Groebner.jl: Fast Gröbner Tracing in Julia

Alexander Demin*

Laboratoire d'informatique de l'École polytechnique,
LIX, UMR 7161, CNRS,
1 rue Honoré d'Estienne d'Orves,
91120 Palaiseau, France
demin@lix.polytechnique.fr

Abstract. A standard way to control expression swell in computer algebra is to use multi-modular or evaluation-interpolation methods. In computations involving Gröbner bases, these techniques typically require repeatedly computing Gröbner bases of specializations of the same ideal. These repeated computations can be accelerated through precomputation, notably using Traverso's tracing.
We present `Groebner.jl`, a Julia implementation of the F4 algorithm that exposes Traverso's tracing through a reusable public interface. The implementation supports SIMD-friendly coefficient types, such as tuples of machine integers, which Julia compiles to efficient code with little manual intervention. This lets other Julia software leverage tracing to obtain speedups in applications such as structural identifiability of ordinary differential equation models and polynomial system solving.

Keywords: Gröbner basis, F4 algorithm, Modular integer arithmetic, Julia, SIMD

1 Introduction

When using multi-modular and evaluation-interpolation methods, it is typical to repeatedly compute Gröbner bases of specializations of the same ideal. Traverso's tracing [19] is a method of precomputation that accelerates such repeated calculations. One first performs a *learn* computation, a modified Gröbner basis computation that additionally records which S-polynomials reduce to zero. One may then perform several *apply* computations on specializations of the same ideal, omitting reductions of S-polynomials that are expected to reduce to zero.

High-performance implementations of the F4 algorithm [8] use tracing internally for computing Gröbner bases over the rationals, including `msolve` [2,15] and Giac/Xcas [17]; related ideas appear in [13]. Typically, these systems do not expose tracing through their user interfaces. Many applications, however, use Gröbner basis computations as intermediate subroutines rather than final outputs. Therefore, providing a reusable learn & apply interface would make tracing available to a broader class of symbolic computation workflows.

We discuss `Groebner.jl`, a Julia package that implements the F4 algorithm and exposes Traverso's tracing through public learn and apply functions [10]. To further amortize computational costs, the apply routine can compute Gröbner bases in batches. The implementation uses SIMD-friendly coefficient types and relies on Julia's LLVM-based compiler to produce efficient code across architectures with little additional implementation effort. We evaluate these techniques on applications in structural identifiability [5,7] and polynomial system solving via rational univariate representations [6].

* This work has been supported by an ERC-2023-ADG grant for the ODELIX project (number 101142171).

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Acknowledgements. We would like to thank Roman Pearce, Gleb Pogudin, Fabrice Rouillier, and Chee Yap for useful discussions. We are also grateful to the anonymous referees for helpful comments and suggestions. The project is open-source, and we are grateful to all contributors; in particular, to Matthias Franz.

2 Gröbner tracing

This section briefly describes the ideas behind Gröbner tracing using the F4 algorithm as an example. We refer to [19] for details. Tracing consists of two stages, both of which may be viewed as modifications of the classic F4 algorithm, as illustrated in Figure 1:

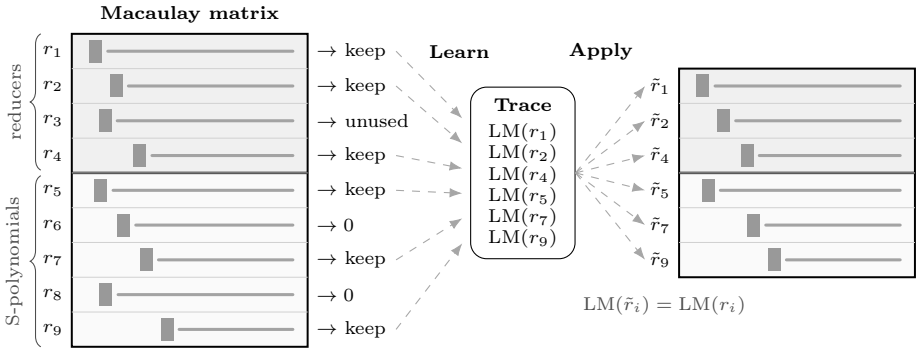

Fig. 1. A single iteration of the F4 algorithm as a row-reduction of a Macaulay matrix. Learn stage (on the left) and apply stage (on the right). LM denotes the leading monomial without coefficient.

Learn stage. **Input:** a system of polynomials F_1. **Output:** a trace. The algorithm performs a modified F4 computation on F_1. At each iteration, it constructs and row-reduces a Macaulay matrix. In addition to the operations of the standard F4 algorithm, it records which rows of the matrix reduce to zero, and stores this information in a coefficient-independent manner in a trace.

Apply stage. **Input:** a system of polynomials F_2 and a trace. **Output:** a Gröbner basis of F_2. The algorithm uses the trace to construct only the rows of the Macaulay matrix that were identified as useful during the learn stage. As a result, it avoids many of the monomial operations performed during learning and may produce a smaller matrix. It then row-reduces the matrix.

Tracing is applicable whenever several Gröbner basis computations share some monomial structure but differ in their coefficients. In this setting, F_1 and F_2 are typically obtained as specializations of a common polynomial system F.

Example 1. Let $F \subseteq \mathbb{F}_p(a)[x_1, \ldots, x_n]$. To compute the reduced Gröbner basis of F, one may first specialize a to obtain a system F_1 and perform the learn stage. The resulting trace can then be reused for many further specializations $F_2, F_3, \ldots$, each requiring only the apply stage. The Gröbner bases of these specializations can subsequently be passed to rational function interpolation to recover the basis over $\mathbb{F}_p(a)$.

3 Interface design

3.1 Generic coefficient arithmetic

`Groebner.jl` computes Gröbner bases of polynomial ideals over fields. It can be used in combination with existing symbolic manipulation packages, such as `Nemo.jl`, or by itself through a low-level interface. A standard workflow is:

```
using Groebner, Nemo                      # load packages

K, t = rational_function_field(QQ, "t") # define Q(t)[x,y,z]
R, (x, y, z) = K["x","y","z"]

F = [t*x^2*y + y^3, x*y^2 + 3//t*z]     # define polynomials

G = groebner(F)                           # compute basis
```

This produces output of the following form:

```
4-element Vector{AbstractAlgebra.Generic.MPoly{...}}:
 y^6 + 9//t*z^2
 x*z - 1//3*y^4
 x*y^2 + 3//t*z
 x^2*y + 1//t*y^3
```

The implementation in `Groebner.jl` is generic with respect to coefficient types. Thus, the package can operate over fields for which the basic arithmetic operations are implemented, whether in `Groebner.jl` itself or in external packages. In the example above, arithmetic in $\mathbb{Q}(t)$ is supplied by `Nemo.jl`. For certain coefficient types, `Groebner.jl` provides more efficient specialized implementations. Such specializations only need to implement coefficient arithmetic and, optionally, a number of performance-critical primitives, while the rest of the implementation is reused unchanged. For example, the implementation over integers modulo a prime p with $p < 2^{64}$ uses machine integers together with specialized routines for linear algebra and vector operations modulo p.

3.2 Learn and Apply

`Groebner.jl` exports tracing via the learn and apply interface. We illustrate it by computing Gröbner bases for two specializations of the Katsura-10 system:

```
kat = Groebner.Examples.katsuran(10)

# use Nemo to produce two copies of the system modulo different primes
kat_zp1 = change_base_ring.(Ref(GF(2^30+3)), kat)
kat_zp2 = change_base_ring.(Ref(GF(2^30+7)), kat)

trace, _ = groebner_learn(kat_zp1);   # learn a trace for the first prime

success, gb = groebner_apply!(trace, kat_zp2);  # apply for another prime
```

The boolean `success` reports whether any inconsistencies were detected by comparing the supports of the polynomials against the trace. This test may produce false positives: no inconsistency may be detected even when the result is not a Gröbner basis. In practice, however, it can still serve as a useful proxy.

The learn and apply pattern is particularly appealing when many applications are required, since each application is faster than an independent Gröbner basis computation. To illustrate this, we compare the running time of the default implementation in `Groebner.jl` against that of an application:

```
using BenchmarkTools

@btime groebner(kat_zp2);
#  731.000 ms (84551 allocations: 139.38 MiB)

@btime groebner_apply!(trace, kat_zp2);
```

```
#  263.576 ms (44002 allocations: 87.02 MiB)
```

Remark 1 (Choice of baseline). The default implementation in `groebner` uses the randomized linear algebra techniques following [3,16], which is state of the art. Consequently, it serves as the baseline throughout this paper. For comparison, the classical deterministic implementation takes 3.5 seconds on this example.

3.3 Application in batches

The application interface supports computing several Gröbner bases simultaneously. For example, one may perform an application with a batch of four systems, all obtained as specializations of the same system:

```
# ... define sys1,...,sys5 appropriately ...
trace, _ = groebner_learn(sys1)
success, (gb2,gb3,gb4,gb5) = groebner_apply!(trace,(sys2,sys3,sys4,sys5))
```

The output corresponds to the Gröbner bases of the systems `sys2`, `sys3`, `sys4`, and `sys5`, respectively. Internally, this allows certain operations to be shared across the four computations (see Section 4.2). For the Katsura-10 example, the application in batches of four takes approximately 680 ms; this corresponds to 170 ms per Gröbner basis, compared to 264 ms for a single application.

4 Efficient Gröbner tracing

4.1 When is tracing worthwhile?

The "learning" of a trace is typically more expensive than a single classic Gröbner basis computation. A natural question is therefore: how many Gröbner bases do we need to compute before this cost is amortized?

Table 1. Running times, and the number of computations required to amortize the cost of learning a trace compared against classic independent Gröbner basis computations.

System	Classic	Learn	Apply	Computations to break even
Chandra-11	1.0 s	3.3 s	0.1 s	4
Chandra-12	5.8 s	19.6 s	0.3 s	4
Katsura-12	2.2 s	17.2 s	0.8 s	13
Katsura-13	14.2 s	153.2 s	6.1 s	19
Cyclic-8	0.4 s	1.5 s	0.2 s	8
Cyclic-9	38.9 s	219.4 s	26.7 s	19
Cholera	13.7 s	40.9 s	7.7 s	7
Yang1	5.8 s	39.2 s	0.4 s	8

In Table 1, for each benchmark system, we present the running time of a standard independent Gröbner basis computation (per Remark 1), together with the running times of the learn and apply stages. Using these timings, we compute the smallest number of Gröbner basis computations for which learning a trace once and subsequently applying it repeatedly is faster than performing all computations independently. The resulting number is shown in the last column.

The first thing we observe is that the learn stage is always slower than the classic computation, and the apply stage is always faster. We also note that the break-even point typically is small. Consequently, when hundreds or more Gröbner bases of specializations are required (which is typical for multi-modular and evaluation-interpolation methods), the learn & apply strategy may be beneficial.

The table also shows that the benefit of tracing depends on the problem. For example, the ratio of running times $\frac{\text{Classic}}{\text{Apply}}$ is only 1.46 for Cyclic-9 but reaches 14.5 for Yang1. In Yang1, a

substantial fraction of the running time in the classic algorithm is spent on monomial operations, many of which are eliminated during the apply stage. In contrast, Cyclic-9 spends a larger fraction of its running time in linear algebra, which must still be performed during application, albeit on smaller matrices.

4.2 Application in batches

Instead of computing Gröbner bases over integers modulo primes $p_1, \ldots, p_N$ independently, one may compute over the product ring

$$\mathbb{Z}/p_1\mathbb{Z} \times \cdots \times \mathbb{Z}/p_N\mathbb{Z}, \tag{1}$$

whose elements are represented as tuples of residues with arithmetic performed component-wise. As a result, operations in the F4 algorithm such as critical-pair processing and monomial arithmetic are performed only once, shared across the primes. Thus, increasing N amortizes the cost of this part of the computation. This approach has been employed by Gräbe [11] and de Kleine and Monagan [14].

`Groebner.jl` can compute over the product rings (1) for primes $p_i < 2^{64}$ with $i = 1, \ldots, N$. The application in batches from Section 3.3 is implemented in this way. Since the F4 implementation is generic with respect to coefficient arithmetic, only a small number of linear algebra kernels need to be specialized for the arithmetic in (1), while the remainder of the learn & apply infrastructure is reused unchanged.

As an illustration of one such kernel, consider addition of a dense vector and a multiple of a sparse vector modulo a prime, one of the bottlenecks of the classic F4 algorithm. Consider implementing this operation over (1) for $N = 4$ using 31-bit primes each represented in `Int64` and using the modular arithmetic from [16]. With the broadcast syntax in Julia, such as `.+`, `.*`, `.-`, etc.[1], this can be implemented as simply as (cf. [16, Section 2.2]):

```
const N = 4
const p = (2^30+3, 2^30+7, 2^30+9, 2^30+15) # p_1, p_2, p_3, p_4
const ProductRingElem = NTuple{N,Int64}

function reduce(
        buffer::Vector{ProductRingElem},    # dense vector
        coeffs::Vector{ProductRingElem},    # sparse vector coeffs.
        idxs::Vector{Int64})                # sparse vector support
    p2 = p .^ 2
    c = buffer[idxs[1]]
    for k in 1:length(idxs)
        j = idxs[k]
        a = buffer[j] .- c .* coeffs[k]
        buffer[j] = a .+ ((a .>> 63) .& p2) # (a < 0) ? a + p2 : a
    end
end
```

This is essentially the implementation used in `Groebner.jl`. Apart from the broadcast operations, it is identical to the scalar implementation for $N = 1$.

In addition to the classical benefits of product-ring arithmetic, the resulting operations are particularly amenable to SIMD vectorization [12]; this contrasts with the scalar case, where vectorization is difficult because of irregular memory access patterns and the low arithmetic intensity in sparse linear algebra. For component-wise tuple operations in product rings we found that Julia, via LLVM, often automatically generates SIMD-optimized assembly. On Intel processors, this leads to substantial amortized speedups; on ARM architectures such as Apple M2 Max, the generated code was less effective in our experiments[2].

[1] For example, in Julia, `(1,2,3,4) .+ (5,6,7,8) == (6,8,10,12)`

[2] It is possible to inspect the assembly of the `reduce` function using the command:
`code_native(reduce, Tuple{Vector{ProductRingElem},Vector{ProductRingElem},Vector{Int64}})`

Remark 2 (The case $p_1 = \cdots = p_N$). When $p_1 = \cdots = p_N$, the computation may be viewed as performing N independent computations modulo the same prime simultaneously. This arises in evaluation-interpolation schemes, where one computes Gröbner bases for many specializations of a parametric polynomial system over a fixed finite field.

4.3 The choice of batch size

Computing over the product ring (1) introduces a parameter N, the number of primes processed simultaneously. As discussed in Section 4.2, operations such as critical-pair processing and monomial arithmetic are shared across all components of the product ring. Increasing N therefore amortizes the cost of these operations. On the other hand, larger values of N increase the amount of coefficient data and may increase memory traffic and reduce cache efficiency.

To study this trade-off, we measure the throughput of the apply stage for different values of N. For each benchmark system and each value of N, Table 2 reports the amortized speedup relative to the scalar implementation with $N = 1$.

Table 2. The amortized speed-up of using the apply stage in batches over the scalar case $N = 1$. **Boldface** marks the best speed-up in a row.

System	$N = 1$	$N = 2$	$N = 4$	$N = 8$	$N = 16$	$N = 32$
Chandra-12	1.00	1.30	1.58	1.87	**2.51**	2.05
Chandra-13	1.00	1.39	1.93	2.05	**2.27**	1.81
Katsura-12	1.00	1.50	2.23	**2.33**	2.19	2.12
Katsura-13	1.00	1.45	2.04	2.11	2.24	**2.26**
Cyclic-8	1.00	1.69	2.79	3.14	**3.28**	2.68
Cyclic-9	1.00	1.50	2.09	2.19	**2.20**	2.06
Cholera	1.00	1.46	1.94	1.96	1.98	**2.00**
Yang1	1.00	1.04	1.19	**1.24**	1.14	0.95

The results show that batching consistently improves throughput. The best speedups range from 1.24 to 3.28. We also observe that the gains from increasing N saturate quickly: values between 8 and 16 often provide the best performance, while larger values rarely yield much additional improvement.

The Yang1 system again provides an instructive example. As discussed in Section 4.1, tracing eliminates a substantial amount of monomial operations in this benchmark, which have been the bottleneck. Since batching amortizes the same type of work, its additional benefit is comparatively modest.

Remark 3 (Memory consumption). Increasing N also increases memory consumption. For Katsura-13, the maximum RAM usage grows from 2.7 GB at $N = 1$ to 29.3 GB at $N = 32$. As the throughput appears to saturate well before this point, large values of N are unattractive. In particular, `Groebner.jl` internally uses $N = 4$ for multi-modular computation over the rationals.

5 Applications

In this paper, we are using `Groebner.jl` version 0.10.3. All experiments we carried out on an Intel Core i9-13900 running Linux with 128 GB of RAM; the CPU does not support AVX512.

5.1 Polynomial system solving

One of the standard approaches to solving zero-dimensional polynomial systems over the rationals is via rational univariate representations [18]. In modern implementations, the computation is typically performed using multi-modular techniques, requiring many Gröbner basis computations modulo different primes. Our experiments use `RationalUnivariateRepresentation.jl`, which implements the algorithms from [6].

Table 3. Total running times of rational univariate representation computation.

System	Backend: `groebner`	Backend: `groebner_apply`	Backend: `groebner_apply` ($N = 4$)	Speedup
Chandra-10	82.4 s	32.5 s	29.3 s	2.8
Chandra-11	851.4 s	395.5 s	373.5 s	2.3
Eco-10	1.0 s	0.6 s	0.4 s	2.5
Eco-11	11.2 s	6.5 s	4.2 s	2.7
Katsura-11	71.0 s	39.8 s	28.2 s	2.5
SEIR36	63.7 s	11.2 s	9.6 s	6.6
cp_d3_n6_p6	66.4 s	49.0 s	35.9 s	1.8

We compare three methods on several benchmark systems. The first uses `groebner` for all Gröbner basis computations. The second learns a trace from the first modular computation and then reuses it for the remaining primes. The third, shown in the `groebner_apply` ($N = 4$) column, uses the application in batches from Section 3.3 to process four primes at once during the apply stage.

The results show that both tracing and batching can substantially accelerate solving, with the overall speedups range from 1.8 to 6.6.

5.2 Identifiable functions of dynamical models

Structural identifiability is the problem of determining whether the parameters $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_m)$ of a dynamical model can be recovered from the observed data. For models defined by ordinary differential equations, an important task is to compute generators of the field of identifiable functions, a subfield of $\mathbb{Q}(\boldsymbol{\mu})$.

The algorithm of [5], implemented in `StructuralIdentifiability.jl` [7], computes such generators by interpolating selected coefficients of the reduced Gröbner basis of an ideal in $\mathbb{Q}(\boldsymbol{\mu})[\mathbf{x}]$. The workflow repeatedly specializes the parameters $\boldsymbol{\mu}$ and reduces modulo primes, producing a sequence of Gröbner basis computations in $\mathbb{Z}/p\mathbb{Z}[x]$ with fixed monomial structure but varying coefficients. The learn-and-apply strategy is therefore a natural fit.

Table 4. The number of evaluation points and the total running times for computing generators of the field of identifiable functions via rational function interpolation using `groebner` and `groebner_apply`.

Model	Evaluation points	Backend: `groebner`	Backend: `groebner_apply`	Speedup
EAIHRD [9]	420	300 s	49 s	6.1
LinComp2 [1]	4388	275 s	164 s	1.7
Pharm [4]	34	54 s	36 s	1.5

We consider the models EAIHRD [9], LinComp2 [1], and Pharm [4]. For each model, we run `StructuralIdentifiability.jl` and compare a workflow using only `groebner` with the one combining `groebner_learn` and `groebner_apply`; see Table 4. The results confirm that tracing can accelerate this interpolation-based workflow.

References

1. Ahmed, S., Crepeau, N., Dessauer, P.R., Edozie, A., Garcia-Lopez, O., Grimsley, T., Garcia, J.L., Neri, V., Shiu, A.: Identifiability of directed-cycle and catenary linear compartmental models. SIAM Journal on Applied Dynamical Systems **25**(1), 304–350 (2026). `https://doi.org/10.1137/25M1728636`
2. Berthomieu, J., Eder, C., Safey El Din, M.: `msolve`: A library for solving polynomial systems. In: Proceedings of the 2021 International Symposium on Symbolic and Algebraic Computation. pp. 51–58 (2021). `https://doi.org/10.1145/3452143.3465545`

3. Bosma, W., Cannon, J., Playoust, C.: The Magma algebra system. I. The user language. J. Symbolic Comput. **24**(3-4), 235–265 (1997). https://doi.org/10.1006/jsco.1996.0125, http://dx.doi.org/10.1006/jsco.1996.0125, Computational algebra and number theory (London, 1993)
4. Demignot, S., Domurado, D.: Effect of prosthetic sugar groups on the pharmacokinetics of glucose-oxidase. Drug Design and Delivery **1**(4), 333–348 (May 1987), https://pubmed.ncbi.nlm.nih.gov/2855567/
5. Demin, A., Pogudin, G.: Simple generators of rational function fields. arXiv preprint arXiv:2602.10878 (2026), https://arxiv.org/abs/2602.10878
6. Demin, A., Rouillier, F., Ruiz, J.: Reading rational univariate representations on lexicographic Gröbner bases. arXiv preprint arXiv:2402.07141 (2024), https://arxiv.org/abs/2402.07141
7. Dong, R., Goodbrake, C., Harrington, H.A., Pogudin, G.: Differential elimination for dynamical models via projections with applications to structural identifiability. SIAM Journal on Applied Algebra and Geometry **7**(1), 194–235 (2023). https://doi.org/10.1137/22M1469067, https://doi.org/10.1137/22M1469067
8. Faugère, J.C.: A new efficient algorithm for computing Gröbner bases without reduction to zero (F4). Journal of Pure and Applied Algebra **139**(1–3), 61–88 (1999). https://doi.org/10.1016/S0022-4049(99)00005-5
9. Fokas, A., Cuevas-Maraver, J., Kevrekidis, P.: A quantitative framework for exploring exit strategies from the COVID-19 lockdown. Chaos, Solitons & Fractals **140**, 110244 (2020), http://dx.doi.org/10.1016/j.chaos.2020.110244
10. Gowda, S., Demin, A.: Groebner.jl: A package for Gröbner bases computations in Julia. arXiv preprint arXiv:2304.06935 (2023), https://arxiv.org/abs/2304.06935
11. Gräbe, H.G.: On lucky primes. Journal of Symbolic Computation **15**(2), 199–209 (1993), https://www.informatik.uni-leipzig.de/~graebe/ComputerAlgebra/Publications/olp.pdf
12. van der Hoeven, J., Lecerf, G., Quintin, G.: Modular simd arithmetic in mathemagix. ACM Trans. Math. Softw. **43**(1) (Aug 2016). https://doi.org/10.1145/2876503, https://doi.org/10.1145/2876503
13. Joux, A., Vitse, V.: A variant of the F4 algorithm. Cryptology ePrint Archive, Paper 2010/158 (2010), https://eprint.iacr.org/2010/158
14. de Kleine, J., Monagan, M.: A modular design and implementation of Buchberger's algorithm. MS project report, Simon Fraser Univ. (2001)
15. Kouba, R., Neiger, V., Din, M.S.E.: A complexity analysis of the f4 gröbner basis algorithm with tracer data (2026), https://arxiv.org/abs/2603.16378
16. Monagan, M., Pearce, R.: A compact parallel implementation of F4. In: Proceedings of the 2015 International Workshop on Parallel Symbolic Computation. pp. 95–100. PASCO '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2790282.2790293, https://doi.org/10.1145/2790282.2790293
17. Parisse, B., De Graeve, R.: Giac/xcas. http://www-fourier.univ-grenoble-alpes.fr/~parisse/giac.html (2026), version 2.0.0
18. Rouillier, F.: Solving zero-dimensional systems through the rational univariate representation. Applicable Algebra in Engineering, Communication and Computing **9**(5), 433–461 (1999). https://doi.org/10.1007/s002000050114
19. Traverso, C.: Gröbner trace algorithms. In: Gianni, P. (ed.) Symbolic and Algebraic Computation. pp. 125–138. Springer, Berlin, Heidelberg (1989). https://doi.org/10.1007/3-540-51084-2_12