To get roots from a specific domain set the `filter` flag with one of the following specifiers: Z, Q, R, I, C. By default all roots are returned (this is equivalent to setting `filter='C'`).

By default a dictionary is returned giving a compact result in case of multiple roots. However to get a list containing all those roots set the `multiple` flag to True; the list will have identical roots appearing next to each other in the result. (For a given Poly, the all_roots method will give the roots in sorted numerical order.)

If the `strict` flag is True, `UnsolvableFactorError` will be raised if the roots found are known to be incomplete (because some roots are not expressible in radicals).

Examples

```
>>> from sympy import Poly, roots, degree
>>> from sympy.abc import x, y
```

```
>>> roots(x**2 - 1, x)
{-1: 1, 1: 1}
```

```
>>> p = Poly(x**2-1, x)
>>> roots(p)
{-1: 1, 1: 1}
```

```
>>> p = Poly(x**2-y, x, y)
```

```
>>> roots(Poly(p, x))
{-sqrt(y): 1, sqrt(y): 1}
```

```
>>> roots(x**2 - y, x)
{-sqrt(y): 1, sqrt(y): 1}
```

```
>>> roots([1, 0, -1])
{-1: 1, 1: 1}
```

`roots` will only return roots expressible in radicals. If the given polynomial has some or all of its roots inexpressible in radicals, the result of `roots` will be incomplete or empty respectively.

Example where result is incomplete:

```
>>> roots((x-1)*(x**5-x+1), x)
{1: 1}
```

In this case, the polynomial has an unsolvable quintic factor whose roots cannot be expressed by radicals. The polynomial has a rational root (due to the factor $(x-1)$), which is returned since `roots` always finds all rational roots.

Example where result is empty:

```
>>> roots(x**7-3*x**2+1, x)
{}
```

Here, the polynomial has no roots expressible in radicals, so `roots` returns an empty dictionary.

The result produced by `roots` is complete if and only if the sum of the multiplicity of each root is equal to the degree of the polynomial. If strict=True, UnsolvableFactorError will be raised if the result is incomplete.

The result can be be checked for completeness as follows:

```
>>> f = x**3-2*x**2+1
>>> sum(roots(f, x).values()) == degree(f, x)
True
>>> f = (x-1)*(x**5-x+1)
>>> sum(roots(f, x).values()) == degree(f, x)
False
```

References

[R720]

Special polynomials

sympy.polys.specialpolys.**swinnerton_dyer_poly**(*n*, *x=None*, *polys=False*)

Generates n-th Swinnerton-Dyer polynomial in x.

Parameters

n : int

n decides the order of polynomial

x : optional

polys : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

sympy.polys.specialpolys.**interpolating_poly**(*n*, *x*, *X='x'*, *Y='y'*)

Construct Lagrange interpolating polynomial for `n` data points. If a sequence of values are given for `X` and `Y` then the first `n` values will be used.

sympy.polys.specialpolys.**cyclotomic_poly**(*n*, *x=None*, *polys=False*)

Generates cyclotomic polynomial of order n in x.

Parameters

n : int

n decides the order of polynomial

x : optional

polys : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

sympy.polys.specialpolys.**symmetric_poly**(*n*, **gens*, ***args*)

Generates symmetric polynomial of order n.

Returns a Poly object when `polys=True`, otherwise (default) returns an expression.

sympy.polys.specialpolys.**random_poly**(*x*, *n*, *inf*, *sup*, *domain=ZZ*, *polys=False*)

Generates a polynomial of degree `n` with coefficients in `[inf, sup]`.

Parameters

x

x is the independent term of polynomial

n : int

n decides the order of polynomial

inf

Lower limit of range in which coefficients lie

sup

Upper limit of range in which coefficients lie

domain : optional

Decides what ring the coefficients are supposed to belong. Default is set to Integers.

polys : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

Orthogonal polynomials

sympy.polys.orthopolys.**chebyshevt_poly**(*n*, *x=None*, *polys=False*)

Generates Chebyshev polynomial of the first kind of degree n in x.

Parameters

n : int

n decides the degree of polynomial

x : optional

polys : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

sympy.polys.orthopolys.**chebyshevu_poly**(*n*, *x=None*, *polys=False*)

Generates Chebyshev polynomial of the second kind of degree n in x.

Parameters

n : int

n decides the degree of polynomial

x : optional

polys : bool, optional

polys=True returns an expression, otherwise (default) returns an expression.

sympy.polys.orthopolys.**gegenbauer_poly**(*n*, *a*, *x=None*, *polys=False*)

Generates Gegenbauer polynomial of degree n in x.

Parameters

n : int

n decides the degree of polynomial

x : optional

a

Decides minimal domain for the list of coefficients.

polys : bool, optional

polys=True returns an expression, otherwise (default) returns an expression.

sympy.polys.orthopolys.**hermite_poly**(*n*, *x=None*, *polys=False*)

Generates Hermite polynomial of degree n in x.

Parameters

n : int

n decides the degree of polynomial

x : optional

polys : bool, optional

polys=True returns an expression, otherwise (default) returns an expression.

sympy.polys.orthopolys.**jacobi_poly**(*n*, *a*, *b*, *x=None*, *polys=False*)

Generates Jacobi polynomial of degree n in x.

Parameters

n : int

n decides the degree of polynomial

a

Lower limit of minimal domain for the list of coefficients.

b

Upper limit of minimal domain for the list of coefficients.

x : optional

polys : bool, optional

polys=True returns an expression, otherwise (default) returns an expression.

sympy.polys.orthopolys.**legendre_poly**(*n*, *x=None*, *polys=False*)

Generates Legendre polynomial of degree n in x.

Parameters

n : int

n decides the degree of polynomial

x : optional

polys : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.`**laguerre_poly**(*n*, *x=None*, *alpha=None*, *polys=False*)

Generates Laguerre polynomial of degree n in x.

Parameters

n : int

n decides the degree of polynomial

x : optional

alpha

Decides minimal domain for the list of coefficients.

polys : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.`**spherical_bessel_fn**(*n*, *x=None*, *polys=False*)

Coefficients for the spherical Bessel functions.

Those are only needed in the jn() function.

The coefficients are calculated from:

fn(0, z) = 1/z fn(1, z) = 1/z**2 fn(n-1, z) + fn(n+1, z) == (2*n+1)/z * fn(n, z)

Parameters

n : int

n decides the degree of polynomial

x : optional

polys : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

Examples

```
>>> from sympy.polys.orthopolys import spherical_bessel_fn as fn
>>> from sympy import Symbol
>>> z = Symbol("z")
>>> fn(1, z)
z**(-2)
>>> fn(2, z)
-1/z + 3/z**3
>>> fn(3, z)
-6/z**2 + 15/z**4
>>> fn(4, z)
1/z - 45/z**3 + 105/z**5
```

Manipulation of rational functions

sympy.polys.rationaltools.**together**(*expr*, *deep=False*, *fraction=True*)

Denest and combine rational expressions using symbolic methods.

This function takes an expression or a container of expressions and puts it (them) together by denesting and combining rational subexpressions. No heroic measures are taken to minimize degree of the resulting numerator and denominator. To obtain completely reduced expression use *cancel()* (page 2376). However, *together()* (page 2442) can preserve as much as possible of the structure of the input expression in the output (no expansion is performed).

A wide variety of objects can be put together including lists, tuples, sets, relational objects, integrals and others. It is also possible to transform interior of function applications, by setting deep flag to True.

By definition, *together()* (page 2442) is a complement to *apart()* (page 2443), so apart(together(expr)) should return expr unchanged. Note however, that *together()* (page 2442) uses only symbolic methods, so it might be necessary to use *cancel()* (page 2376) to perform algebraic simplification and minimize degree of the numerator and denominator.

Examples

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

```
>>> together(1/x + 1/y)
(x + y)/(x*y)
>>> together(1/x + 1/y + 1/z)
(x*y + x*z + y*z)/(x*y*z)
```

```
>>> together(1/(x*y) + 1/y**2)
(x + y)/(x*y**2)
```

```
>>> together(1/(1 + 1/x) + 1/(1 + 1/y))
(x*(y + 1) + y*(x + 1))/((x + 1)*(y + 1))
```

```
>>> together(exp(1/x + 1/y))
exp(1/y + 1/x)
>>> together(exp(1/x + 1/y), deep=True)
exp((x + y)/(x*y))
```

```
>>> together(1/exp(x) + 1/(x*exp(x)))
(x + 1)*exp(-x)/x
```

```
>>> together(1/exp(2*x) + 1/(x*exp(3*x)))
(x*exp(x) + 1)*exp(-3*x)/x
```

Partial fraction decomposition

sympy.polys.partfrac.**apart**(*f*, *x=None*, *full=False*, ***options*)

Compute partial fraction decomposition of a rational function.

Given a rational function `f`, computes the partial fraction decomposition of `f`. Two algorithms are available: One is based on the undertermined coefficients method, the other is Bronstein's full partial fraction decomposition algorithm.

The undetermined coefficients method (selected by `full=False`) uses polynomial factorization (and therefore accepts the same options as factor) for the denominator. Per default it works over the rational numbers, therefore decomposition of denominators with non-rational roots (e.g. irrational, complex roots) is not supported by default (see options of factor).

Bronstein's algorithm can be selected by using `full=True` and allows a decomposition of denominators with non-rational roots. A human-readable result can be obtained via `doit()` (see examples below).

Examples

```
>>> from sympy.polys.partfrac import apart
>>> from sympy.abc import x, y
```

By default, using the undetermined coefficients method:

```
>>> apart(y/(x + 2)/(x + 1), x)
-y/(x + 2) + y/(x + 1)
```

The undetermined coefficients method does not provide a result when the denominators roots are not rational:

```
>>> apart(y/(x**2 + x + 1), x)
y/(x**2 + x + 1)
```

You can choose Bronstein's algorithm by setting `full=True`:

```
>>> apart(y/(x**2 + x + 1), x, full=True)
RootSum(_w**2 + _w + 1, Lambda(_a, (-2*_a*y/3 - y/3)/(-_a + x)))
```

Calling `doit()` yields a human-readable result:

```
>>> apart(y/(x**2 + x + 1), x, full=True).doit()
(-y/3 - 2*y*(-1/2 - sqrt(3)*I/2)/3)/(x + 1/2 + sqrt(3)*I/2) + (-y/3 -
    2*y*(-1/2 + sqrt(3)*I/2)/3)/(x + 1/2 - sqrt(3)*I/2)
```

See also:

apart_list (page 2443), *assemble_partfrac_list* (page 2445)

sympy.polys.partfrac.**apart_list**(*f*, *x=None*, *dummies=None*, ***options*)

Compute partial fraction decomposition of a rational function and return the result in structured form.

Given a rational function `f` compute the partial fraction decomposition of `f`. Only Bronstein's full partial fraction decomposition algorithm is supported by this method. The

return value is highly structured and perfectly suited for further algorithmic treatment rather than being human-readable. The function returns a tuple holding three elements:

- The first item is the common coefficient, free of the variable x used for decomposition. (It is an element of the base field K.)
- The second item is the polynomial part of the decomposition. This can be the zero polynomial. (It is an element of $K[x]$.)
- The third part itself is a list of quadruples. Each quadruple has the following elements in this order:
  - The (not necessarily irreducible) polynomial D whose roots w_i appear in the linear denominator of a bunch of related fraction terms. (This item can also be a list of explicit roots. However, at the moment `apart_list` never returns a result this way, but the related `assemble_partfrac_list` function accepts this format as input.)
  - The numerator of the fraction, written as a function of the root w
  - The linear denominator of the fraction *excluding its power exponent*, written as a function of the root w.
  - The power to which the denominator has to be raised.

On can always rebuild a plain expression by using the function `assemble_partfrac_list`.

Examples

A first example:

```
>>> from sympy.polys.partfrac import apart_list, assemble_partfrac_list
>>> from sympy.abc import x, t
```

```
>>> f = (2*x**3 - 2*x) / (x**2 - 2*x + 1)
>>> pfd = apart_list(f)
>>> pfd
(1,
Poly(2*x + 4, x, domain='ZZ'),
[(Poly(_w - 1, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
2*x + 4 + 4/(x - 1)
```

Second example:

```
>>> f = (-2*x - 2*x**2) / (3*x**2 - 6*x)
>>> pfd = apart_list(f)
>>> pfd
(-1,
Poly(2/3, x, domain='QQ'),
[(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 2), Lambda(_a, -_a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-2/3 - 2/(x - 2)
```

Another example, showing symbolic parameters:

```
>>> pfd = apart_list(t/(x**2 + x + t), x)
>>> pfd
(1,
Poly(0, x, domain='ZZ[t]'),
[(Poly(_w**2 + _w + t, _w, domain='ZZ[t]'),
Lambda(_a, -2*_a*t/(4*t - 1) - t/(4*t - 1)),
Lambda(_a, -_a + x),
1)])
```

```
>>> assemble_partfrac_list(pfd)
RootSum(_w**2 + _w + t, Lambda(_a, (-2*_a*t/(4*t - 1) - t/(4*t - 1))/(-_
↪a + x)))
```

This example is taken from Bronstein's original paper:

```
>>> f = 36 / (x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2)
>>> pfd = apart_list(f)
>>> pfd
(1,
Poly(0, x, domain='ZZ'),
[(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1),
(Poly(_w**2 - 1, _w, domain='ZZ'), Lambda(_a, -3*_a - 6), Lambda(_a, -_a␣
↪+ x), 2),
(Poly(_w + 1, _w, domain='ZZ'), Lambda(_a, -4), Lambda(_a, -_a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-4/(x + 1) - 3/(x + 1)**2 - 9/(x - 1)**2 + 4/(x - 2)
```

See also:

apart (page 2443), *assemble_partfrac_list* (page 2445)

References

[R721]

sympy.polys.partfrac.**assemble_partfrac_list**(*partial_list*)

Reassemble a full partial fraction decomposition from a structured result obtained by the function `apart_list`.

Examples

This example is taken from Bronstein's original paper:

```
>>> from sympy.polys.partfrac import apart_list, assemble_partfrac_list
>>> from sympy.abc import x
```

```
>>> f = 36 / (x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2)
>>> pfd = apart_list(f)
```

(continues on next page)

(continued from previous page)

```
>>> pfd
(1,
Poly(0, x, domain='ZZ'),
[(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1),
(Poly(_w**2 - 1, _w, domain='ZZ'), Lambda(_a, -3*_a - 6), Lambda(_a, -_a␣
↪+ x), 2),
(Poly(_w + 1, _w, domain='ZZ'), Lambda(_a, -4), Lambda(_a, -_a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-4/(x + 1) - 3/(x + 1)**2 - 9/(x - 1)**2 + 4/(x - 2)
```

If we happen to know some roots we can provide them easily inside the structure:

```
>>> pfd = apart_list(2/(x**2-2))
>>> pfd
(1,
Poly(0, x, domain='ZZ'),
[(Poly(_w**2 - 2, _w, domain='ZZ'),
Lambda(_a, _a/2),
Lambda(_a, -_a + x),
1)])
```

```
>>> pfda = assemble_partfrac_list(pfd)
>>> pfda
RootSum(_w**2 - 2, Lambda(_a, _a/(-_a + x)))/2
```

```
>>> pfda.doit()
-sqrt(2)/(2*(x + sqrt(2))) + sqrt(2)/(2*(x - sqrt(2)))
```

```
>>> from sympy import Dummy, Poly, Lambda, sqrt
>>> a = Dummy("a")
>>> pfd = (1, Poly(0, x, domain='ZZ'), [([sqrt(2),-sqrt(2)], Lambda(a, a/
↪2), Lambda(a, -a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-sqrt(2)/(2*(x + sqrt(2))) + sqrt(2)/(2*(x - sqrt(2)))
```

See also:

apart (page 2443), *apart_list* (page 2443)

Dispersion of Polynomials

sympy.polys.dispersion.**dispersionset**(*p*, *q=None*, **gens*, ***args*)

Compute the *dispersion set* of two polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion set $\mathrm{J}(f, g)$ is defined as:

$$\begin{aligned}\mathrm{J}(f,g) &:= \{a \in \mathbb{N}_0 | \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 | \deg \gcd(f(x), g(x+a)) \geq 1\}\end{aligned}$$

For a single polynomial one defines $\mathrm{J}(f) := \mathrm{J}(f, f)$.

Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 +␣
↪2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

See also:

dispersion (page 2448)

References

[R722], [R723], [R724], [R725]

sympy.polys.dispersion.**dispersion**(*p*, *q=None*, **gens*, ***args*)

Compute the *dispersion* of polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion $\operatorname{dis}(f, g)$ is defined as:

$$\begin{aligned}\operatorname{dis}(f, g) &:= \max\{J(f, g) \cup \{0\}\} \\ &= \max\{\{a \in \mathbb{N} | \gcd(f(x), g(x + a)) \neq 1\} \cup \{0\}\}\end{aligned}$$

and for a single polynomial $\operatorname{dis}(f) := \operatorname{dis}(f, f)$. Note that we make the definition $\max\{\} := -\infty$.

Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

The maximum of an empty set is defined to be $-\infty$ as seen in this example.

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 +␣
↪2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

See also:

dispersionset (page 2446)

References

[R726], [R727], [R728], [R729]

AGCA - Algebraic Geometry and Commutative Algebra Module

Introduction

> Algebraic geometry is a mixture of the ideas of two Mediterranean cultures. It is the superposition of the Arab science of the lightening calculation of the solutions of equations over the Greek art of position and shape. This tapestry was originally woven on European soil and is still being refined under the influence of international fashion. Algebraic geometry studies the delicate balance between the geometrically plausible and the algebraically possible. Whenever one side of this mathematical teeter-totter outweighs the other, one immediately loses interest and runs off in search of a more exciting amusement.
>
> —George R. Kempf (1944 – 2002)

Algebraic Geometry refers to the study of geometric problems via algebraic methods (and sometimes vice versa). While this is a rather old topic, algebraic geometry as understood today is very much a 20th century development. Building on ideas of e.g. Riemann and Dedekind, it was realized that there is an intimate connection between properties of the set of solutions of a system of polynomial equations (called an algebraic variety) and the behavior of the set of polynomial functions on that variety (called the coordinate ring).

As in many geometric disciplines, we can distinguish between local and global questions (and methods). Local investigations in algebraic geometry are essentially equivalent to the study of certain rings, their ideals and modules. This latter topic is also called commutative algebra. It is the basic local toolset of algebraic geometers, in much the same way that differential analysis is the local toolset of differential geometers.

A good conceptual introduction to commutative algebra is [Atiyah69]. An introduction more geared towards computations, and the work most of the algorithms in this module are based on, is [Greuel2008].

This module aims to eventually allow expression and solution of both local and global geometric problems, both in the classical case over a field and in the more modern arithmetic cases. So far, however, there is no geometric functionality at all. Currently the module only provides tools for computational commutative algebra over fields.

All code examples assume:

```
>>> from sympy import *
>>> x, y, z = symbols('x,y,z')
>>> init_printing(use_unicode=True, wrap_line=False)
```

Reference

In this section we document the usage of the AGCA module. For convenience of the reader, some definitions and examples/explanations are interspersed.

Base Rings

Almost all computations in commutative algebra are relative to a "base ring". (For example, when asking questions about an ideal, the base ring is the ring the ideal is a subset of.) In principle all polys "domains" can be used as base rings. However, useful functionality is only implemented for polynomial rings over fields, and various localizations and quotients thereof.

As demonstrated in the examples below, the most convenient method to create objects you are interested in is to build them up from the ground field, and then use the various methods to create new objects from old. For example, in order to create the local ring of the nodal cubic $y^2 = x^3$ at the origin, over $\mathbb{Q}$, you do:

```
>>> lr = QQ.old_poly_ring(x, y, order="ilex") / [y**2 - x**3]
>>> lr
ℚ[x, y, order=ilex]
──────────────────
    ⎛   3    2⎞
    ⎝- x  + y ⎠
```

Note how the python list notation can be used as a short cut to express ideals. You can use the `convert` method to return ordinary sympy objects into objects understood by the AGCA module (although in many cases this will be done automatically – for example the list was automatically turned into an ideal, and in the process the symbols x and y were automatically converted into other representations). For example:

```
>>> X, Y = lr.convert(x), lr.convert(y) ; X
    ⎛   3    2⎞
x + ⎝- x  + y ⎠

>>> x**3 == y**2
False

>>> X**3 == Y**2
True
```

When no localisation is needed, a more mathematical notation can be used. For example, let us create the coordinate ring of three-dimensional affine space $\mathbb{A}^3$:

```
>>> ar = QQ.old_poly_ring(x, y, z); ar
ℚ[x, y, z]
```

For more details, refer to the following class documentation. Note that the base rings, being domains, are the main point of overlap between the AGCA module and the rest of the polys module. All domains are documented in detail in the polys reference, so we show here only an abridged version, with the methods most pertinent to the AGCA module.

class sympy.polys.domains.ring.**Ring**

Represents a ring domain.

free_module(*rank*)

Generate a free module of rank `rank` over self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

ideal(**gens*)

Generate an ideal of `self`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x**2)
<x**2>
```

quotient_ring(*e*)

Form a quotient ring of `self`.

Here `e` can be an ideal or an iterable.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).quotient_ring(QQ.old_poly_ring(x).ideal(x**2))
QQ[x]/<x**2>
>>> QQ.old_poly_ring(x).quotient_ring([x**2])
QQ[x]/<x**2>
```

The division operator has been overloaded for this:

```
>>> QQ.old_poly_ring(x)/[x**2]
QQ[x]/<x**2>
```

sympy.polys.domains.polynomialring.**PolynomialRing**(*domain_or_ring*, *symbols=None*, *order=None*)

A class for representing multivariate polynomial rings.

class sympy.polys.domains.quotientring.**QuotientRing**(*ring*, *ideal*)

Class representing (commutative) quotient rings.

You should not usually instantiate this by hand, instead use the constructor from the base ring in the construction.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**3 + 1)
>>> QQ.old_poly_ring(x).quotient_ring(I)
QQ[x]/<x**3 + 1>
```

Shorter versions are possible:

```
>>> QQ.old_poly_ring(x)/I
QQ[x]/<x**3 + 1>
```

```
>>> QQ.old_poly_ring(x)/[x**3 + 1]
QQ[x]/<x**3 + 1>
```

Attributes:

- ring - the base ring
- base_ideal - the ideal used to form the quotient

Modules, Ideals and their Elementary Properties

Let A be a ring. An A-module is a set M, together with two binary operations $+ : M \times M \to M$ and $\times : R \times M \to M$ called addition and scalar multiplication. These are required to satisfy certain axioms, which can be found in e.g. [Atiyah69]. In this way modules are a direct generalisation of both vector spaces (A being a field) and abelian groups ($A = \mathbb{Z}$). A *submodule* of the A-module M is a subset $N \subset M$, such that the binary operations restrict to N, and N becomes an A-module with these operations.

The ring A itself has a natural A-module structure where addition and multiplication in the module coincide with addition and multiplication in the ring. This A-module is also written as A. An A-submodule of A is called an *ideal* of A. Ideals come up very naturally in algebraic geometry. More general modules can be seen as a technically convenient "elbow room" beyond talking only about ideals.

If M, N are A-modules, then there is a natural (componentwise) A-module structure on $M \times N$. Similarly there are A-module structures on cartesian products of more components. (For the categorically inclined: the cartesian product of finitely many A-modules, with this A-module structure, is the finite biproduct in the category of all A-modules. With infinitely many components, it is the direct product (but the infinite direct sum has to be constructed differently).) As usual, repeated product of the A-module M is denoted $M, M^2, M^3 \ldots$, or M^I for arbitrary index sets I.

An A-module M is called *free* if it is isomorphic to the A-module A^I for some (not necessarily finite) index set I (refer to the next section for a definition of isomorphism). The cardinality of I is called the *rank* of M; one may prove this is well-defined. In general, the AGCA module only works with free modules of finite rank, and other closely related modules. The easiest way to create modules is to use member methods of the objects they are made up from. For example, let us create a free module of rank 4 over the coordinate ring of $\mathbb{A}^2$ we created above, together with a submodule:

```
>>> F = ar.free_module(4) ; F
         4
ℚ[x, y, z]

>>> S = F.submodule([1, x, x**2, x**3], [0, 1, 0, y]) ; S
╱[       2   3]              ╲
╲[1, x, x , x ], [0, 1, 0, y]╱
```

Note how python lists can be used as a short-cut notation for module elements (vectors). As usual, the `convert` method can be used to convert sympy/python objects into the internal

AGCA representation (see detailed reference below).

Here is the detailed documentation of the classes for modules, free modules, and submodules:

class sympy.polys.agca.modules.**Module**(*ring*)

Abstract base class for modules.

Do not instantiate - use ring explicit constructors instead:

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

Attributes:

- dtype - type of elements
- ring - containing ring

Non-implemented methods:

- submodule
- quotient_module
- is_zero
- is_submodule
- multiply_ideal

The method convert likely needs to be changed in subclasses.

contains(*elem*)

Return True if `elem` is an element of this module.

convert(*elem*, *M=None*)

Convert `elem` into internal representation of this module.

If `M` is not None, it should be a module containing it.

identity_hom()

Return the identity homomorphism on `self`.

is_submodule(*other*)

Returns True if `other` is a submodule of `self`.

is_zero()

Returns True if `self` is a zero module.

multiply_ideal(*other*)

Multiply `self` by the ideal `other`.

quotient_module(*other*)

Generate a quotient module.

submodule(**gens*)

Generate a submodule.

subset(*other*)

Returns True if `other` is is a subset of `self`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.subset([(1, x), (x, 2)])
True
>>> F.subset([(1/x, x), (x, 2)])
False
```

class sympy.polys.agca.modules.**FreeModule**(*ring*, *rank*)

Abstract base class for free modules.

Additional attributes:

- rank - rank of the free module

Non-implemented methods:

- submodule

basis()

Return a set of basis elements.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(3).basis()
([1, 0, 0], [0, 1, 0], [0, 0, 1])
```

convert(*elem*, *M=None*)

Convert `elem` into the internal representation.

This method is called implicitly whenever computations involve elements not in the internal representation.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.convert([1, 0])
[1, 0]
```

dtype

alias of *FreeModuleElement* (page 2456)

identity_hom()

Return the identity homomorphism on `self`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).identity_hom()
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2
[0, 1]])
```

is_submodule(*other*)

Returns True if `other` is a submodule of `self`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([2, x])
>>> F.is_submodule(F)
True
>>> F.is_submodule(M)
True
>>> M.is_submodule(F)
False
```

is_zero()

Returns True if `self` is a zero module.

(If, as this implementation assumes, the coefficient ring is not the zero ring, then this is equivalent to the rank being zero.)

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(0).is_zero()
True
>>> QQ.old_poly_ring(x).free_module(1).is_zero()
False
```

multiply_ideal(*other*)

Multiply `self` by the ideal `other`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x)
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.multiply_ideal(I)
<[x, 0], [0, x]>
```

quotient_module(*submodule*)

Return a quotient module.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2)
>>> M.quotient_module(M.submodule([1, x], [x, 2]))
QQ[x]**2/<[1, x], [x, 2]>
```

Or more conicisely, using the overloaded division operator:

```
>>> QQ.old_poly_ring(x).free_module(2) / [[1, x], [x, 2]]
QQ[x]**2/<[1, x], [x, 2]>
```

class sympy.polys.agca.modules.**FreeModuleElement**(*module*, *data*)

Element of a free module. Data stored as a tuple.

class sympy.polys.agca.modules.**SubModule**(*gens*, *container*)

Base class for submodules.

Attributes:

- container - containing module
- gens - generators (subset of containing module)
- rank - rank of containing module

Non-implemented methods:

- _contains
- _syzygies
- _in_terms_of_generators
- _intersect
- _module_quotient

Methods that likely need change in subclasses:

- reduce_element

convert(*elem*, *M=None*)

Convert `elem` into the internal represantition.

Mostly called implicitly.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([1, x])
>>> M.convert([2, 2*x])
[2, 2*x]
```

identity_hom()

Return the identity homomorphism on `self`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([x, x]).identity_
↪hom()
Matrix([
[1, 0], : <[x, x]> -> <[x, x]>
[0, 1]])
```

in_terms_of_generators(*e*)

Express element `e` of `self` in terms of the generators.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([1, 0], [1, 1])
>>> M.in_terms_of_generators([x, x**2])
[-x**2 + x, x**2]
```

inclusion_hom()

Return a homomorphism representing the inclusion map of `self`.

That is, the natural map from `self` to `self.container`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([x, x]).inclusion_
↪hom()
Matrix([
[1, 0], : <[x, x]> -> QQ[x]**2
[0, 1]])
```

intersect(*other*, ***options*)

Returns the intersection of `self` with submodule `other`.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x, y).free_module(2)
>>> F.submodule([x, x]).intersect(F.submodule([y, y]))
<[x*y, x*y]>
```

Some implementation allow further options to be passed. Currently, to only one implemented is `relations=True`, in which case the function will return a triple `(res, rela, relb)`, where `res` is the intersection module, and `rela` and `relb` are lists of coefficient vectors, expressing the generators of `res` in terms of the generators of `self` (`rela`) and `other` (`relb`).

```
>>> F.submodule([x, x]).intersect(F.submodule([y, y]), relations=True)
(<[x*y, x*y]>, [(y,)], [(x,)])
```

The above result says: the intersection module is generated by the single element $(-xy, -xy) = -y(x, x) = -x(y, y)$, where (x, x) and (y, y) respectively are the unique generators of the two modules being intersected.

is_full_module()

Return True if `self` is the entire free module.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_full_module()
False
>>> F.submodule([1, 1], [1, 2]).is_full_module()
True
```

is_submodule(*other*)

Returns True if `other` is a submodule of `self`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([2, x])
>>> N = M.submodule([2*x, x**2])
>>> M.is_submodule(M)
True
>>> M.is_submodule(N)
True
>>> N.is_submodule(M)
False
```

is_zero()

Return True if `self` is a zero module.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_zero()
False
>>> F.submodule([0, 0]).is_zero()
True
```

module_quotient(*other*, ***options*)

Returns the module quotient of `self` by submodule `other`.

That is, if `self` is the module M and `other` is N, then return the ideal $\{f \in R | fN \subset M\}$.

Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x, y
>>> F = QQ.old_poly_ring(x, y).free_module(2)
>>> S = F.submodule([x*y, x*y])
>>> T = F.submodule([x, x])
>>> S.module_quotient(T)
<y>
```

Some implementations allow further options to be passed. Currently, the only one implemented is `relations=True`, which may only be passed if `other` is principal. In this case the function will return a pair `(res, rel)` where `res` is the ideal, and `rel` is a list of coefficient vectors, expressing the generators of the ideal, multiplied by the generator of `other` in terms of generators of `self`.

```
>>> S.module_quotient(T, relations=True)
(<y>, [[1]])
```

This means that the quotient ideal is generated by the single element y, and that $y(x, x) = 1(xy, xy)$, (x, x) and (xy, xy) being the generators of T and S, respectively.

multiply_ideal(I)

Multiply `self` by the ideal `I`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**2)
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([1, 1])
>>> I*M
<[x**2, x**2]>
```

quotient_module(*other*, ***opts*)

Return a quotient module.

This is the same as taking a submodule of a quotient of the containing module.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> S1 = F.submodule([x, 1])
>>> S2 = F.submodule([x**2, x])
>>> S1.quotient_module(S2)
<[x, 1] + <[x**2, x]>>
```

Or more coincisely, using the overloaded division operator:

```
>>> F.submodule([x, 1]) / [(x**2, x)]
<[x, 1] + <[x**2, x]>>
```

reduce_element(*x*)

Reduce the element x of our ring modulo the ideal `self`.

Here "reduce" has no specific meaning, it could return a unique normal form, simplify the expression a bit, or just do nothing.

submodule(**gens*)

Generate a submodule.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([x, 1])
>>> M.submodule([x**2, x])
<[x**2, x]>
```

syzygy_module(***opts*)

Compute the syzygy module of the generators of `self`.

Suppose M is generated by $f_1, \ldots, f_n$ over the ring R. Consider the homomorphism $\phi : R^n \to M$, given by sending $(r_1, \ldots, r_n) \to r_1 f_1 + \cdots + r_n f_n$. The syzygy module is defined to be the kernel of ϕ.

Examples

The syzygy module is zero iff the generators generate freely a free submodule:

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([1, 0], [1, 1]).
↪syzygy_module().is_zero()
True
```

A slightly more interesting example:

```
>>> M = QQ.old_poly_ring(x, y).free_module(2).submodule([x, 2*x], [y,␣
↪2*y])
>>> S = QQ.old_poly_ring(x, y).free_module(2).submodule([y, -x])
>>> M.syzygy_module() == S
True
```

union(*other*)

Returns the module generated by the union of `self` and `other`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(1)
>>> M = F.submodule([x**2 + x]) # <x(x+1)>
>>> N = F.submodule([x**2 - 1]) # <(x-1)(x+1)>
>>> M.union(N) == F.submodule([x+1])
True
```

Ideals are created very similarly to modules. For example, let's verify that the nodal cubic is indeed singular at the origin:

```
>>> I = lr.ideal(x, y)
>>> I == lr.ideal(x)
False

>>> I == lr.ideal(y)
False
```

We are using here the fact that a curve is non-singular at a point if and only if the maximal ideal of the local ring is principal, and that in this case at least one of x and y must be generators.

This is the detailed documentation of the class ideal. Please note that most of the methods regarding properties of ideals (primality etc.) are not yet implemented.

class sympy.polys.agca.ideals.**Ideal**(*ring*)

Abstract base class for ideals.

Do not instantiate - use explicit constructors in the ring class instead:

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> QQ.old_poly_ring(x).ideal(x+1)
<x + 1>
```

Attributes

- ring - the ring this ideal belongs to

Non-implemented methods:

- _contains_elem
- _contains_ideal
- _quotient
- _intersect
- _union
- _product
- is_whole_ring
- is_zero
- is_prime, is_maximal, is_primary, is_radical
- is_principal
- height, depth
- radical

Methods that likely should be overridden in subclasses:

- reduce_element

contains(*elem*)

Return True if `elem` is an element of this ideal.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x+1, x-1).contains(3)
True
>>> QQ.old_poly_ring(x).ideal(x**2, x**3).contains(x)
False
```

depth()

Compute the depth of `self`.

height()

Compute the height of `self`.

intersect(*J*)

Compute the intersection of self with ideal J.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> R = QQ.old_poly_ring(x, y)
>>> R.ideal(x).intersect(R.ideal(y))
<x*y>
```

is_maximal()

Return True if `self` is a maximal ideal.

is_primary()

Return True if `self` is a primary ideal.

is_prime()

Return True if `self` is a prime ideal.

is_principal()

Return True if `self` is a principal ideal.

is_radical()

Return True if `self` is a radical ideal.

is_whole_ring()

Return True if `self` is the whole ring.

is_zero()

Return True if `self` is the zero ideal.

product(J)

Compute the ideal product of `self` and `J`.

That is, compute the ideal generated by products xy, for x an element of `self` and $y \in J$.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> QQ.old_poly_ring(x, y).ideal(x).product(QQ.old_poly_ring(x, y).
↪ideal(y))
<x*y>
```

quotient(J, ***opts*)

Compute the ideal quotient of `self` by `J`.

That is, if `self` is the ideal I, compute the set $I : J = \{x \in R | xJ \subset I\}$.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> R = QQ.old_poly_ring(x, y)
>>> R.ideal(x*y).quotient(R.ideal(x))
<y>
```

radical()

Compute the radical of `self`.

reduce_element(x)

Reduce the element `x` of our ring modulo the ideal `self`.

Here "reduce" has no specific meaning: it could return a unique normal form, simplify the expression a bit, or just do nothing.

saturate(J)

Compute the ideal saturation of `self` by `J`.

That is, if `self` is the ideal I, compute the set $I : J^\infty = \{x \in R | xJ^n \subset I \text{ for some } n\}$.

subset(*other*)

Returns True if `other` is is a subset of `self`.

Here `other` may be an ideal.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x+1)
>>> I.subset([x**2 - 1, x**2 + 2*x + 1])
True
>>> I.subset([x**2 + 1, x + 1])
False
>>> I.subset(QQ.old_poly_ring(x).ideal(x**2 - 1))
True
```

union(J)

Compute the ideal generated by the union of `self` and `J`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x**2 - 1).union(QQ.old_poly_ring(x).
↪ideal((x+1)**2)) == QQ.old_poly_ring(x).ideal(x+1)
True
```

If M is an A-module and N is an A-submodule, we can define two elements x and y of M to be equivalent if $x - y \in N$. The set of equivalence classes is written M/N, and has a natural A-module structure. This is called the quotient module of M by N. If K is a submodule of M

containing N, then K/N is in a natural way a submodule of M/N. Such a module is called a subquotient. Here is the documentation of quotient and subquotient modules:

class sympy.polys.agca.modules.**QuotientModule**(*ring*, *base*, *submodule*)

Class for quotient modules.

Do not instantiate this directly. For subquotients, see the SubQuotientModule class.

Attributes:

- base - the base module we are a quotient of
- killed_module - the submodule used to form the quotient
- rank of the base

convert(*elem*, *M=None*)

Convert `elem` into the internal representation.

This method is called implicitly whenever computations involve elements not in the internal representation.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> F.convert([1, 0])
[1, 0] + <[1, 2], [1, x]>
```

dtype

alias of *QuotientModuleElement* (page 2467)

identity_hom()

Return the identity homomorphism on `self`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> M.identity_hom()
Matrix([
[1, 0], : QQ[x]**2/<[1, 2], [1, x]> -> QQ[x]**2/<[1, 2], [1, x]>
[0, 1]])
```

is_submodule(*other*)

Return True if `other` is a submodule of `self`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> Q = QQ.old_poly_ring(x).free_module(2) / [(x, x)]
>>> S = Q.submodule([1, 0])
>>> Q.is_submodule(S)
True
>>> S.is_submodule(Q)
False
```

is_zero()

Return True if `self` is a zero module.

This happens if and only if the base module is the same as the submodule being killed.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> (F/[(1, 0)]).is_zero()
False
>>> (F/[(1, 0), (0, 1)]).is_zero()
True
```

quotient_hom()

Return the quotient homomorphism to `self`.

That is, return a homomorphism representing the natural map from `self.base` to `self`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> M.quotient_hom()
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2/<[1, 2], [1, x]>
[0, 1]])
```

submodule(**gens*, ***opts*)

Generate a submodule.

This is the same as taking a quotient of a submodule of the base module.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> Q = QQ.old_poly_ring(x).free_module(2) / [(x, x)]
>>> Q.submodule([x, 0])
<[x, 0] + <[x, x]>>
```

class sympy.polys.agca.modules.**QuotientModuleElement**(*module*, *data*)

Element of a quotient module.

eq(*d1*, *d2*)

Equality comparison.

class sympy.polys.agca.modules.**SubQuotientModule**(*gens*, *container*, ***opts*)

Submodule of a quotient module.

Equivalently, quotient module of a submodule.

Do not instantiate this, instead use the submodule or quotient_module constructing methods:

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> S = F.submodule([1, 0], [1, x])
>>> Q = F/[(1, 0)]
>>> S/[(1, 0)] == Q.submodule([5, x])
True
```

Attributes:

- base - base module we are quotient of
- killed_module - submodule used to form the quotient

is_full_module()

Return True if `self` is the entire free module.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_full_module()
False
>>> F.submodule([1, 1], [1, 2]).is_full_module()
True
```

quotient_hom()

Return the quotient homomorphism to self.

That is, return the natural map from `self.base` to `self`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = (QQ.old_poly_ring(x).free_module(2) / [(1, x)]).submodule([1,␣
↪0])
>>> M.quotient_hom()
Matrix([
[1, 0], : <[1, 0], [1, x]> -> <[1, 0] + <[1, x]>, [1, x] + <[1, x]>>
[0, 1]])
```

Module Homomorphisms and Syzygies

Let M and N be A-modules. A mapping $f : M \to N$ satisfying various obvious properties (see [Atiyah69]) is called an A-module homomorphism. In this case M is called the *domain* and N the *codomain*. The set $\{x \in M | f(x) = 0\}$ is called the *kernel* $ker(f)$, whereas the set $\{f(x) | x \in M\}$ is called the *image* $im(f)$. The kernel is a submodule of M, the image is a submodule of N. The homomorphism f is injective if and only if $ker(f) = 0$ and surjective if and only if $im(f) = N$. A bijective homomorphism is called an *isomorphism*. Equivalently, $ker(f) = 0$ and $im(f) = N$. (A related notion, which currently has no special name in the AGCA module, is that of the *cokernel*, $coker(f) = N/im(f)$.)

Suppose now M is an A-module. M is called *finitely generated* if there exists a surjective homomorphism $A^n \to M$ for some n. If such a morphism f is chosen, the images of the standard basis of A^n are called the *generators* of M. The module $ker(f)$ is called *syzygy module* with respect to the generators. A module is called *finitely presented* if it is finitely generated with a finitely generated syzygy module. The class of finitely presented modules is essentially the largest class we can hope to be able to meaningfully compute in.

It is an important theorem that, for all the rings we are considering, all submodules of finitely generated modules are finitely generated, and hence finitely generated and finitely presented modules are the same.

The notion of syzygies, while it may first seem rather abstract, is actually very computational. This is because there exist (fairly easy) algorithms for computing them, and more general questions (kernels, intersections, ...) are often reduced to syzygy computation.

Let us say a few words about the definition of homomorphisms in the AGCA module. Suppose first that $f : M \to N$ is an arbitrary morphism of A-modules. Then if K is a submodule of M, f naturally defines a new homomorphism $g : K \to N$ (via $g(x) = f(x)$), called the *restriction* of f to K. If now K contained in the kernel of f, then moreover f defines in a natural homomorphism $g : M/K \to N$ (same formula as above!), and we say that f *descends* to M/K. Similarly, if L is a submodule of N, there is a natural homomorphism $g : M \to N/L$, we say that g *factors* through f. Finally, if now L contains the image of f, then there is a natural homomorphism $g : M \to L$ (defined, again, by the same formula), and we say g is obtained from f by restriction of codomain. Observe also that each of these four operations is reversible, in the sense that given g, one can always (non-uniquely) find f such that g is obtained from f in the above way.

Note that all modules implemented in AGCA are obtained from free modules by taking a succession of submodules and quotients. Hence, in order to explain how to define a homomorphism between arbitrary modules, in light of the above, we need only explain how to define homomorphisms of free modules. But, essentially by the definition of free module, a homomorphism from a free module A^n to any module M is precisely the same as giving n elements of M (the images of the standard basis), and giving an element of a free module

A^m is precisely the same as giving m elements of A. Hence a homomorphism of free modules $A^n \to A^m$ can be specified via a matrix, entirely analogously to the case of vector spaces.

The functions `restrict_domain` etc. of the class `Homomorphism` can be used to carry out the operations described above, and homomorphisms of free modules can in principle be instantiated by hand. Since these operations are so common, there is a convenience function `homomorphism` to define a homomorphism between arbitrary modules via the method outlined above. It is essentially the only way homomorphisms need ever be created by the user.

`sympy.polys.agca.homomorphisms.`**`homomorphism`**(*domain*, *codomain*, *matrix*)

Create a homomorphism object.

This function tries to build a homomorphism from `domain` to `codomain` via the matrix `matrix`.

Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> R = QQ.old_poly_ring(x)
>>> T = R.free_module(2)
```

If `domain` is a free module generated by $e_1, \ldots, e_n$, then `matrix` should be an n-element iterable $(b_1, \ldots, b_n)$ where the b_i are elements of `codomain`. The constructed homomorphism is the unique homomorphism sending e_i to b_i.

```
>>> F = R.free_module(2)
>>> h = homomorphism(F, T, [[1, x], [x**2, 0]])
>>> h
Matrix([
[1, x**2], : QQ[x]**2 -> QQ[x]**2
[x,    0]])
>>> h([1, 0])
[1, x]
>>> h([0, 1])
[x**2, 0]
>>> h([1, 1])
[x**2 + 1, x]
```

If `domain` is a submodule of a free module, them `matrix` determines a homomoprhism from the containing free module to `codomain`, and the homomorphism returned is obtained by restriction to `domain`.

```
>>> S = F.submodule([1, 0], [0, x])
>>> homomorphism(S, T, [[1, x], [x**2, 0]])
Matrix([
[1, x**2], : <[1, 0], [0, x]> -> QQ[x]**2
[x,    0]])
```

If `domain` is a (sub)quotient N/K, then `matrix` determines a homomorphism from N to `codomain`. If the kernel contains K, this homomorphism descends to `domain` and is returned; otherwise an exception is raised.

```
>>> homomorphism(S/[(1, 0)], T, [0, [x**2, 0]])
Matrix([
[0, x**2], : <[1, 0] + <[1, 0]>, [0, x] + <[1, 0]>, [1, 0] + <[1, 0]>> ->
↪ QQ[x]**2
[0,    0]])
>>> homomorphism(S/[(0, x)], T, [0, [x**2, 0]])
Traceback (most recent call last):
...
ValueError: kernel <[1, 0], [0, 0]> must contain sm, got <[0,x]>
```

Finally, here is the detailed reference of the actual homomorphism class:

class sympy.polys.agca.homomorphisms.**ModuleHomomorphism**(*domain*, *codomain*)

Abstract base class for module homomoprhisms. Do not instantiate.

Instead, use the `homomorphism` function:

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [0, 1]])
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2
[0, 1]])
```

Attributes:

- ring - the ring over which we are considering modules
- domain - the domain module
- codomain - the codomain module
- _ker - cached kernel
- _img - cached image

Non-implemented methods:

- _kernel
- _image
- _restrict_domain
- _restrict_codomain
- _quotient_domain
- _quotient_codomain
- _apply
- _mul_scalar
- _compose
- _add

image()

Compute the image of `self`.

That is, if `self` is the homomorphism $\phi : M \to N$, then compute $im(\phi) = \{\phi(x) | x \in M\}$. This is a submodule of N.

Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [x, 0]]).image() == F.submodule([1,␣
↪0])
True
```

is_injective()

Return True if `self` is injective.

That is, check if the elements of the domain are mapped to the same codomain element.

Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_injective()
False
>>> h.quotient_domain(h.kernel()).is_injective()
True
```

is_isomorphism()

Return True if `self` is an isomorphism.

That is, check if every element of the codomain has precisely one preimage. Equivalently, `self` is both injective and surjective.

Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h = h.restrict_codomain(h.image())
>>> h.is_isomorphism()
False
>>> h.quotient_domain(h.kernel()).is_isomorphism()
True
```

is_surjective()

Return True if `self` is surjective.

That is, check if every element of the codomain has at least one preimage.

Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_surjective()
False
>>> h.restrict_codomain(h.image()).is_surjective()
True
```

is_zero()

Return True if `self` is a zero morphism.

That is, check if every element of the domain is mapped to zero under self.

Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_zero()
False
>>> h.restrict_domain(F.submodule()).is_zero()
True
```

(continues on next page)

(continued from previous page)

```
>>> h.quotient_codomain(h.image()).is_zero()
True
```

kernel()

Compute the kernel of `self`.

That is, if `self` is the homomorphism $\phi : M \to N$, then compute $ker(\phi) = \{x \in M | \phi(x) = 0\}$. This is a submodule of M.

Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [x, 0]]).kernel()
<[x, -1]>
```

quotient_codomain(*sm*)

Return `self` with codomain replaced by `codomain/sm`.

Here `sm` must be a submodule of `self.codomain`.

Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.quotient_codomain(F.submodule([1, 1]))
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2/<[1, 1]>
[0, 0]])
```

This is the same as composing with the quotient map on the left:

```
>>> (F/[(1, 1)]).quotient_hom() * h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2/<[1, 1]>
[0, 0]])
```

quotient_domain(*sm*)

Return `self` with domain replaced by `domain/sm`.

Here `sm` must be a submodule of `self.kernel()`.

Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.quotient_domain(F.submodule([-x, 1]))
Matrix([
[1, x], : QQ[x]**2/<[-x, 1]> -> QQ[x]**2
[0, 0]])
```

restrict_codomain(*sm*)

Return `self`, with codomain restricted to to `sm`.

Here `sm` has to be a submodule of `self.codomain` containing the image.

Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.restrict_codomain(F.submodule([1, 0]))
Matrix([
[1, x], : QQ[x]**2 -> <[1, 0]>
[0, 0]])
```

restrict_domain(*sm*)

Return `self`, with the domain restricted to `sm`.

Here `sm` has to be a submodule of `self.domain`.

Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.restrict_domain(F.submodule([1, 0]))
Matrix([
[1, x], : <[1, 0]> -> QQ[x]**2
[0, 0]])
```

This is the same as just composing on the right with the submodule inclusion:

```
>>> h * F.submodule([1, 0]).inclusion_hom()
Matrix([
[1, x], : <[1, 0]> -> QQ[x]**2
[0, 0]])
```

Finite Extensions

Let A be a (commutative) ring and B an extension ring of A. An element t of B is a generator of B (over A) if all elements of B can be represented as polynomials in t with coefficients in A. The representation is unique if and only if t satisfies no non-trivial polynomial relation, in which case B can be identified with a (univariate) polynomial ring over A.

The polynomials having t as a root form a non-zero ideal in general. The most important case in practice is that of an ideal generated by a single monic polynomial. If t satisfies such a polynomial relation, then its highest power t^n can be written as linear combination of lower powers. It follows, inductively, that all higher powers of t also have such a representation. Hence the lower powers t^i ($i = 0, \ldots, n-1$) form a basis of B, which is then called a finite extension of A, or, more precisely, a monogenic finite extension as it is generated by a single element t.

class sympy.polys.agca.extensions.**MonogenicFiniteExtension**(*mod*)

Finite extension generated by an integral element.

The generator is defined by a monic univariate polynomial derived from the argument `mod`.

A shorter alias is `FiniteExtension`.

Examples

Quadratic integer ring $\mathbb{Z}[\sqrt{2}]$:

```
>>> from sympy import Symbol, Poly
>>> from sympy.polys.agca.extensions import FiniteExtension
>>> x = Symbol('x')
>>> R = FiniteExtension(Poly(x**2 - 2)); R
ZZ[x]/(x**2 - 2)
>>> R.rank
2
>>> R(1 + x)*(3 - 2*x)
x - 1
```

Finite field $GF(5^3)$ defined by the primitive polynomial $x^3 + x^2 + 2$ (over $\mathbb{Z}_5$).

```
>>> F = FiniteExtension(Poly(x**3 + x**2 + 2, modulus=5)); F
GF(5)[x]/(x**3 + x**2 + 2)
>>> F.basis
(1, x, x**2)
>>> F(x + 3)/(x**2 + 2)
-2*x**2 + x + 2
```

Function field of an elliptic curve:

```
>>> t = Symbol('t')
>>> FiniteExtension(Poly(t**2 - x**3 - x + 1, t, field=True))
ZZ(x)[t]/(t**2 - x**3 - x + 1)
```

dtype

alias of *ExtensionElement* (page 2476)

class sympy.polys.agca.extensions.**ExtensionElement**(*rep*, *ext*)

Element of a finite extension.

A class of univariate polynomials modulo the `modulus` of the extension `ext`. It is represented by the unique polynomial `rep` of lowest degree. Both `rep` and the representation `mod` of `modulus` are of class DMP.

inverse()

Multiplicative inverse.

Raises

NotInvertible

If the element is a zero divisor.