Toward high-performance computer algebra with 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

Abstract

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: they increased in a spectacular manner the range of problems solvable by computer algebra systems.

In this report, we present `modpn`, a MAPLE library dedicated to fast arithmetic for multivariate polynomials. The main objective of `modpn` is to provide highly efficient routines for supporting the implementation of modular methods in MAPLE. We demonstrate in this work that `modpn` allows us to re-implement core operations in MAPLE bringing huge performance increases and offering to MAPLE the ability of solving problems which were previously out of reach.

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

With the need for high performance in practical problems, research in symbolic computation has entered a new era where implementation techniques have become as important as theoretical developments. This phenomenon is heavily accentuated by the revival and the democratization of parallel architectures. The previous works in parallel symbolic computing are obsolete due to the emerging programing paradigms and architectures; they had also a limited impact and, the attention of the symbolic computation community was withdrawn from parallelism during the last decade. One of the themes of our MITACS project *Mathematics of Computer Algebra and Analysis (MOCAA)* is to deliver the mathematical algorithms, implementation techniques and software for symbolic computation to exploit these new computing resources, from SMP to multi-core laptops.

In previous work [2, 7, 9] we have investigated the integration of asymptotically fast arithmetic operations into the computer algebra system AXIOM. Since AXIOM is based today on GNU Common Lisp (`GCL`), we took the following approach. We realized highly optimized implementations of our 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 enhance existing AXIOM polynomial types with our fast routines (for univariate multiplication, division, GCD etc.) and call them in existing generic packages (for instance, for univariate square-free factorization). See [9] for details.

In the present report, 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

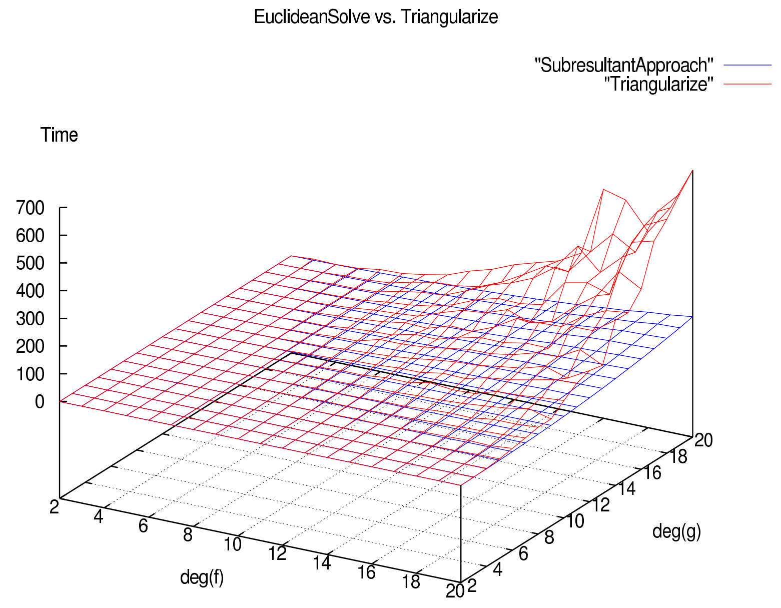

Figure 1: ModularGenericSolve2 vs. `Triangularize` in $\mathbb{Z}/p\mathbb{Z}[X_1, X_2]$, pure MAPLE code

`RegularChains` library, our main focus here, which could greatly benefit from our fast routines for triangular decompositions [10, 8]. This integration is made more difficult by the following factors.

First, compiled C code and MAPLE interpreted code are executed by two different runtime systems. This leads to data conversion overheads. Secondly, end-user objects must be allocated and de-allocated by MAPLE, which implies that most data conversions between C and MAPLE must be performed on the MAPLE side. (Clearly, one would prefer 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.

With `modpn`, our MAPLE library dedicated to fast arithmetic for multivariate polynomials, we attempt to provide elements of answers to these issues, see [11] for a technical exposition. In this report, we focus on the significance and outcome of `modpn`. We start by illustrating the impact of fast polynomial arithmetic on a simple modular method: we compare its implementations in MAPLE using both classical arithmetic and the `modpn` library, see Section 1. Then, we discuss the design of `modpn`. In particular we present the software architecture. The last part of this report is dedicated to performances and impact: we describe how the core operations of the `RegularChains` library are being re-implemented such that the `modpn` library could bring huge speed-up factors.

2 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 have realized two implementations of the modular algorithm ModularGenericSolve2 of [8]. 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 above 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 [5]. Note that, over finite fields, the `Triangularize`

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 $\mathbb{Z}/p\mathbb{Z}[X_1, X_2]$

command does not use any modular algorithms or fast arithmetic.

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 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.

3 The design of `modpn`

`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 [6, 2, 7, 10], 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.

We use five polynomial encodings in our implementation, shown 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. For instance the *C-Dag* representation supports computations with straight-line programs in the triangular Hensel lifting whereas the *C-Cube* representation supports multi-dimensional FFT computations for fast multivariate evaluation and interpolation.

The efficiency of our C routines follows from a variety of code optimization techniques. For instance, in order to optimize cache locality, while performing evaluation / interpolation, we transpose the data of our C-cube polynomials, such that every single evaluation / interpolation pass goes through a block of contiguous memory.

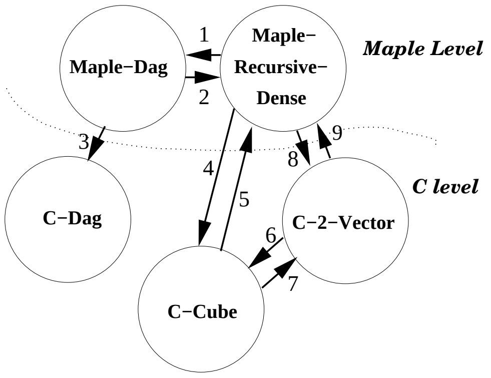

Figure 3: The polynomial data representations.

4 Performances and impact

Based on the `modpn` library, we have developed a new module of the `RegularChains` library where we have re-implemented core operations such as *polynomials GCDs modulo a regular chain* or *invertibility test of a polynomial modulo a regular chain*. For these operations, we have transformed the original algorithms of [12] in order to create opportunities for using fast polynomial arithmetic. We report on an example below.

For a prime number 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 "solve" the system $F_1 = F_2 = 0$ by computing the resultant R of F_1 and F_2 w.r.t. X_3 and the GCD of F_1 and F_2 w.r.t. the square-free part of R. We compared our code (`FastTriangularize`) to the `Triangularize` function of the MAPLE `RegularChains` package. In MAGMA [1], 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 1 summarizes the timings (in seconds) obtained on random dense polynomials by the approaches above (in the same order). Our new code performs significantly better 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.

d_1	d_2	MAPLE		MAGMA	
		`Triangularize`	`FastTriangularize`	Triangular decomposition	Resultant + GCD
2	4	0.3	0.06	0.03	0.01
4	4	1.4	0.15	0.03	0.3
6	4	25	0.27	0.7	12
8	4	257	0.52	6.9	155
10	4	1933	1.02	46.7	1012

Table 1: Solving two equations in three variables

5 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

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

[2] 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.

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

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

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

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

[7] 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.

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

[9] 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.

[10] 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.

[11] X. Li, M. Moreno Maza, R. Rasheed and É. Schost. High-Performance Symbolic Computation in a Hybrid Compiled-Interpreted Programming Environment In *Proc. 2008 International Conference on Computational Sciences and Its Applications*, pages 331–341. IEEE Computer Society, 2009.

[12] 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.