Our second operation is the implementation of fast multivariate evaluation and interpolation on multidimensional rectangular grid (which thus reduces to tensored versions of univariate evaluation / interpolation). When having primitive roots of unity in the base field, and when these roots of unity are not "degenerate cases" for the problem at hand (e.g., do not cancel some leading terms, etc), we use multidimensional DFT/TFT to perform evaluation and interpolation at those roots. For more general cases, we use the algorithms based on subproduct-tree techniques [GG99, Chap. 10]. To optimize the cache locality, before performing evaluation / interpolation, we transpose the data of our C-cube polynomials, such that every single evaluation / interpolation pass will go through a block of contiguous memory.

3 Experimental results

We describe in this section a series of experiments for the various algorithms mentioned before. Our comparison platforms are MAPLE 11 and MAGMA V2.14-8 [BCP97]; all tests are done on a Pentium 4 CPU, 2.80GHz, with 2 GB memory. All timings are in seconds. For all our tests, the base field is $\mathbb{Z}/p\mathbb{Z}$, with $p = 962592769$ (with one exception, see below).

Bivariate systems solving. We extend our comparison of bivariate system solvers to MAGMA. As above, we consider random dense, thus generic, systems. Experimentation with non-generic, and in particular non-equiprojectable systems will be reported in another paper. We choose partial degrees d_1 (in X_1) and d_2 (in X_2); the input polynomials have support $X_1^i X_2^j$, 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 $\deg(T_1, X_1) = d_1 d_2$ and $\deg(T_2, X_2) = 1$.

In Table 1 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. 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. 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 sec.

We refine these first results by comparing in Figure 4 our solver with MAGMA's triangular decomposition

d_1	d_2	Maple				Magma	
		`Basis`	`Solve`	`Trig`	`us`	`GB`	`Trig`
11	2	0.3	37	12	0.1	0.03	0.03
11	5	3	306	62	0.13	0.11	0.12
11	8	18	1028	122	0.16	0.32	0.32
11	11	27	2525	256	0.2	0.61	0.66

Table 1: Generic bivariate systems: all solvers

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.

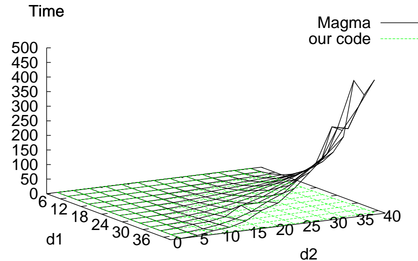

Figure 4: Generic bivariate systems: Magma vs. us.

Triangular Hensel Lifting. We conclude with testing our implementation of the Hensel lifting algorithm for a regular chain. As opposed to the previous problem, there is not a distributed package to which we could compare; hence, our reference tests are run with the original Magma implementation presented in [Sch03]. The underlying algorithms are the same; only implementations differ.

We generated trivariate systems (F_1, F_2) in $\mathbb{K}[X_1, X_2, X_3]$. Seeing X_1 as a free variable, these systems admit a Gröbner basis of the form $T_2(X_1, X_2), T_3(X_1, X_2, X_3)$ in $\mathbb{K}(X_1)[X_2, X_3]$. In our experiments, we set $\deg(T_3, X_3)$ to 2 or 4. This was achieved by generating random sparse systems f_1, f_2, and taking

$$F_1 = \prod_{j=0}^{k-1} f_1(X_1, X_2, \omega^j X_3), \quad F_2 = \prod_{j=0}^{k-1} f_2(X_1, X_2, \omega^j X_3),$$

with $\omega = -1$ or $\sqrt{-1}$, and correspondingly $k = 2$ or 4. These systems were generated by Maple and kept in unexpanded form, so that both lifting implementations could benefit from their low complexity of evaluation. We show in Figure 5 and 6 the results obtained for the cases $\deg(T_3, X_3) = 2$ and $\deg(T_3, X_3) = 4$, respectively. For the largest examples, the ratio in our favor is $21032/3206 \approx 6.5$.

4 Conclusion

To our knowledge, `modpn` is the first library making FFT/TFT-based multivariate arithmetic available to Maple end users. As illustrated in this short report, this can improve the implementation of modular algorithms in a spectacular manner. We are currently re-implementing the core operations of the `RegularChains` library by means of such algorithms creating opportunities for using the `modpn` library.

References

[BCP97] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997.

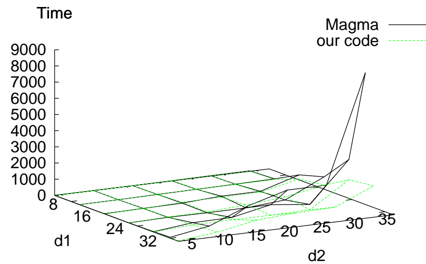

Figure 5: Lifting, MAGMA vs. us.

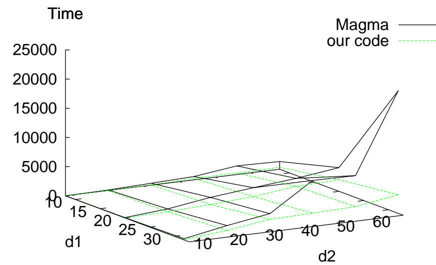

Figure 6: Lifting, MAGMA vs. us.

[DJMS08] X. Dahan, X. Jin, M. Moreno Maza, and É Schost. Change of ordering for regular chains in positive dimension. *Theoretical Computer Science*, 392(1-3):37–65, 2008.

[FLMS06] A. Filatei, X. Li, M. Moreno Maza, and É. Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *Proc. ISSAC'06*, pages 93–100, New York, NY, USA, 2006. ACM Press.

[GCL92] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[GG99] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[Li05] X. Li. Efficient management of symbolic computations with polynomials, 2005. University of Western Ontario.

[LM06] X. Li and M. Moreno Maza. Efficient implementation of polynomial arithmetic in a multiple-level programming environment. In A. Iglesias and N. Takayama, editors, *Proc.* International Congress of Mathematical Software - ICMS 2006, pages 12–23. Springer, 2006.

[LMR08] X. Li, M. Moreno Maza, and R. Rasheed. Fast arithmetic and modular techniques for polynomial gcds modulo regular chains, 2008.

[LMS07] X. Li, M. Moreno Maza, and É. Schost. Fast arithmetic for triangular sets: From theory to practice. In *Proc. ISSAC'07*, pages 269–276. ACM Press, 2007.

[LMX05] F. Lemaire, M. Moreno Maza, and Y. Xie. The `RegularChains` library. In Ilias S. Kotsireas, editor, Maple Conference 2005, pages 355–368, 2005.

[Sch03] É. Schost. Complexity results for triangular sets. *J. Symb. Comp.*, 36(3-4):555–594, 2003.