A probabilistic parallel modular algorithm for rational univariate representation

Bernard Parisse

A probabilistic parallel modular algorithm for rational univariate representation.

Bernard Parisse
Institut Fourier
UMR 5582 du CNRS
Université de Grenoble Alpes

June 2021

Abstract

This article does not introduce new mathematical ideas, it is more a state of the art June 2021 picture about solving polynomial systems efficiently by reconstructing a rational univariate representation with a very high probability of correctness using Groebner revlex computation, Berlekamp-Massey algorithm and Hankel linear system solving modulo several primes in parallel.

This algorithm is implemented in Giac/Xcas since version 1.7.0-13, it has (June 2021) leading performances on multiple CPU, at least for an open-source software.

1 Introduction

Polynomial system solving can be performed by doing several eliminations. If the variables are $x_1, .., x_n$, after eliminating $x_1, ..., x_{n-1}$, one has to solve one (large degree) univariate polynomial, and then one finds other unknowns by back substitution and univariate polynomial solving. Unfortunately this method, named regular chains, requires building tower of algebraic extensions over $\mathbb{Q}$, which is computation intensive.

It is more efficient to build one algebraic extension of $\mathbb{Q}$ (or more if the system factors) such that all components of the solutions of the system will live in this extension. This can be performed by computing a Gröbner basis of I, the ideal spanned by the multivariate polynomials of the system. Then, if the ideal is 0 dimensional, select one variable (say x_n), find the minimal polynomial of this variable. If this polynomial m has the right degree (the dimension of the polynomials modulo the ideal as a vector space) and is square free, for all other variables, find P_i such that $x_i - P_i(x_n)$ is inside the ideal. Then the system solutions are $P_1(x_n), P_2(x_n), ..., x_n$ for all roots x_n of m.

Rouillier ([8]) found that it is more efficient to compute Q_i such that $x_i - Q_i(x_n)/m'(x_n)$ is inside the ideal, and this is called rational univariate representation.

If for all unknowns, the minimal polynomial degree is too small, a linear separating form (linear combination of the x_i), must be found such that the minimal polynomial is of degree the dimension of the vector space of the polynomials modulo the ideal. If

the ideal is not radical (i.e. if there is a polynomial P that does not belongs to I but $P^k \in I$ for some $k > 0$), the method must be adapted.

Modular algorithms are well known techniques in CAS to make efficient computations, they are also good candidates for parallelization. In our context, algorithms were presented early, like in [6] (A Modular Method to Compute the Rational Univariate Representation of Zero-dimensional Ideals). For more recent results, see [2].

We will review some of these algorithms as well as some algorithms of Faugère and Mou ([4]) for rur computation in $\mathbb{Z}/p\mathbb{Z}$ in Section 2. Section 3 will give more informations on the implementation inside Giac/Xcas and gives some benchmarks.

2 Algorithms for RUR computations

2.1 Gröbner basis over $\mathbb{Z}/p\mathbb{Z}$ (degree rev. lex. ordering)

For a basis computation without additional hypothesis, it seems that F4 ([5]) is a very good choice. Since the same basis is computed several times for different primes, we can store some informations during the first run, like s-pairs reducing to 0, in order to speed up computation for the next primes, this is described in more details in [7] (A probabilistic and deterministic modular algorithm for computing Groebner basis over $\mathbb{Q}$).

2.2 From Gröbner basis to RUR over $\mathbb{Z}/p\mathbb{Z}$

If the ideal I spanned by the polynomials of the system is 0-dimensional, the polynomials modulo I belong to V, a vector space of finite dimension d. We can compute a basis of V by collecting all the monomials smaller than the leading ones in G.

The reduction of $1, x_n, .., x_n^d$ with respect to the Gröbner basis is not free, there is a minimal polynomial m of degree at most d such that $m(x_n) = 0 \pmod I$. Computing m is a linear algebra kernel computation (for a matrix with columns the components of the reduction mod I of x_n^i). This is an $O(d^3)$ computation with a naive Gauss pivoting method. Fortunately, it can be computed faster, by observing that if

$$\sum_i m_i x_n^i = 0 \pmod I$$

then it's still true after multiplication by a power of x_n, and therefore the scalar product of a fixed vector and the reduced vector is 0, then the m_i coefficients can be found by mean of the Berlekam-Massey algorithm, using the half-gcd fast version in $O(d\log(d))$ operations (see e.g. [9], Yap), once the scalar products are computed.

There is still a naive $O(d^3)$ part in this algorithm, computing the $x_n^k \pmod I$. This is done by computing the matrix of the multiplication by $x_n \pmod I$ in the basis of V, by reducing all monomials of this basis multiplied by x_n. In many situations, the multiplication by x_n will give a monomial of the basis of V, and the corresponding column of M is trivial (one 1 and all other coefficients 0). Or the multiplication by x_n will return the leading monomial of one element of G and the reduction is trivial (take the opposite of all remaining monomials of this element of G). The remaining

products by x_n must be reduced mod G, this can be done simultaneously for all these products like in the F4 algorithm. The multiplication of a vector by M is still $O(d^2)$, but it becomes faster if the matrix has many trivial columns, and it's a simple operation that can benefit from the CPU instruction set.

In a generic situation, the minimal polynomial of x_n is of maximal degree d the dimension of V the vector space of polynomials modulo I and every element of V (i.e. any polynomial modulo the ideal I) can be expressed as a polynomial in x_n, of degree $< d$. Finding the polynomial corresponding to $x_1, .., x_{n-1}$ will give the solution to the initial system. It can be seen as a linear system of matrix with columns the powers 0 to $d-1$ of x_n reduced modulo I and with second member x_i modulo I. Which means solving $n-1$ systems with the same $d \times d$ matrix, where $n << d$, at a $O(d^3)$ cost (naive algorithm). Fortunately, this can be improved, by observing that if $x_i = P_i(x_n) \pmod I$ then $x_n^k x_i = x_n^k P_i(x_n) \pmod I$, then one can do the scalar product of this equation with any fixed vector, and get a linear equation in the coefficients of P_i that we are computing. Doing that for $k = 0, ..., d-1$ will bring a linear system with a structured matrix. This matrix is named a Hankel matrix, it's coefficients are already computed: it's the coefficients of the Berlekamp-Massey algorithm that returned the minimal polynomial of x_n. Hankel matrices can be inverted using an extended GCD and a Bezoutian matrix, cf. for example wikipedia and the cost for computing a Bezoutian matrix is $O(d^2)$, cf. e.g. [3] (Fast computation of the Bezout and Dixon resultant matrices)

2.3 Non generic situations

If the minimal polynomial of x_n is not of maximal degree, one can try the other monomials x_1 to x_{n-1} and if it does not work, a random linear combination of $x_1, ..., x_n$, this is called a separating linear form.

If the minimal polynomial is of maximal degree d but is not squarefree, then the ideal I is not radical, in that case one can add the squarefree part of the minimal polynomial to the generators of the ideal and compute a new Gröbner basis, until the ideal is radical. Cf. Faugère and Mou ([4]) for alternative methods.

2.4 Rational reconstruction

It is of course possible to compute a Gröbner basis over $\mathbb{Q}$ and run the same kind of computations, but field operations in $\mathbb{Q}$ are not performed in $O(1)$ time, that's why a multi-modular algorithm is most of the time more efficient.

The first step is to cancel denominators so that the coefficients belong to $\mathbb{Z}$. A prime p is said to be a good reduction prime if the steps of the computation over $\mathbb{Z}/p\mathbb{Z}$ are the reduction modulo p of the steps over $\mathbb{Q}$. This is true if the leading monomials of the s-pairs do not cancel mod p, if the basis of the vector space of the polynomials modulo the ideal remains the same, if the degree of the minimal polynomial of the separating linear form remains d. Therefore before trying to reconstruct a rur in $\mathbb{Q}$ from several primes, we must check the consistency of these primes. Two primes are compatible if the leading monomials of the Gröbner basis have the same power exponents. If one

prime has a gbasis with less elements, or a basis of V with less elements, it must be discarded.

Reconstruction is done coefficient by coefficient using Farey algorithm. If the reconstructed rur modulo the next prime p matches the computation over $\mathbb{Z}/P\mathbb{Z}$ (where P is the product of the previous primes), then the probability of a bad rur reconstruction is very low and can be as low as desired by checking with a few other primes rur computations. Certifying a rur is however very costly, because we must either reduce the rur elements modulo the $\mathbb{Q}$-gbasis (which must be certified as well) or check that the solutions verify the initial system, and that's a very large computation with univariate fractions with rational coefficients.

My estimate for the probability to have a bad prime for a today-large computation is less than `1e-4` (with about `1e4` leading coefficients for a prime of size about `5e8` in Giac), hence if a few thousands primes are required to stabilize the computation over $\mathbb{Q}$, the probability to meet one bad prime would be less than 0.1.

3 Giac/Xcas implementation and benchmarks

3.1 Implementation

- Step 1: compute the gbasis for revlex order modulo a prime p . Giac implementation details of the gbasis algorithm with learning are described in [7]. If p is not the first prime, compare if the current prime is compatible with previous one (leading monomials of the gbasis must be the same), if not discard it (or all previous primes).

- Step 2 (for the first prime): find the dimension and a basis of V made of monomials. We collect the leading monomials of the gbasis. For every variable x_i we search a leading monomial $x_i^{d_i}$ that is a power of this variable. This will bound any monomial exponent in V by $(d_1, ..., d_n)$. The dimension d of V is smaller than D the product of d_i. For any integer $0 \leq i < D$, write i in multi-basis $d_1, ..., d_n$

$$i = (..(i_1 d_2 + i_2)d_3 + ... + i_{n-1})d_n + i_n, \quad 0 \leq i_k < d_k$$

  and check if $x_1^{i_1}...x_n^{i_n}$ is greater than a leading monomial of the gbasis, if not add it to the basis.

- Step 3: compute the matrix of multiplication by x_n in our basis of V. If x_n times the monomial is itself a monomial in V, we do not store a column with one 1 and $d-1$ zeros, instead we store the pairs of indices of the monomials, this is a mixed storage (dense part/sparse part).

- Step 4: compute the coefficients of the Hankel matrix (the dense multiplication part can take advantage of the AVX2 instruction set if available). For the dense part of the multiplication, we avoid divisions (except the final one) by computing representants $0 \leq r < p^2$, after an addition of a multiplication of coefficients in $[0, p)$ we substract $-p^2$ and add p^2 if the result is negative without testing (for a

63 bits signed integer i this is done by `i += (i>>63) & p2`). In Giac, we do that for 4 additions at a time (using representants in $[0, 4p^2)$ where $p < 2^{29}$).

- Step 5: find m the minimal polynomial of x_n by the halfgcd Berlekamp-Massey algorithm. If it is not of maximal degree d, replace x_n by another variable $x_1, .., x_{n-1}$ and go to step 3. If none of the variables fill the degree condition, try with a random integer linear combination of x_i. The separating linear form will be recorded for further primes.

- Step 6: if the minimal polynomial m is not squarefree, add the square free part `m/gcd(m,m')` to the gbasis, and go to step 1.

- Step 7: find the polynomials P_i such that $x_i - P_i(x_n) = 0 \pmod I$ by solving Hankel systems (using fast inversion of the Hankel matrix with bezoutians).

- Step 8: compute $Q_i = P_i m' \pmod m$

- Step 9: (if not at the first prime) Check if the Farey rational reconstruction for previous primes matches this prime for $Q_i \pmod p$ (check for a few monomials before doing a complete reconstruction check). If so, return the Farey reconstruction. Otherwise, apply the Chinese Remainder Theorem for $Q_i \pmod p$ and previous primes and go to step 1 for a next prime.

Steps 1 to 8 can be parallelized. Trying to parallelize step 9 does not speed up the computation because it requires a lot of memory allocations, and this seems to always be thread-mutually exclusive.

3.2 Benchmarks

- Giac/Xcas 1.7.0-13 timings are for a rur computation with AVX2 enabled, on an Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz. The computation were run with 16 threads in parallel, with a few exceptions with 8 threads in order to spare memory.

- msolve timings are for fglm computation with AVX2 enabled, on an Intel (R) Xeon (R) CPU E7-4820v4 @ 2.00GHz, as reported by the msolve authors, they should be multipled by about 0.77 to account for the different frequencies. On the other hand, CPU time for multi-threaded implementations are always greater than for one-threaded implementations, especially for relatively small computations or for computations requiring much memory for each prime. For examples, Katsura 9 computation takes 4.2s CPU time with 1 thread instead of 8.4s with 16 threads (real time 1.36s), Katsura 10 takes 35s with 1 thread instead of 55s with 16 threads (real time 7.9s) and Katsura 11 takes 359s with 1 thread instead of 496s with 16 threads (real time 61s)

- The current version of msolve does not support multiple CPUs, but it will certainly do in the near future.

- In order to compile Giac/Xcas with AVX2 support with gcc, install VCL vector-class by Agner Fog and run
`export CXXFLAGS='-O2 -g -mfma -mavx2 -fabi-version=0'`
before running `./configure` in the Giac/Xcas source root directory.
- The Giac/Xcas script files for these benchmarks are available here
- The last 2 columns give the time required to isolate all real roots of the minimial polynomial of the separating linear form. The algorithm is a C++-transcription of Xcas user code sent by Alkis Akritas ([1]). It is a little bit parallelized, by running isolation of positive and negative real roots in separate threads. It is most of the time one at least one order of magnitude faster than computing the polynomial, and is therefore not a priority for further optimizations.

	threads	real	msolve	CPU	RAM	realroot real	CPU
Kat9	16	1.36	4.89	8.41	67M	0.5	0.5
Kat10	16	7.9	43.7	55	181M	1.7	2.6
Kat11	16	61	424	496	573M	19.6	31.3
Kat12	16	734	6262	7922	2.4G	154	276
Kat13	16	11.2e3	89e3	136e3	9G	1640	2870
Kat14	8	0.21e6	1.3e6	1.49e6	21.5G	16e3	29.7e3
Noon7	16	452	4040	4000	1.4G	24.2	43.1
Noon8	8	0.155e6	0.599e6	1.1e6	13G	413	814
Phuoc	16	236	4467	4026	1.9G	7.66	13
Henrion6	16	6.08	138	52.6	147M	1.37	1.39
Henrion7	16	7.2e3	118e3	87.8e3	3.7G	309	314
Eco10	16	1.33	12.5	9.21	106M	0.057	0.07
Eco11	16	7.6	90.3	70.2	280M	0.37	0.39
Eco12	16	64	877	724	1.06G	2.52	2.82
Eco13	16	712	12137	9370	4G	25	27.5
Eco14	16	11.3e3	168e3	154e3	15G	112	123
cp352	16	3.6	18.1	23	81M	0.21	0.26
cp362	16	49	390	429	270	1.25	1.75
cp372	16	1.39e3	9.64e3	15e3	1.4G	52	91
cp382	8	32.5e3	270e3	213e3	4.9G	992	1655
cp443	16	6	40.9	45	115M	0.47	0.6
cp453	16	3.5e3	21.5e3	47e3	2.7G	178	200
cp366	16	36	255	290	232M	1.33	2.14
cp377	16	1.77e3	12.4e3	22e3	1.7G	45	68

This leads to the following observations :

- One bad prime (over 1352) was observed for `noon7`, $p = 534856027$, and 5 bad primes (over 4060) for `noon8`.
- Some examples above do not require as many primes as msolve. Probably because the rur stabilizes earlier than fglm.

- Except for noon7 and 8 where maple reported timings in [2] are better, the real multi-threaded timings of giac are currently the best timings. Once msolve is multi-threaded, I expect that it should be a little bit faster than Giac for most examples.

Example of Giac/Xcas code:

```
threads:=16;
//debug_infolevel:=1;
kat10:=[x1 + 2*x2 + 2*x3 + 2*x4 + 2*x5 + 2*x6 + 2*x7 + 2*x8 + 2*x9
 + 2*x10 - 1, x1^2 + 2*x2^2 + 2*x3^2 + 2*x4^2 + 2*x5^2 + 2*x6^2 +
 2*x7^2 + 2*x8^2 + 2*x9^2 + 2*x10^2 - x1, 2*x1*x2 + 2*x2*x3 +
 2*x3*x4 + 2*x4*x5 + 2*x5*x6 + 2*x6*x7 + 2*x7*x8 + 2*x8*x9 +
 2*x9*x10 - x2, x2^2 + 2*x1*x3 + 2*x2*x4 + 2*x3*x5 + 2*x4*x6 +
 2*x5*x7 + 2*x6*x8 + 2*x7*x9 + 2*x8*x10 - x3, 2*x2*x3 + 2*x1*x4 +
 2*x2*x5 + 2*x3*x6 + 2*x4*x7 + 2*x5*x8 + 2*x6*x9 + 2*x7*x10 - x4,
 x3^2 + 2*x2*x4 + 2*x1*x5 + 2*x2*x6 + 2*x3*x7 + 2*x4*x8 + 2*x5*x9 +
 2*x6*x10 - x5, 2*x3*x4 + 2*x2*x5 + 2*x1*x6 + 2*x2*x7 + 2*x3*x8 +
 2*x4*x9 + 2*x5*x10 - x6, x4^2 + 2*x3*x5 + 2*x2*x6 + 2*x1*x7 +
 2*x2*x8 + 2*x3*x9 + 2*x4*x10 - x7, 2*x4*x5 + 2*x3*x6 + 2*x2*x7 +
 2*x1*x8 + 2*x2*x9 + 2*x3*x10 - x8, x5^2 + 2*x4*x6 + 2*x3*x7 +
 2*x2*x8 + 2*x1*x9 + 2*x2*x10 - x9];
vars:=[x1,x2,x3,x4,x5,x6,x7,x8,x9,x10];
proba_epsilon:=1e-7;
time(H:=gbasis(kat10 ,vars,rur));
write("Hkat10",H); // use archive instead of write for fast read
// check, this is very slow
debug_infolevel:=0;
l:=subst(kat10,vars,H[4..size(H)-1]/H[3]):;
map(l,x->rem(numer(x),H[2],H[1]));
// real root isolation
time(R:=realroot(eval(H[2],1)));
write("Rkat10",R);
size(R);
```

4 Conclusion

We have now efficient probabilistic algorithms for rur computations over $\mathbb{Q}$. The probability to return a wrong solution is very low and may be as low as desired. Moreover, it is always possible to make a numerical check with high precision on the initial polynomial system. But certifying as fast as computing is still open and is maybe impossible.

References

[1] A. G. Akritas and A. W. Strzebonski. A comparative study of two real root isolation methods. *Nonlinear Analysis: Modelling and Control*, 10(4):297–304, 2005.

[2] J. Berthomieu, C. Eder, and M. S. E. Din. msolve: A library for solving polynomial systems. *arXiv preprint arXiv:2104.03572*, 2021.

[3] E.-W. Chionh, M. Zhang, and R. N. Goldman. Fast computation of the bezout and dixon resultant matrices. *Journal of Symbolic Computation*, 33(1):13–29, 2002.

[4] J.-C. Faugère and C. Mou. Sparse fglm algorithms. *Journal of Symbolic Computation*, 80:538–569, 2017.

[5] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases (F4). *Journal of Pure and Applied Algebra*, 139(1–3):61–88, June 1999.

[6] M. Noro and K. Yokoyama. A modular method to compute the rational univariate representation of zero-dimensional ideals. *Journal of Symbolic Computation*, 28(1-2):243–263, 1999.

[7] B. Parisse. A probabilistic and deterministic modular algorithm for computing groebner basis over $\mathbb{Q}$. *arXiv preprint arXiv:1309.4044*, 2013.

[8] F. Rouillier. Solving zero-dimensional systems through the rational univariate representation. *Applicable Algebra in Engineering, Communication and Computing*, 9(5):433–461, 1999.

[9] C.-K. Yap et al. *Fundamental problems of algorithmic algebra*, volume 49. Oxford University Press Oxford, 2000.