Groebner.jl

The Groebner.jl developers and contributors

September 4, 2025

Contents

Part I

Home

Chapter 1

Home

Groebner.jl is a package for computing Gröbner bases written in Julia.

Note

This documentation is also available in PDF format: Groebner.jl.pdf.

1.1 Installation

To install Groebner.jl, run the following in the Julia REPL:

```
using Pkg; Pkg.add("Groebner")
```

1.2 Features

Groebner.jl features:

- Gröbner basis over integers modulo a prime and over the rationals
- Gröbner trace algorithms
- Multi-threading

1.3 Contacts

This library is maintained by Alexander Demin (asdemin_2@edu.hse.ru).

1.4 Citation

```
@misc{demin2024groebnerjl,
      title={Groebner.jl: A package for Gr\"obner bases computations in Julia},
      author={Alexander Demin and Shashi Gowda},
      year={2024},
      eprint={2304.06935},
      archivePrefix={arXiv},
```

```
      primaryClass={cs.MS}
}
```

1.5 Acknowledgement

We would like to acknowledge the developers of the msolve library (https://msolve.lip6.fr/), as several components of Groebner.jl were adapted from msolve. In our F4 implementation, we adapt and adjust the code of monomial hashtable, critical pair handling and symbolic preprocessing, and linear algebra from msolve. The source code of msolve is available at https://github.com/algebraic-solving/msolve.

We thank Vladimir Kuznetsov for helpful discussions and providing the sources of his F4 implementation.

We are grateful to The Max Planck Institute for Informatics, The MAX team at l'X, and the OURAGAN team at Inria for providing computational resources.

Part II

Examples

Chapter 2

Examples

Groebner.jl supports polynomials from the following frontends:

- AbstractAlgebra.jl
- Nemo.jl
- DynamicPolynomials.jl

Additionally, Groebner.jl has a low-level entry point that accepts raw polynomial data.

2.1 Using AbstractAlgebra.jl

First, we import AbstractAlgebra.jl. Then, we create an array of polynomials over a finite field

```
using AbstractAlgebra

R, (x, y, z) = polynomial_ring(GF(2^31 - 1), ["x", "y", "z"])
polys = [x^2 + y + z, x*y + z];
```

```
2-element Vector{AbstractAlgebra.Generic.MPoly{AbstractAlgebra.GFElem{Int64}}}:
 x^2 + y + z
 x*y + z
```

and compute a Gröbner basis with the `groebner` command

```
using Groebner

basis = groebner(polys)
```

```
4-element Vector{AbstractAlgebra.Generic.MPoly{AbstractAlgebra.GFElem{Int64}}}:
 y^3 + y^2*z + z^2
 x*z + 2147483646*y^2 + 2147483646*y*z
 x*y + z
 x^2 + y + z
```

We can check if a set of polynomials forms a Gröbner basis

```
isgroebner(basis)
```

```
true
```

Groebner.jl also provides several monomial orderings. For example, we can eliminate z from the above system:

```
ordering = Lex(z) * DegRevLex(x, y)  # z > x, y
groebner(polys, ordering=ordering)
```

```
2-element Vector{AbstractAlgebra.Generic.MPoly{AbstractAlgebra.GFElem{Int64}}}:
 x^2 + 2147483646*x*y + y
 x*y + z
```

You can find more information on monomial orderings in Groebner.jl in Monomial orderings.

2.2 Using DynamicPolynomials.jl

Computing the Gröbner basis of some system:

```
using DynamicPolynomials, Groebner

@polyvar x1 x2
system = [10*x1*x2^2 - 11*x1 + 10,
        10*x1^2*x2 - 11*x2 + 10]

groebner(system)
```

```
3-element
↪  Vector{DynamicPolynomials.Polynomial{DynamicPolynomials.Commutative{DynamicPolynomials.CreationOrder},
↪  MultivariatePolynomials.Graded{MultivariatePolynomials.LexOrder}, Rational{BigInt}}}:
 10//11x2 - 10//11x1 - x2² + x1²
 1//1 - 11//10x2 - 10//11x2² + 10//11x1x2 + x2³
 1//1 - 11//10x1 + x1x2²
```

2.3 Using Low-level interface

Some functions in the interface have a low-level entry point. Low-level functions accept and output ''raw'' exponent vectors and coefficients. This could be convenient when one does not want to depend on a frontend.

For example,

```
using Groebner

# define {x * y - 1, x^3 + 7 * y^2} modulo 65537 in DRL
ring = Groebner.PolyRing(2, Groebner.DegRevLex(), 65537)
monoms = [ [[1, 1], [0, 0]], [[3, 0], [0, 2]] ]
coeffs = [ [    1,     -1 ], [    1,      7 ] ]

# compute a GB
gb_monoms, gb_coeffs = Groebner.groebner(ring, monoms, coeffs)
```

```
(Vector{Vector{UInt32}}[[[0x00000001, 0x00000001], [0x00000000, 0x00000000]], [[0x00000000,
↪  0x00000003], [0x00000002, 0x00000000]], [[0x00000003, 0x00000000], [0x00000000, 0x00000002]]],
↪  Vector{UInt32}[[0x00000001, 0x00010000], [0x00000001, 0x00004925], [0x00000001, 0x00000007]])
```

The list of functions that provide a low-level entry point: `groebner`, `normalform`, `isgroebner`, `groebner_learn`, `groebner_apply`.

Low-level functions may be faster than their user-facing analogues since they bypass data conversions. Low-level functions do not make any specific assumptions on input polynomials, that is, all of these cases are correctly handled: unsorted monomials, non-normalized coefficients, duplicate terms, aliasing memory.

2.4 Generic coefficients

The implementation in Groebner.jl uses a generic type for coefficients. Hence, in theory, Groebner.jl can compute Gröbner bases over any type that behaves like a field.

For the following ground fields Groebner.jl runs an efficient native implementation:

- integers modulo a prime,
- rationals numbers.

For other ground fields, a possibly slower generic fallback is used. In this case, coefficients of polynomials are treated as black-boxes which implement field operations: `zero`, `one`, `inv`, `==`, `+`, `*`, `-`.

For example, we can compute a Gröbner basis over a univariate rational function field over a finite field:

```
using Groebner, AbstractAlgebra

R, t = GF(101)["t"]
ff = fraction_field(R)
_, (x, y) = ff["x","y"]

sys = [(t//t+1)*x*y - t^3, y^2 + t]

gb = groebner(sys)
```

```
2-element
↪  Vector{AbstractAlgebra.Generic.MPoly{AbstractAlgebra.Generic.FracFieldElem{AbstractAlgebra.Generic.Poly{AbstractAl
 y^2 + t
 x + 51*t^2*y
```

Many functions reuse the core implementation, so they can also be used over generic fields:

```
@assert isgroebner(gb)
normalform(gb, x*y)
```

Computing over floating point intervals

In the following example, we combine low-level interface and generic coefficients.

We are going to compute a basis of the hexapod system over tuples (Z_p, Interval): each coefficient is treated as a pair, the first coordinate is a finite field element used for zero testing, and the second coordinate is a floating point interval with some fixed precision, the payload. For floating point arithmetic, we will be using MPFI.jl.

```
using Pkg;
Pkg.add(url="https://gitlab.inria.fr/ckatsama/mpfi.jl")

import Base: +, -, *, zero, iszero, one, isone, inv
using AbstractAlgebra, Groebner, MPFI

PRECISION = 1024 # For MPFI intervals

struct Zp_And_FloatInterval{Zp, FloatInterval}
    a::Zp
    b::FloatInterval
end

# Pretend it is a field and hakuna matata
+(x::Zp_And_FloatInterval, y::Zp_And_FloatInterval) = Zp_And_FloatInterval(x.a + y.a, x.b + y.b)
*(x::Zp_And_FloatInterval, y::Zp_And_FloatInterval) = Zp_And_FloatInterval(x.a * y.a, x.b * y.b)
-(x::Zp_And_FloatInterval, y::Zp_And_FloatInterval) = Zp_And_FloatInterval(x.a - y.a, x.b - y.b)
zero(x::Zp_And_FloatInterval) = Zp_And_FloatInterval(zero(x.a), zero(x.b))
one(x::Zp_And_FloatInterval) = Zp_And_FloatInterval(one(x.a), one(x.b))
inv(x::Zp_And_FloatInterval) = Zp_And_FloatInterval(inv(x.a), inv(x.b))
iszero(x::Zp_And_FloatInterval) = iszero(x.a)
isone(x::Zp_And_FloatInterval) = isone(x.a)

@info "Computing Hexapod over QQ"
c_zp = Groebner.Examples.hexapod(k=AbstractAlgebra.GF(2^30+3));
c_qq = Groebner.Examples.hexapod(k=AbstractAlgebra.QQ);
@time gb_truth = groebner(c_qq);
gbcoeffs_truth = map(f -> collect(coefficients(f)), gb_truth);
@info "Coefficient size (in bits): $(maximum(f -> maximum(c -> log2(abs(numerator(c))) +
↪  log2(denominator(c)), f), gbcoeffs_truth))"

@info "Computing Hexapod over (Zp, Interval). Precision = $PRECISION bits"
ring = Groebner.PolyRing(nvars(parent(c_qq[1])), Groebner.DegRevLex(), 0, :generic); # Note
↪  :generic
```

```
exps = map(f -> collect(exponent_vectors(f)), c_zp);
cfs_qq = map(f -> collect(coefficients(f)), c_qq);
cfs_zp = map(f -> collect(coefficients(f)), c_zp);
cfs = map(f -> map(c -> Groebner.CoeffGeneric(Zp_And_FloatInterval(c[1], BigInterval(c[2],
↪  precision=PRECISION))), zip(f...)), zip(cfs_zp, cfs_qq));
@time gbexps, gbcoeffs = groebner(ring, exps, cfs);

to_inspect = gbcoeffs[end][end]
@info "
    Inspect one coefficient in the basis:
    Zp         = $(to_inspect.data.a)
    Interval   = $(to_inspect.data.b)
    Diam       = $(diam(to_inspect.data.b))
    Diam (rel) = $(diam_rel(to_inspect.data.b))"

# Sanity check
all_are_inside(x::Zp_And_FloatInterval, truth) = is_inside(BigInterval(truth; precision=PRECISION),
↪  x.b)
all_are_inside(x::Groebner.CoeffGeneric, truth) = all_are_inside(x.data, truth)
all_are_inside(x::AbstractVector, truth) = all(map(all_are_inside, x, truth))
@assert all_are_inside(gbcoeffs, gbcoeffs_truth)

# Max |midpoint - truth|
max_error(x::Zp_And_FloatInterval, y; rel=false) = abs(mid(x.b) - y) / ifelse(rel, max(abs(y), 0),
↪  1)
max_error(x::Groebner.CoeffGeneric, y; rel=false) = max_error(x.data, y; rel=rel)
max_error(x::AbstractVector, y::AbstractVector; rel=false) = maximum(map(f -> max_error(f...;
↪  rel=rel), zip(x, y)))
@info "
    Max error      : $(max_error(gbcoeffs, gbcoeffs_truth))
    Max error (rel): $(max_error(gbcoeffs, gbcoeffs_truth; rel=true))"

# Max diameter
max_diam(x::Zp_And_FloatInterval; rel=false) = ifelse(rel, diam_rel(x.b), diam(x.b))
max_diam(x::Groebner.CoeffGeneric; rel=false) = max_diam(x.data; rel=rel)
max_diam(x::AbstractVector; rel=false) = maximum(map(f -> max_diam(f; rel=rel), x))
@info "
    Max diam      : $(max_diam(gbcoeffs))
    Max diam (rel): $(max_diam(gbcoeffs; rel=true))"
```

```
    Updating git-repo `https://gitlab.inria.fr/ckatsama/mpfi.jl`
   Resolving package versions...
  No Changes to `~/work/Groebner.jl/Groebner.jl/docs/Project.toml`
  No Changes to `~/work/Groebner.jl/Groebner.jl/docs/Manifest.toml`
[ Info: Computing Hexapod over QQ
  3.125505 seconds (2.15 M allocations: 326.378 MiB, 1.41% gc time)
[ Info: Coefficient size (in bits):
↪  26417.5122166029034596333893085177067171186322390155015446773007503040818673238 3
[ Info: Computing Hexapod over (Zp, Interval). Precision = 1024 bits
  3.463183 seconds (11.60 M allocations: 1.143 GiB, 3.90% gc time, 59.34% compilation time)
┌ Info:
│     Inspect one coefficient in the basis:
│     Zp         = 431552538
│     Interval   =
↪  [-558257.17996429349910325466588465107213351577509565990822936464755115027493270119312246422553692665255473494522 1
↪  -558257.17996429349910325466588465107213351577509565990822936464755115027493270119312246422553692665255473494522 19
```

```
│     Diam       =
↪  1.67202715891965197600686931011412241966390077343620455000046015167079584015042991423390737526928542546780995239585
└     Diam (rel) =
↪  1.67202715891965197600686931011412241966390077343620455000046015167079584015042991423390737526928542546780995239585
┌ Info:
│     Max error      :
↪  1.22886432627974213343406034241021464975373087889084813061548048566280994147211le-51
└     Max error (rel):
↪  2.12784979150975703688718260738123517736109881415212621782083160189710558530116e-77
┌ Info:
│     Max diam      :
↪  1.20246021137705299168427973026298928069512267664693371933319893847333822066624300591120890725448959981538999160560
└     Max diam (rel):
↪  1.20246021137705299168427973026298928069512267664693371933319893847333822066624300591120890725448959981538999160560
```

However, if we lower MPFI precision to 256 bits, some of the intervals become NaN.

Part III

Interface

Chapter 3

Interface

3.1 Main functions

`Groebner.groebner` – Function.

```
groebner(polynomials; options...)
```

Computes a Groebner basis of the ideal generated by `polynomials`.

Arguments

- `polynomials`: an array of polynomials. Supports polynomials from AbstractAlgebra.jl, Nemo.jl, and DynamicPolynomials.jl.

Returns

- `basis`: an array of polynomials, a Groebner basis.

Possible Options

- `reduced`: A bool, if the returned basis must be autoreduced and unique. Default is `true`.
- `ordering`: Specifies the monomial ordering. Available monomial orderings are:
  - `InputOrdering()` for inferring the ordering from the given `polynomials` (default),
  - `Lex(args...)` for lexicographic,
  - `DegLex(args...)` for degree lexicographic,
  - `DegRevLex(args...)` for degree reverse lexicographic,
  - `WeightedOrdering(args...)` for weighted ordering,
  - `ProductOrdering(args...)` for block ordering,
  - `MatrixOrdering(args...)` for matrix ordering.

  For details and examples see the corresponding documentation page.
- `certify`: A bool, whether to certify the obtained basis. When this option is `false`, the algorithm is randomized and the result is correct with high probability. When this option is `true`, the result is guaranteed to be correct in case the ideal is homogeneous. Default is `false`.
- `linalg`: A symbol, linear algebra backend. Available options are:

  - `:auto` for the automatic choice (default),
  - `:deterministic` for deterministic sparse linear algebra,
  - `:randomized` for probabilistic sparse linear algebra.
- `threaded`: The use of multi-threading. Available options are:
  - `:auto` for the automatic choice (default),
  - `:no` never use multi-threading,
  - `:yes` allow the use of multi-threading.

  Additionally, it is possible to set the environment variable `GROEBNER_NO_THREADED` to 1 to disable all multi-threading in Groebner.jl. In this case, the environment variable takes precedence over the `threaded` option.
- `monoms`: Monomial representation used in the computations. Available options are:
  - `:auto` for the automatic choice (default),
  - `:dense` for classic dense exponent vectors,
  - `:packed` for packed representation.
- `modular`: Modular computation algorithm. Only has effect when computing basis over rational numbers. Available options are:
  - `:auto` for the automatic choice (default),
  - `:classic_modular` for the classic multi-modular algorithm,
  - `:learn_and_apply` for the learn & apply algorithm.
- `seed`: The seed for randomization. Default is 42.
- `homogenize`: Controls the use of homogenization in the algorithm. Available options are:
  - `:auto`, for the automatic choice (default).
  - `:yes`, always homogenize the input ideal,
  - `:no`, never homogenize the input ideal,

Example

Using DynamicPolynomials.jl:

```
using Groebner, DynamicPolynomials
@polyvar x y
groebner([x*y^2 + x, y*x^2 + y])
```

Using AbstractAlgebra.jl:

```
using Groebner, AbstractAlgebra
R, (x, y) = QQ["x", "y"]
groebner([x*y^2 + x, y*x^2 + y])
```

Using Nemo.jl:

```
using Groebner, Nemo
R, (x, y) = GF(2^30+3)["x", "y"]
groebner([x*y^2 + x, y*x^2 + y])
```

Or, say, in another monomial ordering:

```
# lex with y > x
groebner([x*y^2 + x, y*x^2 + y], ordering=Lex(y, x))

# degree reverse lexicographic
groebner([x*y^2 + x, y*x^2 + y], ordering=DegRevLex())
```

Notes

- The function is thread-safe.
- For AbstractAlgebra.jl and Nemo.jl, the function is most efficient for polynomials over `GF(p)`, `Native.GF(p)`, and `QQ`.
- The default algorithm is probabilistic (with `certify=false`). Results are correct with high probability, however, no precise bound on the probability is known.

source

`Groebner.isgroebner` – Function.

```
isgroebner(polynomials; options...)
```

Checks if `polynomials` forms a Groebner basis.

Arguments

- `polynomials`: an array of polynomials. Supports polynomials from AbstractAlgebra.jl, Nemo.jl, and DynamicPolynomials.jl.

Returns

- `flag`: a bool, whether `polynomials` is a Groebner basis of the ideal generated by `polynomials`.

Possible Options

- `ordering`: Specifies the monomial ordering. Available monomial orderings are:
  - `InputOrdering()` for inferring the ordering from the given `polynomials` (default),
  - `Lex()` for lexicographic,
  - `DegLex()` for degree lexicographic,
  - `DegRevLex()` for degree reverse lexicographic,
  - `WeightedOrdering(weights)` for weighted ordering,
  - `ProductOrdering(args...)` for block ordering,
  - `MatrixOrdering(matrix)` for matrix ordering.

  For details and examples see the corresponding documentation page.
- `certify`: a bool, whether to use a deterministic algorithm. Default is `false`.
- `seed`: The seed for randomization. Default value is 42.

Example

Using `DynamicPolynomials`:

```
using Groebner, DynamicPolynomials
@polyvar x y;
isgroebner([x*y^2 + x, y*x^2 + y])
```

Using `AbstractAlgebra`:

```
using Groebner, AbstractAlgebra
R, (x, y) = QQ["x", "y"]
isgroebner([x*y^2 + x, y*x^2 + y])
```

Notes

- The function is thread-safe.
- For AbstractAlgebra.jl and Nemo.jl, the function is most efficient for polynomials over `GF(p)`, `Native.GF(p)`, and `QQ`.
- The default algorithm is probabilistic (with `certify=false`). Results are correct with high probability, however, no precise bound on the probability is known.

source

`Groebner.normalform` – Function.

```
normalform(basis, to_be_reduced; options...)
```

Computes the normal form of polynomials `to_be_reduced` with respect to a Groebner basis `basis`.

Arguments

- `basis`: an array of polynomials, a Groebner basis. Supports polynomials from AbstractAlgebra.jl, Nemo.jl, and DynamicPolynomials.jl.
- `to_be_reduced`: either a single polynomial or an array of polynomials. Supports polynomials from AbstractAlgebra.jl, Nemo.jl, and DynamicPolynomials.jl.

Returns

- `reduced`: either a single polynomial or an array of polynomials, the normal forms.

Possible Options

- `check`: Check if `basis` forms a Groebner basis. Default is `true`.
- `ordering`: Specifies the monomial ordering. Available monomial orderings are:
  - `InputOrdering()` for inferring the ordering from the given `polynomials` (default),
  - `Lex()` for lexicographic,
  - `DegLex()` for degree lexicographic,
  - `DegRevLex()` for degree reverse lexicographic,
  - `WeightedOrdering(weights)` for weighted ordering,
  - `ProductOrdering(args...)` for block ordering,

- `MatrixOrdering(matrix)` for matrix ordering.

For details and examples see the corresponding documentation page.

Example

Fining the normal form a single polynomial:

```
using Groebner, DynamicPolynomials
@polyvar x y;
normalform([y^2 + x, x^2 + y], x^2 + y^2 + 1)
```

Or, reducing two polynomials at a time:

```
using Groebner, DynamicPolynomials
@polyvar x y;
normalform([y^2 + x, x^2 + y], [x^2 + y^2 + 1, x^10*y^10])
```

Notes

- The function is thread-safe.
- For AbstractAlgebra.jl and Nemo.jl, the function is most efficient for polynomials over `GF(p)`, `Native.GF(p)`, and `QQ`.
- The default algorithm is probabilistic (with `certify=false`). Results are correct with high probability, however, no precise bound on the probability is known.

source

`Groebner.leading_ideal` – Function.

```
leading_ideal(polynomials; options...)
```

Returns generators of the ideal of the leading terms.

If the input is not a Groebner basis, computes a Groebner basis.

Arguments

- `polynomials`: an array of polynomials. Supports polynomials from AbstractAlgebra.jl, Nemo.jl, and DynamicPolynomials.jl.

Returns

- `basis`: the basis of the ideal of the leading terms.

Possible Options

- `ordering`: Specifies the monomial ordering. Available monomial orderings are:
  - `InputOrdering()` for inferring the ordering from the given `polynomials` (default),
  - `Lex()` for lexicographic,

- `DegLex()` for degree lexicographic,
- `DegRevLex()` for degree reverse lexicographic,
- `WeightedOrdering(weights)` for weighted ordering,
- `ProductOrdering(args...)` for block ordering,
- `MatrixOrdering(matrix)` for matrix ordering.

For details and examples see the corresponding documentation page.

Example

Using AbstractAlgebra.jl:

```
using Groebner, Nemo
R, (x, y) = QQ["x", "y"]
leading_ideal([x*y^2 + x, y*x^2 + y])
```

Notes

- The function is thread-safe.
- For AbstractAlgebra.jl and Nemo.jl, the function is most efficient for polynomials over `GF(p)`, `Native.GF(p)`, and `QQ`.

source

`Groebner.dimension` – Function.

```
dimension(polynomials; options...)
```

Computes the (Krull) dimension of the ideal generated by `polynomials`.

If input is not a Groebner basis, computes a Groebner basis.

Arguments

- `polynomials`: an array of polynomials. Supports polynomials from AbstractAlgebra.jl, Nemo.jl, and DynamicPolynomials.jl.

Returns

- `dimension`: an integer, the dimension.

Example

Using AbstractAlgebra.jl:

```
using Groebner, Nemo
R, (x, y) = QQ["x", "y"]
dimension([x*y^2 + x, y*x^2 + y])
```

Notes

- The function is thread-safe.
- For AbstractAlgebra.jl and Nemo.jl, the function is most efficient for polynomials over `GF(p)`, `Native.GF(p)`, and `QQ`.

source

`Groebner.quotient_basis` – Function.

```
quotient_basis(polynomials; options...)
```

Returns a monomial basis of the quotient algebra of a zero-dimensional ideal.

If the input is not a Groebner basis, computes a Groebner basis. If the input is not a zero-dimensional ideal, an error is raised.

Arguments

- `polynomials`: an array of polynomials. Supports polynomials from AbstractAlgebra.jl, Nemo.jl, and DynamicPolynomials.jl.

Returns

- `basis`: an array of monomials, a quotient basis.

Possible Options

- `ordering`: Specifies the monomial ordering. Available monomial orderings are:
  - `InputOrdering()` for inferring the ordering from the given `polynomials` (default),
  - `Lex()` for lexicographic,
  - `DegLex()` for degree lexicographic,
  - `DegRevLex()` for degree reverse lexicographic,
  - `WeightedOrdering(weights)` for weighted ordering,
  - `ProductOrdering(args...)` for block ordering,
  - `MatrixOrdering(matrix)` for matrix ordering.

  For details and examples see the corresponding documentation page.

Example

Using AbstractAlgebra.jl:

```
using Groebner, Nemo
R, (x, y) = QQ["x", "y"]
quotient_basis([x*y^2 + x, y*x^2 + y])
```

Notes

- The function is thread-safe.
- For AbstractAlgebra.jl and Nemo.jl, the function is most efficient for polynomials over `GF(p)`, `Native.GF(p)`, and `QQ`.

source

`Groebner.groebner_with_change_matrix` – Function.

```
groebner_with_change_matrix(polynomials; options...)
```

Computes a Groebner basis of the ideal generated by `polynomials` and emits a change matrix, that is, a map from the original generators to basis elements.

Arguments

- `polynomials`: an array of polynomials. Supports polynomials from AbstractAlgebra.jl, Nemo.jl, and DynamicPolynomials.jl. For AbstractAlgebra.jl and Nemo.jl, coefficients of polynomials must belong to `GF(p)`, `Native.GF(p)`, or `QQ`.

Returns

Returns a tuple (`basis`, `matrix`).

- `basis`: an array of polynomials, a Groebner basis.
- `matrix`: a matrix, so that `matrix * polynomials == basis`.

Possible Options

Same as for `groebner`.

Example

Using AbstractAlgebra.jl:

```
using Groebner, AbstractAlgebra
R, (x, y) = QQ["x", "y"]
f = [x*y^2 + x, y*x^2 + y]

g, m = groebner_with_change_matrix(f, ordering=DegRevLex())

@assert isgroebner(g, ordering=DegRevLex())
@assert m * f == g
```

Notes

- Only DegRevLex ordering is supported.
- The function is thread-safe.
- The default algorithm is probabilistic (with `certify=false`). Results are correct with high probability, however, no precise bound on the probability is known.

source

3.2 Monomial orderings

Note

Some frontends, for example, AbstractAlgebra.jl, may not support weighted/product/matrix orderings from Groebner.jl. In such cases, the basis is computed in the ordering requested by user, but the terms of polynomials in the output are ordered w.r.t. some other ordering that is supported by the frontend.

`Groebner.Lex` – Type.

```
Lex()
Lex(variables)
Lex(variables...)
```

Lexicographical monomial ordering.

We use the definition from Chapter 1, Computational Commutative Algebra 1, by Martin Kreuzer, Lorenzo Robbiano. DOI: https://doi.org/10.1007/978-3-540-70628-1.

Dura Lex, sed Lex.

Example

```
using Groebner, AbstractAlgebra
R, (x, y) = QQ["x", "y"];

# Lexicographical ordering with x > y
groebner([x*y + x, x + y^2], ordering=Lex())

# Lexicographical ordering with y > x
groebner([x*y + x, x + y^2], ordering=Lex([y, x]))

# Lexicographical ordering with x > y
# Both syntax are allowed -- Lex([...]) and Lex(...)
groebner([x*y + x, x + y^2], ordering=Lex(x, y))
```

source

`Groebner.DegLex` – Type.

```
DegLex()
DegLex(variables)
DegLex(variables...)
```

Degree lexicographical monomial ordering.

We use the definition from Chapter 1, Computational Commutative Algebra 1, by Martin Kreuzer, Lorenzo Robbiano. DOI: https://doi.org/10.1007/978-3-540-70628-1.

Example

```
using Groebner, AbstractAlgebra
R, (x, y) = QQ["x", "y"];

# Degree lexicographical ordering with x > y
groebner([x*y + x, x + y^2], ordering=DegLex())

# Degree lexicographical ordering with y > x
groebner([x*y + x, x + y^2], ordering=DegLex([y, x]))
```

source

`Groebner.DegRevLex` – Type.

```
DegRevLex()
DegRevLex(variables)
DegRevLex(variables...)
```

Degree reverse lexicographical monomial ordering.

We use the definition from Chapter 1, Computational Commutative Algebra 1, by Martin Kreuzer, Lorenzo Robbiano. DOI: https://doi.org/10.1007/978-3-540-70628-1.

Example

```
using Groebner, AbstractAlgebra
R, (x, y) = QQ["x", "y"];

# Degree reverse lexicographical ordering with x > y
groebner([x*y + x, x + y^2], ordering=DegRevLex())

# Degree reverse lexicographical ordering with y > x
groebner([x*y + x, x + y^2], ordering=DegRevLex(y, x))
```

source

`Groebner.InputOrdering` – Type.

```
InputOrdering()
```

Preserves the monomial ordering defined on the input polynomials.

This is the default value for the `ordering` keyword argument.

Example

```
using Groebner, AbstractAlgebra
R, (x, y) = QQ["x", "y"]

# Uses the ordering `InputOrdering`, which, in this case,
# defaults to the lexicographical ordering with x > y
groebner([x*y + x, x + y^2])
```

source

`Groebner.WeightedOrdering` – Type.

```
WeightedOrdering(weights)
```

Weighted monomial ordering.

Only positive weights are supported.

Example

```
using Groebner, AbstractAlgebra
R, (x, y) = QQ["x", "y"];

# x has weight 3, y has weight 1
ord = WeightedOrdering(x => 3, y => 1)
groebner([x*y + x, x + y^2], ordering=ord)
```

source

`Groebner.ProductOrdering` – Type.

```
ProductOrdering(ord1, ord2)
```

Product monomial ordering. Compares by `ord1`, breaks ties by `ord2`.

Can also be constructed with *.

Example

```
using Groebner, AbstractAlgebra
R, (x, y, z, w) = QQ["x", "y", "z", "w"];

# Ordering with x, y > w, z
ord = ProductOrdering(DegRevLex(x, y), DegRevLex(w, z))
groebner([x*y + w, y*z - w], ordering=ord)
```

It is possible to use the * operator:

```
using Groebner, AbstractAlgebra
R, (x, y, z, t) = QQ["x", "y", "z", "t"];

ord1 = Lex(t)
ord2 = DegRevLex(x, y, z)
# t >> x, y, z
ord = ord1 * ord2
groebner([x*y*z + z, t * z - 1], ordering=ord)
```

source

`Groebner.MatrixOrdering` – Type.

```
MatrixOrdering(matrix)
MatrixOrdering(Vector{Vector})
```

Matrix monomial ordering.

Example

```
using Groebner, AbstractAlgebra
R, (x, y, z, w) = QQ["x", "y", "z", "w"];

# the number of columns equal to the number of variables
ord = MatrixOrdering(
    [x,y,z,w],
    [
    1 0  0  2;
    0 0  1  2;
    0 1  1  1;
    ])
groebner([x*y + w, y*z - w], ordering=ord)
```

source

3.3 Learn and Apply

`Groebner.groebner_learn` – Function.

```
groebner_learn(polynomials; options...)
```

Computes a Groebner basis of the ideal generated by `polynomials` and emits a trace.

The trace can be used to speed up the computation of Groebner bases of specializations of the same ideal as the one `groebner_learn` had been applied to.

See also `groebner_apply!`.

Arguments

- `polynomials`: an array of polynomials. Must be polynomials from AbstractAlgebra.jl or Nemo.jl over `GF(p)` or `Native.GF(p)`.

Returns

Returns a tuple (`trace`, `basis`).

- `trace`: an object, a trace. Can be used in `groebner_apply!`.
- `basis`: an array of polynomials, a Groebner basis.

Possible Options

Same as for `groebner`.

Example

Using `groebner_learn` and `groebner_apply!` over the same ground field:

```
using Groebner, AbstractAlgebra
R, (x, y) = GF(2^31-1)["x", "y"]

# Learn
trace, gb_1 = groebner_learn([x*y^2 + x, y*x^2 + y])

# Apply (same support, different coefficients)
flag, gb_2 = groebner_apply!(trace, [2x*y^2 + 3x, 4y*x^2 + 5y])

@assert flag
```

Using `groebner_learn` and `groebner_apply!` over different ground fields:

```
using Groebner, AbstractAlgebra
R, (x, y) = GF(2^31-1)["x", "y"]

# Learn
trace, gb_1 = groebner_learn([x*y^2 + x, y*x^2 + y], ordering=DegRevLex())

# Create a ring with a different modulo
R2, (x2, y2) = GF(2^30+3)["x", "y"]

# Apply (different modulo)
flag, gb_2 = groebner_apply!(
    trace,
    [2x2*y2^2 + 3x2, 4y2*x2^2 + 5y2],
    ordering=DegRevLex()
)

@assert flag
@assert gb_2 == groebner([2x2*y2^2 + 3x2, 4y2*x2^2 + 5y2], ordering=DegRevLex())
```

Using `groebner_apply!` in batches:

```
using Groebner, AbstractAlgebra
R, (x, y) = polynomial_ring(GF(2^31-1), ["x", "y"], internal_ordering=:degrevlex)

# Learn
trace, gb_1 = groebner_learn([x*y^2 + x, y*x^2 + y])

# Create rings with some other moduli
R2, (x2, y2) = polynomial_ring(GF(2^30+3), ["x", "y"], internal_ordering=:degrevlex)
R3, (x3, y3) = polynomial_ring(GF(2^27+29), ["x", "y"], internal_ordering=:degrevlex)

# Two specializations of the same ideal
batch = ([2x2*y2^2 + 3x2, 4y2*x2^2 + 5y2], [4x3*y3^2 + 4x3, 5y3*x3^2 + 7y3])
```

```
# Apply for two sets of polynomials at once
flag, (gb_2, gb_3) = groebner_apply!(trace, batch)

@assert flag
@assert (gb_2, gb_3) == map(groebner, batch)
```

Perhaps, in a more involved example, we will compute Groebner bases of the Katsura-9 system:

```
using Groebner, AbstractAlgebra, BenchmarkTools

# Create the system
kat = Groebner.Examples.katsuran(9, k=ZZ, internal_ordering=:degrevlex)

# Reduce the coefficients modulo 5 different primes
kat_0 = map(f -> map_coefficients(c -> GF(2^30 + 3)(c), f), kat)
kat_1 = map(f -> map_coefficients(c -> GF(2^30 + 7)(c), f), kat)
kat_2 = map(f -> map_coefficients(c -> GF(2^30 + 9)(c), f), kat)
kat_3 = map(f -> map_coefficients(c -> GF(2^30 + 15)(c), f), kat)
kat_4 = map(f -> map_coefficients(c -> GF(2^30 + 19)(c), f), kat)

# Learn the trace
trace, gb_0 = groebner_learn(kat_0);

# Compare the performance of applying with 1 input and with 4 different inputs:

# Apply for one system
@btime groebner_apply!($trace, $kat_1);
#  46.824 ms (19260 allocations: 24.48 MiB)

# Apply for a batch of four systems
@btime groebner_apply!($trace, $(kat_1, kat_2, kat_3, kat_4));
#  72.813 ms (23722 allocations: 59.44 MiB)
```

Observe the better amortized performance of the composite `groebner_apply!`.

Notes

- The function is thread-safe.

source

`Groebner.groebner_apply!` – Function.

```
groebner_apply!(trace, polynomials; options...)
groebner_apply!(trace, batch::NTuple{N, Vector}; options...)
```

Computes a Groebner basis of the ideal generated by `polynomials` following the given `trace`.

See also `groebner_learn`.

Arguments

- `trace`: a trace produced by `groebner_learn`.

- `polynomials`: an array of polynomials. Must be polynomials from AbstractAlgebra.jl or Nemo.jl over `GF(p)` or `Nemo.GF(p)`. It is possible to supply a tuple of `N` arrays of polynomials to compute `N` Groebner bases simultaneously. This could be more efficient overall than computing them in separate.

Returns

Returns a tuple (`success`, `basis`).

- `success`: a bool, whether the call succeeded.
- `basis`: an array of polynomials, a Groebner basis.

Possible Options

The `groebner_apply!` function automatically inherits most parameters from the given `trace`.

Example

For examples, see the documentation of `groebner_learn`.

Notes

- In general, `success` may be a false positive. The probability of a false positive is considered to be low enough in some practical applications.
- This function is **not** thread-safe since it mutates `trace`.
- This function is **not** safe against program interruptions. For example, pressing `Ctrl + C` while `groebner_apply!(trace, ...)` is running may leave `trace` corrupted.

source