References

Bradford, Russell J., and James H. Davenport. "Effective tests for cyclotomic polynomials." In International Symposium on Symbolic and Algebraic Computation, pp. 244-251. Springer, Berlin, Heidelberg, 1988.

sympy.polys.factortools.**dup_zz_cyclotomic_poly**(*n*, *K*)

Efficiently generate n-th cyclotomic polynomial.

sympy.polys.factortools.**dup_zz_cyclotomic_factor**(*f*, *K*)

Efficiently factor polynomials $x**n - 1$ and $x**n + 1$ in $Z[x]$.

Given a univariate polynomial f in $Z[x]$ returns a list of factors of f, provided that f is in the form $x**n - 1$ or $x**n + 1$ for $n >= 1$. Otherwise returns None.

Factorization is performed using cyclotomic decomposition of f, which makes this method much faster that any other direct factorization approach (e.g. Zassenhaus's).

References

[R707]

sympy.polys.factortools.**dup_zz_factor_sqf**(*f*, *K*)

Factor square-free (non-primitive) polynomials in $Z[x]$.

sympy.polys.factortools.**dup_zz_factor**(*f*, *K*)

Factor (non square-free) polynomials in $Z[x]$.

Given a univariate polynomial f in $Z[x]$ computes its complete factorization $f_1, ..., f_n$ into irreducibles over integers:

```
f = content(f) f_1**k_1 ... f_n**k_n
```

The factorization is computed by reducing the input polynomial into a primitive square-free polynomial and factoring it using Zassenhaus algorithm. Trial division is used to recover the multiplicities of factors.

The result is returned as a tuple consisting of:

```
(content(f), [(f_1, k_1), ..., (f_n, k_n))
```

Examples

Consider the polynomial $f = 2 * x * *4 - 2$:

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> R.dup_zz_factor(2*x**4 - 2)
(2, [(x - 1, 1), (x + 1, 1), (x**2 + 1, 1)])
```

In result we got the following factorization:

```
f = 2 (x - 1) (x + 1) (x**2 + 1)
```

Note that this is a complete factorization over integers, however over Gaussian integers we can factor the last term.

By default, polynomials $x**n-1$ and $x**n+1$ are factored using cyclotomic decomposition to speedup computations. To disable this behaviour set cyclotomic=False.

References

[R708]

sympy.polys.factortools.**dmp_zz_wang_non_divisors**(*E*, *cs*, *ct*, *K*)

Wang/EEZ: Compute a set of valid divisors.

sympy.polys.factortools.**dmp_zz_wang_test_points**(*f*, *T*, *ct*, *A*, *u*, *K*)

Wang/EEZ: Test evaluation points for suitability.

sympy.polys.factortools.**dmp_zz_wang_lead_coeffs**(*f*, *T*, *cs*, *E*, *H*, *A*, *u*, *K*)

Wang/EEZ: Compute correct leading coefficients.

sympy.polys.factortools.**dmp_zz_diophantine**(*F*, *c*, *A*, *d*, *p*, *u*, *K*)

Wang/EEZ: Solve multivariate Diophantine equations.

sympy.polys.factortools.**dmp_zz_wang_hensel_lifting**(*f*, *H*, *LC*, *A*, *p*, *u*, *K*)

Wang/EEZ: Parallel Hensel lifting algorithm.

sympy.polys.factortools.**dmp_zz_wang**(*f*, *u*, *K*, *mod=None*, *seed=None*)

Factor primitive square-free polynomials in $Z[X]$.

Given a multivariate polynomial f in $Z[x_1, ..., x_n]$, which is primitive and square-free in x_1, computes factorization of f into irreducibles over integers.

The procedure is based on Wang's Enhanced Extended Zassenhaus algorithm. The algorithm works by viewing f as a univariate polynomial in $Z[x_2, ..., x_n][x_1]$, for which an evaluation mapping is computed:

```
x_2 -> a_2, ..., x_n -> a_n
```

where a_i, for $i = 2, \ldots, n$, are carefully chosen integers. The mapping is used to transform f into a univariate polynomial in $Z[x_1]$, which can be factored efficiently using Zassenhaus algorithm. The last step is to lift univariate factors to obtain true multivariate factors. For this purpose a parallel Hensel lifting procedure is used.

The parameter seed is passed to _randint and can be used to seed randint (when an integer) or (for testing purposes) can be a sequence of numbers.

References

[R709], [R710]

sympy.polys.factortools.**dmp_zz_factor**(*f*, *u*, *K*)

Factor (non square-free) polynomials in $Z[X]$.

Given a multivariate polynomial f in $Z[x]$ computes its complete factorization $f_1, \ldots, f_n$ into irreducibles over integers:

```
f = content(f) f_1**k_1 ... f_n**k_n
```

The factorization is computed by reducing the input polynomial into a primitive square-free polynomial and factoring it using Enhanced Extended Zassenhaus (EEZ) algorithm. Trial division is used to recover the multiplicities of factors.

The result is returned as a tuple consisting of:

```
(content(f), [(f_1, k_1), ..., (f_n, k_n))
```

Consider polynomial $f = 2 * (x * *2 - y * *2)$:

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_zz_factor(2*x**2 - 2*y**2)
(2, [(x - y, 1), (x + y, 1)])
```

In result we got the following factorization:

```
f = 2 (x - y) (x + y)
```

References

[R711]

sympy.polys.factortools.**dmp_ext_factor**(*f*, *u*, *K*)
Factor multivariate polynomials over algebraic number fields.

sympy.polys.factortools.**dup_gf_factor**(*f*, *K*)
Factor univariate polynomials over finite fields.

sympy.polys.factortools.**dmp_factor_list**(*f*, *u*, *K0*)
Factor multivariate polynomials into irreducibles in $K[X]$.

sympy.polys.factortools.**dmp_factor_list_include**(*f*, *u*, *K*)
Factor multivariate polynomials into irreducibles in $K[X]$.

sympy.polys.factortools.**dmp_irreducible_p**(*f*, *u*, *K*)
Returns True if a multivariate polynomial f has no factors over its domain.

Groebner basis algorithms

Groebner bases can be used to answer many problems in computational commutative algebra. Their computation in rather complicated, and very performance-sensitive. We present here various low-level implementations of Groebner basis computation algorithms; please see the previous section of the manual for usage.

sympy.polys.groebnertools.**groebner**(*seq*, *ring*, *method=None*)
Computes Groebner basis for a set of polynomials in $K[X]$.

Wrapper around the (default) improved Buchberger and the other algorithms for computing Groebner bases. The choice of algorithm can be changed via `method` argument or *sympy.polys.polyconfig.setup()* (page 2643), where `method` can be either `buchberger` or `f5b`.

`sympy.polys.groebnertools.`**`spoly`**`(`*p1, p2, ring*`)`

Compute LCM(LM(p1), LM(p2))/LM(p1)*p1 - LCM(LM(p1), LM(p2))/LM(p2)*p2 This is the S-poly provided p1 and p2 are monic

`sympy.polys.groebnertools.`**`red_groebner`**`(`*G, ring*`)`

Compute reduced Groebner basis, from BeckerWeispfenning93, p. 216

Selects a subset of generators, that already generate the ideal and computes a reduced Groebner basis for them.

`sympy.polys.groebnertools.`**`is_groebner`**`(`*G, ring*`)`

Check if G is a Groebner basis.

`sympy.polys.groebnertools.`**`is_minimal`**`(`*G, ring*`)`

Checks if G is a minimal Groebner basis.

`sympy.polys.groebnertools.`**`is_reduced`**`(`*G, ring*`)`

Checks if G is a reduced Groebner basis.

`sympy.polys.fglmtools.`**`matrix_fglm`**`(`*F, ring, O_to*`)`

Converts the reduced Groebner basis `F` of a zero-dimensional ideal w.r.t. `O_from` to a reduced Groebner basis w.r.t. `O_to`.

References

[R712]

Groebner basis algorithms for modules are also provided:

`sympy.polys.distributedmodules.`**`sdm_spoly`**`(`*f, g, O, K, phantom=None*`)`

Compute the generalized s-polynomial of `f` and `g`.

The ground field is assumed to be `K`, and monomials ordered according to `O`.

This is invalid if either of `f` or `g` is zero.

If the leading terms of f and g involve different basis elements of F, their s-poly is defined to be zero. Otherwise it is a certain linear combination of f and g in which the leading terms cancel. See [SCA, defn 2.3.6] for details.

If `phantom` is not `None`, it should be a pair of module elements on which to perform the same operation(s) as on `f` and `g`. The in this case both results are returned.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_spoly
>>> from sympy.polys import QQ, lex
>>> f = [((2, 1, 1), QQ(1)), ((1, 0, 1), QQ(1))]
>>> g = [((2, 3, 0), QQ(1))]
>>> h = [((1, 2, 3), QQ(1))]
>>> sdm_spoly(f, h, lex, QQ)
[]
>>> sdm_spoly(f, g, lex, QQ)
[((1, 2, 1), 1)]
```

sympy.polys.distributedmodules.**sdm_ecart**(*f*)

Compute the ecart of f.

This is defined to be the difference of the total degree of f and the total degree of the leading monomial of f [SCA, defn 2.3.7].

Invalid if f is zero.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_ecart
>>> sdm_ecart([((1, 2, 3), 1), ((1, 0, 1), 1)])
0
>>> sdm_ecart([((2, 2, 1), 1), ((1, 5, 1), 1)])
3
```

sympy.polys.distributedmodules.**sdm_nf_mora**(*f, G, O, K, phantom=None*)

Compute a weak normal form of f with respect to G and order O.

The ground field is assumed to be K, and monomials ordered according to O.

Weak normal forms are defined in [SCA, defn 2.3.3]. They are not unique. This function deterministically computes a weak normal form, depending on the order of G.

The most important property of a weak normal form is the following: if R is the ring associated with the monomial ordering (if the ordering is global, we just have $R = K[x_1, \ldots, x_n]$, otherwise it is a certain localization thereof), I any ideal of R and G a standard basis for I, then for any $f \in R$, we have $f \in I$ if and only if $NF(f|G) = 0$.

This is the generalized Mora algorithm for computing weak normal forms with respect to arbitrary monomial orders [SCA, algorithm 2.3.9].

If phantom is not None, it should be a pair of "phantom" arguments on which to perform the same computations as on f, G, both results are then returned.

sympy.polys.distributedmodules.**sdm_groebner**(*G, NF, O, K, extended=False*)

Compute a minimal standard basis of G with respect to order O.

The algorithm uses a normal form NF, for example sdm_nf_mora. The ground field is assumed to be K, and monomials ordered according to O.

Let N denote the submodule generated by elements of G. A standard basis for N is a subset S of N, such that $in(S) = in(N)$, where for any subset X of F, $in(X)$ denotes the submodule generated by the initial forms of elements of X. [SCA, defn 2.3.2]

A standard basis is called minimal if no subset of it is a standard basis.

One may show that standard bases are always generating sets.

Minimal standard bases are not unique. This algorithm computes a deterministic result, depending on the particular order of G.

If `extended=True`, also compute the transition matrix from the initial generators to the groebner basis. That is, return a list of coefficient vectors, expressing the elements of the groebner basis in terms of the elements of `G`.

This functions implements the "sugar" strategy, see

Giovini et al: "One sugar cube, please" OR Selection strategies in Buchberger algorithm.

Options

Options manager for *Poly* (page 2378) and public API functions.

class sympy.polys.polyoptions.**Options**(*gens*, *args*, *flags=None*, *strict=False*)

Options manager for polynomial manipulation module.

Examples

```
>>> from sympy.polys.polyoptions import Options
>>> from sympy.polys.polyoptions import build_options
```

```
>>> from sympy.abc import x, y, z
```

```
>>> Options((x, y, z), {'domain': 'ZZ'})
{'auto': False, 'domain': ZZ, 'gens': (x, y, z)}
```

```
>>> build_options((x, y, z), {'domain': 'ZZ'})
{'auto': False, 'domain': ZZ, 'gens': (x, y, z)}
```

Options

- Expand — boolean option
- Gens — option
- Wrt — option
- Sort — option
- Order — option
- Field — boolean option
- Greedy — boolean option
- Domain — option
- Split — boolean option
- Gaussian — boolean option
- Extension — option

- Modulus — option
- Symmetric — boolean option
- Strict — boolean option

Flags

- Auto — boolean flag
- Frac — boolean flag
- Formal — boolean flag
- Polys — boolean flag
- Include — boolean flag
- All — boolean flag
- Gen — flag
- Series — boolean flag

clone(*updates={}*)

Clone `self` and update specified options.

`sympy.polys.polyoptions.`**build_options**(*gens*, *args=None*)

Construct options from keyword arguments or … options.

Configuration

Configuration utilities for polynomial manipulation algorithms.

`sympy.polys.polyconfig.`**setup**(*key*, *value=None*)

Assign a value to (or reset) a configuration item.

Exceptions

These are exceptions defined by the polynomials module.

TODO sort and explain

class `sympy.polys.polyerrors.`**BasePolynomialError**

Base class for polynomial related exceptions.

class `sympy.polys.polyerrors.`**ExactQuotientFailed**(*f*, *g*, *dom=None*)

class `sympy.polys.polyerrors.`**OperationNotSupported**(*poly*, *func*)

class `sympy.polys.polyerrors.`**HeuristicGCDFailed**

class `sympy.polys.polyerrors.`**HomomorphismFailed**

class `sympy.polys.polyerrors.`**IsomorphismFailed**

class `sympy.polys.polyerrors.`**ExtraneousFactors**

class `sympy.polys.polyerrors.`**EvaluationFailed**

class sympy.polys.polyerrors.**RefinementFailed**

class sympy.polys.polyerrors.**CoercionFailed**

class sympy.polys.polyerrors.**NotInvertible**

class sympy.polys.polyerrors.**NotReversible**

class sympy.polys.polyerrors.**NotAlgebraic**

class sympy.polys.polyerrors.**DomainError**

class sympy.polys.polyerrors.**PolynomialError**

class sympy.polys.polyerrors.**UnificationFailed**

class sympy.polys.polyerrors.**GeneratorsNeeded**

class sympy.polys.polyerrors.**ComputationFailed**(*func*, *nargs*, *exc*)

class sympy.polys.polyerrors.**GeneratorsError**

class sympy.polys.polyerrors.**UnivariatePolynomialError**

class sympy.polys.polyerrors.**MultivariatePolynomialError**

class sympy.polys.polyerrors.**PolificationFailed**(*opt*, *origs*, *exprs*, *seq=False*)

class sympy.polys.polyerrors.**OptionError**

class sympy.polys.polyerrors.**FlagError**

Reference

Modular GCD

sympy.polys.modulargcd.**modgcd_univariate**(*f*, *g*)

Computes the GCD of two polynomials in $\mathbb{Z}[x]$ using a modular algorithm.

The algorithm computes the GCD of two univariate integer polynomials f and g by computing the GCD in $\mathbb{Z}_p[x]$ for suitable primes p and then reconstructing the coefficients with the Chinese Remainder Theorem. Trial division is only made for candidates which are very likely the desired GCD.

Parameters

f : PolyElement

univariate integer polynomial

g : PolyElement

univariate integer polynomial

Returns

h : PolyElement

GCD of the polynomials f and g

cff : PolyElement

cofactor of f, i.e. $\frac{f}{h}$

cfg : PolyElement

cofactor of g, i.e. $\frac{g}{h}$

Examples

```
>>> from sympy.polys.modulargcd import modgcd_univariate
>>> from sympy.polys import ring, ZZ
```

```
>>> R, x = ring("x", ZZ)
```

```
>>> f = x**5 - 1
>>> g = x - 1
```

```
>>> h, cff, cfg = modgcd_univariate(f, g)
>>> h, cff, cfg
(x - 1, x**4 + x**3 + x**2 + x + 1, 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> f = 6*x**2 - 6
>>> g = 2*x**2 + 4*x + 2
```

```
>>> h, cff, cfg = modgcd_univariate(f, g)
>>> h, cff, cfg
(2*x + 2, 3*x - 3, x + 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```


References

1. [Monagan00]

[Monagan00]

sympy.polys.modulargcd.**modgcd_bivariate**(*f*, *g*)

Computes the GCD of two polynomials in $\mathbb{Z}[x, y]$ using a modular algorithm.

The algorithm computes the GCD of two bivariate integer polynomials f and g by calculating the GCD in $\mathbb{Z}_p[x, y]$ for suitable primes p and then reconstructing the coefficients with the Chinese Remainder Theorem. To compute the bivariate GCD over $\mathbb{Z}_p$, the polynomials $f \bmod p$ and $g \bmod p$ are evaluated at $y = a$ for certain $a \in \mathbb{Z}_p$ and then their univariate GCD in $\mathbb{Z}_p[x]$ is computed. Interpolating those yields the bivariate GCD in

$\mathbb{Z}_p[x, y]$. To verify the result in $\mathbb{Z}[x, y]$, trial division is done, but only for candidates which are very likely the desired GCD.

Parameters

f : PolyElement

bivariate integer polynomial

g : PolyElement

bivariate integer polynomial

Returns

h : PolyElement

GCD of the polynomials f and g

cff : PolyElement

cofactor of f, i.e. $\frac{f}{h}$

cfg : PolyElement

cofactor of g, i.e. $\frac{g}{h}$

Examples

```
>>> from sympy.polys.modulargcd import modgcd_bivariate
>>> from sympy.polys import ring, ZZ
```

```
>>> R, x, y = ring("x, y", ZZ)
```

```
>>> f = x**2 - y**2
>>> g = x**2 + 2*x*y + y**2
```

```
>>> h, cff, cfg = modgcd_bivariate(f, g)
>>> h, cff, cfg
(x + y, x - y, x + y)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> f = x**2*y - x**2 - 4*y + 4
>>> g = x + 2
```

```
>>> h, cff, cfg = modgcd_bivariate(f, g)
>>> h, cff, cfg
(x + 2, x*y - x - 2*y + 2, 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

References

1. [Monagan00]

[Monagan00]

sympy.polys.modulargcd.**modgcd_multivariate**(*f*, *g*)

Compute the GCD of two polynomials in $\mathbb{Z}[x_0, \ldots, x_{k-1}]$ using a modular algorithm.

The algorithm computes the GCD of two multivariate integer polynomials f and g by calculating the GCD in $\mathbb{Z}_p[x_0, \ldots, x_{k-1}]$ for suitable primes p and then reconstructing the coefficients with the Chinese Remainder Theorem. To compute the multivariate GCD over $\mathbb{Z}_p$ the recursive subroutine *_modgcd_multivariate_p()* (page 2648) is used. To verify the result in $\mathbb{Z}[x_0, \ldots, x_{k-1}]$, trial division is done, but only for candidates which are very likely the desired GCD.

Parameters

f : PolyElement

multivariate integer polynomial

g : PolyElement

multivariate integer polynomial

Returns

h : PolyElement

GCD of the polynomials f and g

cff : PolyElement

cofactor of f, i.e. $\frac{f}{h}$

cfg : PolyElement

cofactor of g, i.e. $\frac{g}{h}$

Examples

```
>>> from sympy.polys.modulargcd import modgcd_multivariate
>>> from sympy.polys import ring, ZZ
```

```
>>> R, x, y = ring("x, y", ZZ)
```

```
>>> f = x**2 - y**2
>>> g = x**2 + 2*x*y + y**2
```

```
>>> h, cff, cfg = modgcd_multivariate(f, g)
>>> h, cff, cfg
(x + y, x - y, x + y)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> R, x, y, z = ring("x, y, z", ZZ)
```

```
>>> f = x*z**2 - y*z**2
>>> g = x**2*z + z
```

```
>>> h, cff, cfg = modgcd_multivariate(f, g)
>>> h, cff, cfg
(z, x*z - y*z, x**2 + 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

See also:

_modgcd_multivariate_p (page 2648)

References

1. [Monagan00]
2. [Brown71]

[Monagan00], [Brown71]

sympy.polys.modulargcd.**_modgcd_multivariate_p**(*f, g, p, degbound, contbound*)

Compute the GCD of two polynomials in $\mathbb{Z}_p[x_0, \ldots, x_{k-1}]$.

The algorithm reduces the problem step by step by evaluating the polynomials f and g at $x_{k-1} = a$ for suitable $a \in \mathbb{Z}_p$ and then calls itself recursively to compute the GCD in $\mathbb{Z}_p[x_0, \ldots, x_{k-2}]$. If these recursive calls are successful for enough evaluation points, the GCD in k variables is interpolated, otherwise the algorithm returns None. Every time a GCD or a content is computed, their degrees are compared with the bounds. If a degree greater then the bound is encountered, then the current call returns None and a new evaluation point has to be chosen. If at some point the degree is smaller, the correspondent bound is updated and the algorithm fails.

Parameters

f : PolyElement

multivariate integer polynomial with coefficients in $\mathbb{Z}_p$

g : PolyElement

multivariate integer polynomial with coefficients in $\mathbb{Z}_p$

p : Integer

prime number, modulus of f and g

degbound : list of Integer objects

degbound[i] is an upper bound for the degree of the GCD of f and g in the variable x_i

contbound : list of Integer objects

contbound[i] is an upper bound for the degree of the content of the GCD in $\mathbb{Z}_p[x_i][x_0, \ldots, x_{i-1}]$, contbound[0] is not used can therefore be chosen arbitrarily.

Returns

h : PolyElement

GCD of the polynomials f and g or None

References

1. [Monagan00]
2. [Brown71]

[Monagan00], [Brown71]

sympy.polys.modulargcd.**func_field_modgcd**(*f*, *g*)

Compute the GCD of two polynomials f and g in $\mathbb{Q}(\alpha)[x_0, \ldots, x_{n-1}]$ using a modular algorithm.

The algorithm first computes the primitive associate $\check{m}_\alpha(z)$ of the minimal polynomial m_α in $\mathbb{Z}[z]$ and the primitive associates of f and g in $\mathbb{Z}[x_1, \ldots, x_{n-1}][z]/(\check{m}_\alpha)[x_0]$. Then it computes the GCD in $\mathbb{Q}(x_1, \ldots, x_{n-1})[z]/(m_\alpha(z))[x_0]$. This is done by calculating the GCD in $\mathbb{Z}_p(x_1, \ldots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$ for suitable primes p and then reconstructing the coefficients with the Chinese Remainder Theorem and Rational Reconstuction. The GCD over $\mathbb{Z}_p(x_1, \ldots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$ is computed with a recursive subroutine, which evaluates the polynomials at $x_{n-1} = a$ for suitable evaluation points $a \in \mathbb{Z}_p$ and then calls itself recursively until the ground domain does no longer contain any parameters. For $\mathbb{Z}_p[z]/(\check{m}_\alpha(z))[x_0]$ the Euclidean Algorithm is used. The results of those recursive calls are then interpolated and Rational Function Reconstruction is used to obtain the correct coefficients. The results, both in $\mathbb{Q}(x_1, \ldots, x_{n-1})[z]/(m_\alpha(z))[x_0]$ and $\mathbb{Z}_p(x_1, \ldots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$, are verified by a fraction free trial division.

Apart from the above GCD computation some GCDs in $\mathbb{Q}(\alpha)[x_1, \ldots, x_{n-1}]$ have to be calculated, because treating the polynomials as univariate ones can result in a spurious content of the GCD. For this func_field_modgcd is called recursively.

Parameters

f, g : PolyElement

polynomials in $\mathbb{Q}(\alpha)[x_0, \ldots, x_{n-1}]$

Returns

h : PolyElement

monic GCD of the polynomials f and g

cff : PolyElement

cofactor of f, i.e. $\frac{f}{h}$

cfg : PolyElement

cofactor of g, i.e. $\frac{g}{h}$

Examples

```
>>> from sympy.polys.modulargcd import func_field_modgcd
>>> from sympy.polys import AlgebraicField, QQ, ring
>>> from sympy import sqrt
```

```
>>> A = AlgebraicField(QQ, sqrt(2))
>>> R, x = ring('x', A)
```

```
>>> f = x**2 - 2
>>> g = x + sqrt(2)
```

```
>>> h, cff, cfg = func_field_modgcd(f, g)
```

```
>>> h == x + sqrt(2)
True
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> R, x, y = ring('x, y', A)
```

```
>>> f = x**2 + 2*sqrt(2)*x*y + 2*y**2
>>> g = x + sqrt(2)*y
```

```
>>> h, cff, cfg = func_field_modgcd(f, g)
```

```
>>> h == x + sqrt(2)*y
True
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> f = x + sqrt(2)*y
>>> g = x + y
```

```
>>> h, cff, cfg = func_field_modgcd(f, g)
```

```
>>> h == R.one
True
>>> cff * h == f
True
>>> cfg * h == g
True
```

References

1. [Hoeij04]

[Hoeij04]

Undocumented

Many parts of the polys module are still undocumented, and even where there is documentation it is scarce. Please contribute!

Series Manipulation using Polynomials

Any finite Taylor series, for all practical purposes is, in fact a polynomial. This module makes use of the efficient representation and operations of sparse polynomials for very fast multivariate series manipulations. Typical speedups compared to SymPy's `series` method are in the range 20-100, with the gap widening as the series being handled gets larger.

All the functions expand any given series on some ring specified by the user. Thus, the coefficients of the calculated series depend on the ring being used. For example:

```
>>> from sympy.polys import ring, QQ, RR
>>> from sympy.polys.ring_series import rs_sin
>>> R, x, y = ring('x, y', QQ)
>>> rs_sin(x*y, x, 5)
-1/6*x**3*y**3 + x*y
```

`QQ` stands for the Rational domain. Here all coefficients are rationals. It is recommended to use `QQ` with ring series as it automatically chooses the fastest Rational type.

Similarly, if a Real domain is used:

```
>>> R, x, y = ring('x, y', RR)
>>> rs_sin(x*y, x, 5)
-0.166666666666667*x**3*y**3 + x*y
```

Though the definition of a polynomial limits the use of Polynomial module to Taylor series, we extend it to allow Laurent and even Puiseux series (with fractional exponents):

```
>>> from sympy.polys.ring_series import rs_cos, rs_tan
>>> R, x, y = ring('x, y', QQ)

>>> rs_cos(x + x*y, x, 3)/x**3
-1/2*x**(-1)*y**2 - x**(-1)*y - 1/2*x**(-1) + x**(-3)

>>> rs_tan(x**QQ(2, 5)*y**QQ(1, 2), x, 2)
1/3*x**(6/5)*y**(3/2) + x**(2/5)*y**(1/2)
```

By default, `PolyElement` did not allow non-natural numbers as exponents. It converted a fraction to an integer and raised an error on getting negative exponents. The goal of the `ring series` module is fast series expansion, and not to use the `polys` module. The reason we use it as our backend is simply because it implements a sparse representation and most of the basic functions that we need. However, this default behaviour of polys was limiting for ring series.

Note that there is no such constraint (in having rational exponents) in the data-structure used by polys- dict. Sparse polynomials (PolyElement) use the Python dict to store a polynomial term by term, where a tuple of exponents is the key and the coefficient of that term is the value. There is no reason why we can't have rational values in the dict so as to support rational exponents.

So the approach we took was to modify sparse polys to allow non-natural exponents. And it turned out to be quite simple. We only had to delete the conversion to int of exponents in the __pow__ method of PolyElement. So:

```
>>> x**QQ(3, 4)
x**(3/4)
```

and not 1 as was the case earlier.

Though this change violates the definition of a polynomial, it doesn't break anything yet. Ideally, we shouldn't modify polys in any way. But to have all the series capabilities we want, no other simple way was found. If need be, we can separate the modified part of polys from core polys. It would be great if any other elegant solution is found.

All series returned by the functions of this module are instances of the PolyElement class. To use them with other SymPy types, convert them to Expr:

```
>>> from sympy.polys.ring_series import rs_exp
>>> from sympy.abc import a, b, c
>>> series = rs_exp(x, x, 5)
>>> a + series.as_expr()
a + x**4/24 + x**3/6 + x**2/2 + x + 1
```


rs_series

Direct use of elementary ring series functions does give more control, but is limiting at the same time. Creating an appropriate ring for the desired series expansion and knowing which ring series function to call, are things not everyone might be familiar with.

rs_series is a function that takes an arbitrary Expr and returns its expansion by calling the appropriate ring series functions. The returned series is a polynomial over the simplest (almost) possible ring that does the job. It recursively builds the ring as it parses the given expression, adding generators to the ring when it needs them. Some examples:

```
>>> from sympy.polys.ring_series import rs_series
>>> from sympy.functions.elementary.trigonometric import sin
>>> rs_series(sin(a + b), a, 5)
1/24*sin(b)*a**4 - 1/2*sin(b)*a**2 + sin(b) - 1/6*cos(b)*a**3 + cos(b)*a

>>> rs_series(sin(exp(a*b) + cos(a + c)), a, 2)
-sin(c)*cos(cos(c) + 1)*a + cos(cos(c) + 1)*a*b + sin(cos(c) + 1)

>>> rs_series(sin(a + b)*cos(a + c)*tan(a**2 + b), a, 2)
cos(b)*cos(c)*tan(b)*a - sin(b)*sin(c)*tan(b)*a + sin(b)*cos(c)*tan(b)
```

It can expand complicated multivariate expressions involving multiple functions and most importantly, it does so blazingly fast:

```
>>> %timeit ((sin(a) + cos(a))**10).series(a, 0, 5)
1 loops, best of 3: 1.33 s per loop

>>> %timeit rs_series((sin(a) + cos(a))**10, a, 5)
100 loops, best of 3: 4.13 ms per loop
```

rs_series is over 300 times faster. Given an expression to expand, there is some fixed overhead to parse it. Thus, for larger orders, the speed improvement becomes more prominent:

```
>>> %timeit rs_series((sin(a) + cos(a))**10, a, 100)
10 loops, best of 3: 32.8 ms per loop
```

To figure out the right ring for a given expression, rs_series uses the `sring` function, which in turn uses other functions of `polys`. As explained above, non-natural exponents are not allowed. But the restriction is on exponents and not generators. So, `polys` allows all sorts of symbolic terms as generators to make sure that the exponent is a natural number:

```
>>> from sympy.polys.rings import sring
>>> R, expr = sring(1/a**3 + a**QQ(3, 7)); R
Polynomial ring in 1/a, a**(1/7) over ZZ with lex order
```

In the above example, $1/a$ and $a**(1/7)$ will be treated as completely different atoms. For all practical purposes, we could let $b = 1/a$ and $c = a**(1/7)$ and do the manipulations. Effectively, expressions involving $1/a$ and $a**(1/7)$ (and their powers) will never simplify:

```
>>> expr*R(1/a)
(1/a)**4 + (1/a)*(a**(1/7))**3
```

This leads to similar issues with manipulating Laurent and Puiseux series as faced earlier. Fortunately, this time we have an elegant solution and are able to isolate the `series` and `polys` behaviour from one another. We introduce a boolean flag `series` in the list of allowed `Options` for polynomials (see *sympy.polys.polyoptions.Options* (page 2642)). Thus, when we want `sring` to allow rational exponents we supply a `series=True` flag to `sring`:

```
>>> rs_series(sin(a**QQ(1, 3)), a, 3)
-1/5040*a**(7/3) + 1/120*a**(5/3) - 1/6*a + a**(1/3)
```

Contribute

rs_series is not fully implemented yet. As of now, it supports only multivariate Taylor expansions of expressions involving `sin`, `cos`, `exp` and `tan`. Adding the remaining functions is not at all difficult and they will be gradually added. If you are interested in helping, read the comments in `ring_series.py`. Currently, it does not support Puiseux series (though the elementary functions do). This is expected to be fixed soon.

You can also add more functions to `ring_series.py`. Only elementary functions are supported currently. The long term goal is to replace SymPy's current `series` method with `rs_series`.

Manipulation of power series

Functions in this module carry the prefix rs_, standing for "ring series". They manipulate finite power series in the sparse representation provided by polys.ring.ring.

Elementary functions

sympy.polys.ring_series.**rs_log**(*p, x, prec*)

The Logarithm of p modulo O(x**prec).

Notes

Truncation of integral dx p**-1*d p/dx is used.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_log
>>> R, x = ring('x', QQ)
>>> rs_log(1 + x, x, 8)
1/7*x**7 - 1/6*x**6 + 1/5*x**5 - 1/4*x**4 + 1/3*x**3 - 1/2*x**2 + x
>>> rs_log(x**QQ(3, 2) + 1, x, 5)
1/3*x**(9/2) - 1/2*x**3 + x**(3/2)
```

sympy.polys.ring_series.**rs_LambertW**(*p, x, prec*)

Calculate the series expansion of the principal branch of the Lambert W function.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_LambertW
>>> R, x, y = ring('x, y', QQ)
>>> rs_LambertW(x + x*y, x, 3)
-x**2*y**2 - 2*x**2*y - x**2 + x*y + x
```

See also:

LambertW (page 412)

sympy.polys.ring_series.**rs_exp**(*p, x, prec*)

Exponentiation of a series modulo O(x**prec)

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_exp
>>> R, x = ring('x', QQ)
>>> rs_exp(x**2, x, 7)
1/6*x**6 + 1/2*x**4 + x**2 + 1
```

sympy.polys.ring_series.**rs_atan**(*p*, *x*, *prec*)

The arctangent of a series

Return the series expansion of the atan of p, about 0.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_atan
>>> R, x, y = ring('x, y', QQ)
>>> rs_atan(x + x*y, x, 4)
-1/3*x**3*y**3 - x**3*y**2 - x**3*y - 1/3*x**3 + x*y + x
```

See also:

atan (page 397)

sympy.polys.ring_series.**rs_asin**(*p*, *x*, *prec*)

Arcsine of a series

Return the series expansion of the asin of p, about 0.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_asin
>>> R, x, y = ring('x, y', QQ)
>>> rs_asin(x, x, 8)
5/112*x**7 + 3/40*x**5 + 1/6*x**3 + x
```

See also:

asin (page 395)

sympy.polys.ring_series.**rs_tan**(*p*, *x*, *prec*)

Tangent of a series.

Return the series expansion of the tan of p, about 0.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_tan
>>> R, x, y = ring('x, y', QQ)
>>> rs_tan(x + x*y, x, 4)
1/3*x**3*y**3 + x**3*y**2 + x**3*y + 1/3*x**3 + x*y + x
```

See also:

_tan1 (page 2656), *tan* (page 391)

sympy.polys.ring_series.**_tan1**(*p*, *x*, *prec*)

Helper function of *rs_tan()* (page 2655).

Return the series expansion of tan of a univariate series using Newton's method. It takes advantage of the fact that series expansion of atan is easier than that of tan.

Consider $f(x) = y - \arctan(x)$ Let r be a root of f(x) found using Newton's method. Then $f(r) = 0$ Or $y = \arctan(x)$ where $x = \tan(y)$ as required.

sympy.polys.ring_series.**rs_cot**(*p*, *x*, *prec*)

Cotangent of a series

Return the series expansion of the cot of p, about 0.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_cot
>>> R, x, y = ring('x, y', QQ)
>>> rs_cot(x, x, 6)
-2/945*x**5 - 1/45*x**3 - 1/3*x + x**(-1)
```

See also:

cot (page 392)

sympy.polys.ring_series.**rs_sin**(*p*, *x*, *prec*)

Sine of a series

Return the series expansion of the sin of p, about 0.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_sin
>>> R, x, y = ring('x, y', QQ)
>>> rs_sin(x + x*y, x, 4)
-1/6*x**3*y**3 - 1/2*x**3*y**2 - 1/2*x**3*y - 1/6*x**3 + x*y + x
```

(continues on next page)

(continued from previous page)

```
>>> rs_sin(x**QQ(3, 2) + x*y**QQ(7, 5), x, 4)
-1/2*x**(7/2)*y**(14/5) - 1/6*x**3*y**(21/5) + x**(3/2) + x*y**(7/5)
```

See also:

sin (page 389)

sympy.polys.ring_series.**rs_cos**(*p, x, prec*)

Cosine of a series

Return the series expansion of the cos of p, about 0.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_cos
>>> R, x, y = ring('x, y', QQ)
>>> rs_cos(x + x*y, x, 4)
-1/2*x**2*y**2 - x**2*y - 1/2*x**2 + 1
>>> rs_cos(x + x*y, x, 4)/x**QQ(7, 5)
-1/2*x**(3/5)*y**2 - x**(3/5)*y - 1/2*x**(3/5) + x**(-7/5)
```

See also:

cos (page 390)

sympy.polys.ring_series.**rs_cos_sin**(*p, x, prec*)

Return the tuple (rs_cos(p, x, prec)`, `rs_sin(p, x, prec)).

Is faster than calling rs_cos and rs_sin separately

sympy.polys.ring_series.**rs_atanh**(*p, x, prec*)

Hyperbolic arctangent of a series

Return the series expansion of the atanh of p, about 0.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_atanh
>>> R, x, y = ring('x, y', QQ)
>>> rs_atanh(x + x*y, x, 4)
1/3*x**3*y**3 + x**3*y**2 + x**3*y + 1/3*x**3 + x*y + x
```

See also:

atanh (page 406)

sympy.polys.ring_series.**rs_sinh**(*p, x, prec*)

Hyperbolic sine of a series

Return the series expansion of the sinh of p, about 0.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_sinh
>>> R, x, y = ring('x, y', QQ)
>>> rs_sinh(x + x*y, x, 4)
1/6*x**3*y**3 + 1/2*x**3*y**2 + 1/2*x**3*y + 1/6*x**3 + x*y + x
```

See also:

sinh (page 402)

sympy.polys.ring_series.**rs_cosh**(*p*, *x*, *prec*)

Hyperbolic cosine of a series

Return the series expansion of the cosh of p, about 0.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_cosh
>>> R, x, y = ring('x, y', QQ)
>>> rs_cosh(x + x*y, x, 4)
1/2*x**2*y**2 + x**2*y + 1/2*x**2 + 1
```


See also:

cosh (page 403)

sympy.polys.ring_series.**rs_tanh**(*p*, *x*, *prec*)

Hyperbolic tangent of a series

Return the series expansion of the tanh of p, about 0.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_tanh
>>> R, x, y = ring('x, y', QQ)
>>> rs_tanh(x + x*y, x, 4)
-1/3*x**3*y**3 - x**3*y**2 - x**3*y - 1/3*x**3 + x*y + x
```

See also:

tanh (page 403)

sympy.polys.ring_series.**rs_hadamard_exp**(*p1*, *inverse=False*)

Return `sum f_i/i!*x**i` from `sum f_i*x**i`, where x is the first variable.

If `invers=True` return `sum f_i*i!*x**i`

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_hadamard_exp
>>> R, x = ring('x', QQ)
>>> p = 1 + x + x**2 + x**3
>>> rs_hadamard_exp(p)
1/6*x**3 + 1/2*x**2 + x + 1
```

Operations

sympy.polys.ring_series.**rs_mul**(*p1*, *p2*, *x*, *prec*)

Return the product of the given two series, modulo O(x**prec).

x is the series variable or its position in the generators.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_mul
>>> R, x = ring('x', QQ)
>>> p1 = x**2 + 2*x + 1
>>> p2 = x + 1
>>> rs_mul(p1, p2, x, 3)
3*x**2 + 3*x + 1
```


sympy.polys.ring_series.**rs_square**(*p1*, *x*, *prec*)

Square the series modulo O(x**prec)

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_square
>>> R, x = ring('x', QQ)
>>> p = x**2 + 2*x + 1
>>> rs_square(p, x, 3)
6*x**2 + 4*x + 1
```

sympy.polys.ring_series.**rs_pow**(*p1*, *n*, *x*, *prec*)

Return p1**n modulo O(x**prec)

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_pow
>>> R, x = ring('x', QQ)
>>> p = x + 1
>>> rs_pow(p, 4, x, 3)
6*x**2 + 4*x + 1
```

sympy.polys.ring_series.**rs_series_inversion**(*p*, *x*, *prec*)

Multivariate series inversion 1/p modulo O(x**prec).

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_inversion
>>> R, x, y = ring('x, y', QQ)
>>> rs_series_inversion(1 + x*y**2, x, 4)
-x**3*y**6 + x**2*y**4 - x*y**2 + 1
>>> rs_series_inversion(1 + x*y**2, y, 4)
-x*y**2 + 1
>>> rs_series_inversion(x + x**2, x, 4)
x**3 - x**2 + x - 1 + x**(-1)
```


sympy.polys.ring_series.**rs_series_reversion**(*p*, *x*, *n*, *y*)

Reversion of a series.

p is a series with O(x**n) of the form $p = ax + f(x)$ where a is a number different from 0.

$f(x) = \sum_{k=2}^{n-1} a_k x_k$

Parameters

a_k : Can depend polynomially on other variables, not indicated.

x : Variable with name x. y : Variable with name y.

Returns

Solve $p = y$, that is, given $ax + f(x) - y = 0$,

find the solution $x = r(y)$ up to $O(y^n)$.

Algorithm

If r_i is the solution at order i, then: $ar_i + f(r_i) - y = O\left(y^{i+1}\right)$

and if r_{i+1} is the solution at order $i+1$, then: $ar_{i+1} + f(r_{i+1}) - y = O\left(y^{i+2}\right)$

We have, $r_{i+1} = r_i + e$, such that, $ae + f(r_i) = O\left(y^{i+2}\right)$ or $e = -f(r_i)/a$

So we use the recursion relation: $r_{i+1} = r_i - f(r_i)/a$ with the boundary condition: $r_1 = y$

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_reversion, rs_trunc
>>> R, x, y, a, b = ring('x, y, a, b', QQ)
>>> p = x - x**2 - 2*b*x**2 + 2*a*b*x**2
>>> p1 = rs_series_reversion(p, x, 3, y); p1
-2*y**2*a*b + 2*y**2*b + y**2 + y
>>> rs_trunc(p.compose(x, p1), y, 3)
y
```

sympy.polys.ring_series.**rs_nth_root**(*p*, *n*, *x*, *prec*)

Multivariate series expansion of the nth root of p.

Parameters

p : Expr

The polynomial to computer the root of.

n : integer

The order of the root to be computed.

x : *PolyElement* (page 2554)

prec : integer

Order of the expanded series.

Notes

The result of this function is dependent on the ring over which the polynomial has been defined. If the answer involves a root of a constant, make sure that the polynomial is over a real field. It cannot yet handle roots of symbols.

Examples

```
>>> from sympy.polys.domains import QQ, RR
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_nth_root
>>> R, x, y = ring('x, y', QQ)
>>> rs_nth_root(1 + x + x*y, -3, x, 3)
2/9*x**2*y**2 + 4/9*x**2*y + 2/9*x**2 - 1/3*x*y - 1/3*x + 1
>>> R, x, y = ring('x, y', RR)
>>> rs_nth_root(3 + x + x*y, 3, x, 2)
0.160249952256379*x*y + 0.160249952256379*x + 1.44224957030741
```

sympy.polys.ring_series.**rs_trunc**(*p1*, *x*, *prec*)

Truncate the series in the x variable with precision prec, that is, modulo O(x**prec)

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_trunc
>>> R, x = ring('x', QQ)
>>> p = x**10 + x**5 + x + 1
>>> rs_trunc(p, x, 12)
x**10 + x**5 + x + 1
>>> rs_trunc(p, x, 10)
x**5 + x + 1
```

sympy.polys.ring_series.**rs_subs**(*p*, *rules*, *x*, *prec*)

Substitution with truncation according to the mapping in rules.

Return a series with precision prec in the generator x

Note that substitutions are not done one after the other

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_subs
>>> R, x, y = ring('x, y', QQ)
>>> p = x**2 + y**2
>>> rs_subs(p, {x: x+ y, y: x+ 2*y}, x, 3)
2*x**2 + 6*x*y + 5*y**2
>>> (x + y)**2 + (x + 2*y)**2
2*x**2 + 6*x*y + 5*y**2
```

which differs from

```
>>> rs_subs(rs_subs(p, {x: x+ y}, x, 3), {y: x+ 2*y}, x, 3)
5*x**2 + 12*x*y + 8*y**2
```


Parameters

p : *PolyElement* (page 2554) Input series.

rules : dict with substitution mappings.

x : *PolyElement* (page 2554) in which the series truncation is to be done.

prec : *Integer* (page 987) order of the series after truncation.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_subs
>>> R, x, y = ring('x, y', QQ)
>>> rs_subs(x**2+y**2, {y: (x+y)**2}, x, 3)
 6*x**2*y**2 + x**2 + 4*x*y**3 + y**4
```

sympy.polys.ring_series.**rs_diff**(*p*, *x*)

Return partial derivative of p with respect to x.

Parameters

x : *PolyElement* (page 2554) with respect to which p is differentiated.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_diff
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x**2*y**3
>>> rs_diff(p, x)
2*x*y**3 + 1
```

sympy.polys.ring_series.**rs_integrate**(*p*, *x*)

Integrate p with respect to x.

Parameters

x : *PolyElement* (page 2554) with respect to which p is integrated.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_integrate
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x**2*y**3
>>> rs_integrate(p, x)
1/3*x**3*y**3 + 1/2*x**2
```

sympy.polys.ring_series.**rs_newton**(*p*, *x*, *prec*)

Compute the truncated Newton sum of the polynomial p

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_newton
>>> R, x = ring('x', QQ)
>>> p = x**2 - 2
>>> rs_newton(p, x, 5)
8*x**4 + 4*x**2 + 2
```

sympy.polys.ring_series.**rs_compose_add**(*p1*, *p2*)

compute the composed sum `prod(p2(x - beta) for beta root of p1)`

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_compose_add
>>> R, x = ring('x', QQ)
>>> f = x**2 - 2
>>> g = x**2 - 3
>>> rs_compose_add(f, g)
x**4 - 10*x**2 + 1
```

References

[R730]

Utility functions

sympy.polys.ring_series.**rs_is_puiseux**(*p, x*)

Test if p is Puiseux series in x.

Raise an exception if it has a negative power in x.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_is_puiseux
>>> R, x = ring('x', QQ)
>>> p = x**QQ(2,5) + x**QQ(2,3) + x
>>> rs_is_puiseux(p, x)
True
```


sympy.polys.ring_series.**rs_puiseux**(*f, p, x, prec*)

Return the puiseux series for $f(p, x, prec)$.

To be used when function f is implemented only for regular series.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_puiseux, rs_exp
>>> R, x = ring('x', QQ)
>>> p = x**QQ(2,5) + x**QQ(2,3) + x
>>> rs_puiseux(rs_exp,p, x, 1)
1/2*x**(4/5) + x**(2/3) + x**(2/5) + 1
```

sympy.polys.ring_series.**rs_puiseux2**(*f, p, q, x, prec*)

Return the puiseux series for $f(p, q, x, prec)$.

To be used when function f is implemented only for regular series.

sympy.polys.ring_series.**rs_series_from_list**(*p*, *c*, *x*, *prec*, *concur=1*)

Return a series $sumc[n] * p * *n$ modulo $O(x * *prec)$.

It reduces the number of multiplications by summing concurrently.

$ax = [1, p, p * *2, .., p * *(J - 1)]$ $s = sum(c[i] * ax[i]$ for i in $range(r, (r + 1) * J)) * p * *((K - 1) * J)$ with $K >= (n + 1)/J$

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_from_list, rs_trunc
>>> R, x = ring('x', QQ)
>>> p = x**2 + x + 1
>>> c = [1, 2, 3]
>>> rs_series_from_list(p, c, x, 4)
6*x**3 + 11*x**2 + 8*x + 6
>>> rs_trunc(1 + 2*p + 3*p**2, x, 4)
6*x**3 + 11*x**2 + 8*x + 6
>>> pc = R.from_list(list(reversed(c)))
>>> rs_trunc(pc.compose(x, p), x, 4)
6*x**3 + 11*x**2 + 8*x + 6
```

sympy.polys.ring_series.**rs_fun**(*p*, *f*, **args*)

Function of a multivariate series computed by substitution.

The case with f method name is used to compute rs_tan and rs_nth_root of a multivariate series:

$rs\_fun(p, tan, iv, prec)$

tan series is first computed for a dummy variable _x, i.e, $rs\_tan(\_x, iv, prec)$. Then we substitute _x with p to get the desired series

Parameters

p : *PolyElement* (page 2554) The multivariate series to be expanded.

f : $ring\_series$ function to be applied on p.

args[-2] : *PolyElement* (page 2554) with respect to which, the series is to be expanded.

args[-1] : Required order of the expanded series.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_fun, _tan1
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x*y + x**2*y + x**3*y**2
>>> rs_fun(p, _tan1, x, 4)
1/3*x**3*y**3 + 2*x**3*y**2 + x**3*y + 1/3*x**3 + x**2*y + x*y + x
```

sympy.polys.ring_series.**mul_xin**(*p*, *i*, *n*)

Return $p * x_i * *n$.

x_i is the ith variable in p.

sympy.polys.ring_series.**pow_xin**(*p*, *i*, *n*)

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import pow_xin
>>> R, x, y = ring('x, y', QQ)
>>> p = x**QQ(2,5) + x + x**QQ(2,3)
>>> index = p.ring.gens.index(x)
>>> pow_xin(p, index, 15)
x**15 + x**10 + x**6
```

Literature

The following is a non-comprehensive list of publications that were used as a theoretical foundation for implementing polynomials manipulation module.

Poly solvers

This module provides functions for solving systems of linear equations that are used internally in sympy. Low-level linear systems solver.

solve_lin_sys

sympy.polys.solvers.**solve_lin_sys**(*eqs*, *ring*, *_raw=True*)

Solve a system of linear equations from a PolynomialRing

Parameters

eqs: list[PolyElement]

The linear equations to be solved as elements of a PolynomialRing (assumed equal to zero).

ring: PolynomialRing

The polynomial ring from which eqs are drawn. The generators of this ring are the unkowns to be solved for and the domain of the ring is the domain of the coefficients of the system of equations.

_raw: bool

If *_raw* is False, the keys and values in the returned dictionary will be of type Expr (and the unit of the field will be removed from the keys) otherwise the low-level polys types will be returned, e.g. PolyElement: PythonRational.

Returns

None if the system has no solution.

dict[Symbol, Expr] if _raw=False

dict[Symbol, DomainElement] if _raw=True.

Explanation

Solves a system of linear equations given as PolyElement instances of a PolynomialRing. The basic arithmetic is carried out using instance of DomainElement which is more efficient than *Expr* (page 947) for the most common inputs.

While this is a public function it is intended primarily for internal use so its interface is not necessarily convenient. Users are suggested to use the *sympy.solvers.solveset.linsolve()* (page 872) function (which uses this function internally) instead.

Examples

```
>>> from sympy import symbols
>>> from sympy.polys.solvers import solve_lin_sys, sympy_eqs_to_ring
>>> x, y = symbols('x, y')
>>> eqs = [x - y, x + y - 2]
>>> eqs_ring, ring = sympy_eqs_to_ring(eqs, [x, y])
>>> solve_lin_sys(eqs_ring, ring)
{y: 1, x: 1}
```

Passing `_raw=False` returns the same result except that the keys are `Expr` rather than low-level poly types.

```
>>> solve_lin_sys(eqs_ring, ring, _raw=False)
{x: 1, y: 1}
```

See also:

sympy_eqs_to_ring **(page 2668)**
prepares the inputs to `solve_lin_sys`.

linsolve **(page 872)**
`linsolve` uses `solve_lin_sys` internally.

sympy.solvers.solvers.solve **(page 836)**
`solve` uses `solve_lin_sys` internally.

eqs_to_matrix

`sympy.polys.solvers.`**`eqs_to_matrix`**`(`*eqs_coeffs, eqs_rhs, gens, domain*`)`

Get matrix from linear equations in dict format.

Parameters

eqs_coeffs: list[dict[Symbol, DomainElement]]

The left hand sides of the equations as dicts mapping from symbols to coefficients where the coefficients are instances of DomainElement.

eqs_rhs: list[DomainElements]

The right hand sides of the equations as instances of DomainElement.

gens: list[Symbol]

The unknowns in the system of equations.

domain: Domain

The domain for coefficients of both lhs and rhs.

Returns

The augmented matrix representation of the system as a DomainMatrix.

Explanation

Get the matrix representation of a system of linear equations represented as dicts with low-level DomainElement coefficients. This is an *internal* function that is used by solve_lin_sys.

Examples

```
>>> from sympy import symbols, ZZ
>>> from sympy.polys.solvers import eqs_to_matrix
>>> x, y = symbols('x, y')
>>> eqs_coeff = [{x:ZZ(1), y:ZZ(1)}, {x:ZZ(1), y:ZZ(-1)}]
>>> eqs_rhs = [ZZ(0), ZZ(-1)]
>>> eqs_to_matrix(eqs_coeff, eqs_rhs, [x, y], ZZ)
DomainMatrix([[1, 1, 0], [1, -1, 1]], (2, 3), ZZ)
```

See also:

solve_lin_sys **(page 2666)**
Uses *eqs_to_matrix()* (page 2667) internally

sympy_eqs_to_ring

sympy.polys.solvers.**sympy_eqs_to_ring**(*eqs, symbols*)

Convert a system of equations from Expr to a PolyRing

Parameters

eqs: List of Expr

A list of equations as Expr instances

symbols: List of Symbol

A list of the symbols that are the unknowns in the system of equations.

Returns

Tuple[List[PolyElement], Ring]: The equations as PolyElement instances

and the ring of polynomials within which each equation is represented.

Explanation

High-level functions like solve expect Expr as inputs but can use solve_lin_sys internally. This function converts equations from Expr to the low-level poly types used by the solve_lin_sys function.

Examples

```
>>> from sympy import symbols
>>> from sympy.polys.solvers import sympy_eqs_to_ring
>>> a, x, y = symbols('a, x, y')
>>> eqs = [x-y, x+a*y]
>>> eqs_ring, ring = sympy_eqs_to_ring(eqs, [x, y])
>>> eqs_ring
[x - y, x + a*y]
>>> type(eqs_ring[0])
<class 'sympy.polys.rings.PolyElement'>
>>> ring
ZZ(a)[x,y]
```

With the equations in this form they can be passed to solve_lin_sys:

```
>>> from sympy.polys.solvers import solve_lin_sys
>>> solve_lin_sys(eqs_ring, ring)
{y: 0, x: 0}
```


_solve_lin_sys

sympy.polys.solvers.**_solve_lin_sys**(*eqs_coeffs, eqs_rhs, ring*)

Solve a linear system from dict of PolynomialRing coefficients

Explanation

This is an **internal** function used by *solve_lin_sys()* (page 2666) after the equations have been preprocessed. The role of this function is to split the system into connected components and pass those to *_solve_lin_sys_component()* (page 2670).

Examples

Setup a system for $x - y = 0$ and $x + y = 2$ and solve:

```
>>> from sympy import symbols, sring
>>> from sympy.polys.solvers import _solve_lin_sys
>>> x, y = symbols('x, y')
>>> R, (xr, yr) = sring([x, y], [x, y])
>>> eqs = [{xr:R.one, yr:-R.one}, {xr:R.one, yr:R.one}]
>>> eqs_rhs = [R.zero, -2*R.one]
>>> _solve_lin_sys(eqs, eqs_rhs, R)
{y: 1, x: 1}
```

See also:

solve_lin_sys **(page 2666)**
This function is used internally by *solve_lin_sys()* (page 2666).

_solve_lin_sys_component

sympy.polys.solvers.**_solve_lin_sys_component**(*eqs_coeffs, eqs_rhs, ring*)
Solve a linear system from dict of PolynomialRing coefficients

Explanation

This is an **internal** function used by *solve_lin_sys()* (page 2666) after the equations have been preprocessed. After *_solve_lin_sys()* (page 2669) splits the system into connected components this function is called for each component. The system of equations is solved using Gauss-Jordan elimination with division followed by back-substitution.

Examples

Setup a system for $x - y = 0$ and $x + y = 2$ and solve:

```
>>> from sympy import symbols, sring
>>> from sympy.polys.solvers import _solve_lin_sys_component
>>> x, y = symbols('x, y')
>>> R, (xr, yr) = sring([x, y], [x, y])
>>> eqs = [{xr:R.one, yr:-R.one}, {xr:R.one, yr:R.one}]
>>> eqs_rhs = [R.zero, -2*R.one]
>>> _solve_lin_sys_component(eqs, eqs_rhs, R)
{y: 1, x: 1}
```

See also:

solve_lin_sys **(page 2666)**
This function is used internally by *solve_lin_sys()* (page 2666).

Introducing the domainmatrix of the poly module

This page introduces the idea behind domainmatrix which is used in SymPy's *sympy.polys* (page 2360) module. This is a relatively advanced topic so for a better understanding it is recommended to read about *Domain* (page 2504) and *DDM* (page 2690) along with *sympy.matrices* (page 1217) module.

What is domainmatrix?

It is way of associating Matrix with *Domain* (page 2504).

A domainmatrix represents a matrix with elements that are in a particular Domain. Each domainmatrix internally wraps a DDM which is used for the lower-level operations. The idea is that the domainmatrix class provides the convenience routines for converting between Expr and the poly domains as well as unifying matrices with different domains.

In general, we represent a matrix without concerning about the *Domain* (page 2504) as:

```
>>> from sympy import Matrix
>>> from sympy.polys.matrices import DomainMatrix
>>> A = Matrix([
... [1, 2],
... [3, 4]])
>>> A
Matrix([
[1, 2],
[3, 4]])
```

DomainMatrix Class Reference

class sympy.polys.matrices.domainmatrix.**DomainMatrix**(*rows*, *shape*, *domain*, ***, *fmt=None*)

Associate Matrix with *Domain* (page 2504)

Explanation

DomainMatrix uses *Domain* (page 2504) for its internal representation which makes it more faster for many common operations than current SymPy Matrix class, but this advantage makes it not entirely compatible with Matrix. DomainMatrix could be found analogous to numpy arrays with "dtype". In the DomainMatrix, each matrix has a domain such as *ZZ* (page 2525) or *QQ<a>* (page 2539).

Examples

Creating a DomainMatrix from the existing Matrix class:

```
>>> from sympy import Matrix
>>> from sympy.polys.matrices import DomainMatrix
>>> Matrix1 = Matrix([
...    [1, 2],
...    [3, 4]])
>>> A = DomainMatrix.from_Matrix(Matrix1)
>>> A
DomainMatrix({0: {0: 1, 1: 2}, 1: {0: 3, 1: 4}}, (2, 2), ZZ)
```

Driectly forming a DomainMatrix:

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...    [ZZ(1), ZZ(2)],
...    [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> A
DomainMatrix([[1, 2], [3, 4]], (2, 2), ZZ)
```

See also:

DDM (page 2690), *SDM* (page 2692), *Domain* (page 2504), *Poly* (page 2378)

add(*B*)

Adds two DomainMatrix matrices of the same Domain

Parameters

A, B: DomainMatrix

matrices to add

Returns

DomainMatrix

DomainMatrix after Addition

Raises

DMShapeError

If the dimensions of the two DomainMatrix are not equal

ValueError

If the domain of the two DomainMatrix are not same

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...    [ZZ(1), ZZ(2)],
...    [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix([
...    [ZZ(4), ZZ(3)],
...    [ZZ(2), ZZ(1)]], (2, 2), ZZ)
```

```
>>> A.add(B)
DomainMatrix([[5, 5], [5, 5]], (2, 2), ZZ)
```

See also:

sub (page 2686), *matmul* (page 2680)

charpoly()

Returns the coefficients of the characteristic polynomial of the DomainMatrix. These elements will be domain elements. The domain of the elements will be same as domain of the DomainMatrix.

Returns

list

coefficients of the characteristic polynomial

Raises

DMNonSquareMatrixError

If the DomainMatrix is not a not Square DomainMatrix

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)]], (2, 2), ZZ)
```

```
>>> A.charpoly()
[1, -5, -2]
```

columnspace()

Returns the columnspace for the DomainMatrix

Returns

DomainMatrix

The columns of this matrix form a basis for the columnspace.

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [QQ(1), QQ(-1)],
...     [QQ(2), QQ(-2)]], (2, 2), QQ)
>>> A.columnspace()
DomainMatrix([[1], [2]], (2, 1), QQ)
```

convert_to(K)

Change the domain of DomainMatrix to desired domain or field

Parameters

K : Represents the desired domain or field.

Alternatively, `None` may be passed, in which case this method just returns a copy of this DomainMatrix.

Returns

DomainMatrix

DomainMatrix with the desired domain or field

Examples

```
>>> from sympy import ZZ, ZZ_I
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)]], (2, 2), ZZ)
```

```
>>> A.convert_to(ZZ_I)
DomainMatrix([[1, 2], [3, 4]], (2, 2), ZZ_I)
```

det()

Returns the determinant of a Square DomainMatrix

Returns
S.Complexes

determinant of Square DomainMatrix

Raises
ValueError

If the domain of DomainMatrix not a Field

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)]], (2, 2), ZZ)
```



```
>>> A.det()
-2
```

classmethod diag(*diagonal*, *domain*, *shape=None*)

Return diagonal matrix with entries from `diagonal`.

Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import ZZ
>>> DomainMatrix.diag([ZZ(5), ZZ(6)], ZZ)
DomainMatrix({0: {0: 5}, 1: {1: 6}}, (2, 2), ZZ)
```

classmethod eye(*shape*, *domain*)

Return identity matrix of size n

Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import QQ
>>> DomainMatrix.eye(3, QQ)
DomainMatrix({0: {0: 1}, 1: {1: 1}, 2: {2: 1}}, (3, 3), QQ)
```

classmethod from_Matrix(*M*, *fmt='sparse'*, ***kwargs*)

Convert Matrix to DomainMatrix

Parameters

M: Matrix

Returns

Returns DomainMatrix with identical elements as M

Examples

```
>>> from sympy import Matrix
>>> from sympy.polys.matrices import DomainMatrix
>>> M = Matrix([
...     [1.0, 3.4],
...     [2.4, 1]])
>>> A = DomainMatrix.from_Matrix(M)
>>> A
DomainMatrix({0: {0: 1.0, 1: 3.4}, 1: {0: 2.4, 1: 1.0}}, (2, 2), RR)
```

We can keep internal representation as ddm using fmt='dense' >>> from sympy import Matrix, QQ >>> from sympy.polys.matrices import DomainMatrix >>> A = DomainMatrix.from_Matrix(Matrix([[QQ(1, 2), QQ(3, 4)], [QQ(0, 1), QQ(0, 1)]]), fmt='dense') >>> A.rep [[1/2, 3/4], [0, 0]]

See also:

Matrix (page 1361)

classmethod from_dict_sympy(*nrows*, *ncols*, *elemsdict*, ***kwargs*)

Parameters

nrows: number of rows

ncols: number of cols

elemsdict: dict of dicts containing non-zero elements of the DomainMatrix

Returns

DomainMatrix containing elements of elemsdict

Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.abc import x,y,z
>>> elemsdict = {0: {0:x}, 1:{1: y}, 2: {2: z}}
>>> A = DomainMatrix.from_dict_sympy(3, 3, elemsdict)
>>> A
DomainMatrix({0: {0: x}, 1: {1: y}, 2: {2: z}}, (3, 3), ZZ[x,y,z])
```

See also:

from_list_sympy (page 2676)

classmethod from_list(*rows, domain*)

Convert a list of lists into a DomainMatrix

Parameters

rows: list of lists

Each element of the inner lists should be either the single arg, or tuple of args, that would be passed to the domain constructor in order to form an element of the domain. See examples.

Returns

DomainMatrix containing elements defined in rows

Examples


```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import FF, QQ, ZZ
>>> A = DomainMatrix.from_list([[1, 0, 1], [0, 0, 1]], ZZ)
>>> A
DomainMatrix([[1, 0, 1], [0, 0, 1]], (2, 3), ZZ)
>>> B = DomainMatrix.from_list([[1, 0, 1], [0, 0, 1]], FF(7))
>>> B
DomainMatrix([[1 mod 7, 0 mod 7, 1 mod 7], [0 mod 7, 0 mod 7, 1 mod␣
↪7]], (2, 3), GF(7))
>>> C = DomainMatrix.from_list([[(1, 2), (3, 1)], [(1, 4), (5, 1)]],␣
↪QQ)
>>> C
DomainMatrix([[1/2, 3], [1/4, 5]], (2, 2), QQ)
```

See also:

from_list_sympy (page 2676)

classmethod from_list_sympy(*nrows, ncols, rows, **kwargs*)

Convert a list of lists of Expr into a DomainMatrix using construct_domain

Parameters

nrows: number of rows

ncols: number of columns

rows: list of lists