Examples

```
>>> from sympy import ZZ
>>> from sympy import symbols
>>> from sympy.holonomic.holonomic import DifferentialOperators
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> R
Univariate Differential Operator Algebra in intermediate Dx over the␣
↪base ring
ZZ[x]
```

See also:

DifferentialOperator (page 2315)

Operations on holonomic functions

Addition and Multiplication

Two holonomic functions can be added or multiplied with the result also a holonomic functions.

```
>>> from sympy.holonomic.holonomic import HolonomicFunction,␣
↪DifferentialOperators
>>> from sympy.polys.domains import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x),'Dx')
```

p and q here are holonomic representation of e^x and $\sin(x)$ respectively.

```
>>> p = HolonomicFunction(Dx - 1, x, 0, [1])
>>> q = HolonomicFunction(Dx**2 + 1, x, 0, [0, 1])
```

Holonomic representation of $e^x + \sin(x)$

```
>>> p + q
HolonomicFunction((-1) + (1)*Dx + (-1)*Dx**2 + (1)*Dx**3, x, 0, [1,␣
↪2, 1])
```

Holonomic representation of $e^x \cdot \sin(x)$

```
>>> p * q
HolonomicFunction((2) + (-2)*Dx + (1)*Dx**2, x, 0, [0, 1])
```

Integration and Differentiation

HolonomicFunction.**integrate**(*limits*, *initcond=False*)

Integrates the given holonomic function.

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction,␣
↪DifferentialOperators
>>> from sympy import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x),'Dx')
>>> HolonomicFunction(Dx - 1, x, 0, [1]).integrate((x, 0, x))  # e^x - 1
HolonomicFunction((-1)*Dx + (1)*Dx**2, x, 0, [0, 1])
>>> HolonomicFunction(Dx**2 + 1, x, 0, [1, 0]).integrate((x, 0, x))
HolonomicFunction((1)*Dx + (1)*Dx**3, x, 0, [0, 1, 0])
```

HolonomicFunction.**diff**(**args*, ***kwargs*)

Differentiation of the given Holonomic function.

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction,␣
↪DifferentialOperators
>>> from sympy import ZZ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x),'Dx')
>>> HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]).diff().to_expr()
cos(x)
>>> HolonomicFunction(Dx - 2, x, 0, [1]).diff().to_expr()
2*exp(2*x)
```

See also:

integrate (page 2318)

Composition with polynomials

HolonomicFunction.**composition**(*expr*, **args*, ***kwargs*)

Returns function after composition of a holonomic function with an algebraic function. The method cannot compute initial conditions for the result by itself, so they can be also be provided.

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction,␣
↪DifferentialOperators
>>> from sympy import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x),'Dx')
>>> HolonomicFunction(Dx - 1, x).composition(x**2, 0, [1])  # e^(x**2)
HolonomicFunction((-2*x) + (1)*Dx, x, 0, [1])
>>> HolonomicFunction(Dx**2 + 1, x).composition(x**2 - 1, 1, [1, 0])
HolonomicFunction((4*x**3) + (-1)*Dx + (x)*Dx**2, x, 1, [1, 0])
```

See also:

from_hyper (page 2323)

Convert to holonomic sequence

HolonomicFunction.**to_sequence**(*lb=True*)

Finds recurrence relation for the coefficients in the series expansion of the function about x_0, where x_0 is the point at which the initial condition is stored.

Explanation

If the point x_0 is ordinary, solution of the form $[(R, n_0)]$ is returned. Where R is the recurrence relation and n_0 is the smallest n for which the recurrence holds true.

If the point x_0 is regular singular, a list of solutions in the format (R, p, n_0) is returned, i.e. $[(R, p, n_0), ...]$. Each tuple in this vector represents a recurrence relation R associated with a root of the indicial equation p. Conditions of a different format can also be provided in this case, see the docstring of HolonomicFunction class.

If it's not possible to numerically compute a initial condition, it is returned as a symbol C_j, denoting the coefficient of $(x - x_0)^j$ in the power series about x_0.

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction,␣
↪DifferentialOperators
>>> from sympy import QQ
>>> from sympy import symbols, S
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x),'Dx')
>>> HolonomicFunction(Dx - 1, x, 0, [1]).to_sequence()
[(HolonomicSequence((-1) + (n + 1)Sn, n), u(0) = 1, 0)]
>>> HolonomicFunction((1 + x)*Dx**2 + Dx, x, 0, [0, 1]).to_sequence()
[(HolonomicSequence((n**2) + (n**2 + n)Sn, n), u(0) = 0, u(1) = 1, u(2)␣
↪= -1/2, 2)]
>>> HolonomicFunction(-S(1)/2 + x*Dx, x, 0, {S(1)/2: [1]}).to_sequence()
[(HolonomicSequence((n), n), u(0) = 1, 1/2, 1)]
```

See also:

HolonomicFunction.series (page 2320)

References

[R523], [R524]

Series expansion

HolonomicFunction.**series**(*n=6*, *coefficient=False*, *order=True*, *_recur=None*)

Finds the power series expansion of given holonomic function about x_0.

Explanation

A list of series might be returned if x_0 is a regular point with multiple roots of the indicial equation.

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction,␣
↪DifferentialOperators
>>> from sympy import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x),'Dx')
>>> HolonomicFunction(Dx - 1, x, 0, [1]).series()  # e^x
1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + O(x**6)
>>> HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]).series(n=8)  # sin(x)
x - x**3/6 + x**5/120 - x**7/5040 + O(x**8)
```

See also:

HolonomicFunction.to_sequence (page 2319)

Numerical evaluation

HolonomicFunction.**evalf**(*points*, *method='RK4'*, *h=0.05*, *derivatives=False*)

Finds numerical value of a holonomic function using numerical methods. (RK4 by default). A set of points (real or complex) must be provided which will be the path for the numerical integration.

Explanation

The path should be given as a list $[x_1, x_2, \ldots x_n]$. The numerical values will be computed at each point in this order $x_1 \to x_2 \to x_3 \cdots \to x_n$.

Returns values of the function at $x_1, x_2, \ldots x_n$ in a list.

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction,␣
↪DifferentialOperators
>>> from sympy import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x),'Dx')
```

A straight line on the real axis from (0 to 1)

```
>>> r = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
```

Runge-Kutta 4th order on e^x from 0.1 to 1. Exact solution at 1 is 2.71828182845905

```
>>> HolonomicFunction(Dx - 1, x, 0, [1]).evalf(r)
[1.10517083333333, 1.22140257085069, 1.34985849706254, 1.49182424008069,
1.64872063859684, 1.82211796209193, 2.01375162659678, 2.22553956329232,
2.45960141378007, 2.71827974413517]
```

Euler's method for the same

```
>>> HolonomicFunction(Dx - 1, x, 0, [1]).evalf(r, method='Euler')
[1.1, 1.21, 1.331, 1.4641, 1.61051, 1.771561, 1.9487171, 2.14358881,
2.357947691, 2.5937424601]
```

One can also observe that the value obtained using Runge-Kutta 4th order is much more accurate than Euler's method.

Convert to a linear combination of hypergeometric functions

HolonomicFunction.**to_hyper**(*as_list=False*, *_recur=None*)

Returns a hypergeometric function (or linear combination of them) representing the given holonomic function.

Explanation

Returns an answer of the form: $a_1 \cdot x^{b_1} \cdot hyper() + a_2 \cdot x^{b_2} \cdot hyper() \ldots$

This is very useful as one can now use hyperexpand to find the symbolic expressions/functions.

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction,␣
↪DifferentialOperators
>>> from sympy import ZZ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x),'Dx')
>>> # sin(x)
>>> HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]).to_hyper()
x*hyper((), (3/2,), -x**2/4)
>>> # exp(x)
>>> HolonomicFunction(Dx - 1, x, 0, [1]).to_hyper()
hyper((), (), x)
```

See also:

from_hyper (page 2323), *from_meijerg* (page 2323)

Convert to a linear combination of Meijer G-functions

HolonomicFunction.**to_meijerg**()

Returns a linear combination of Meijer G-functions.

Examples

```
>>> from sympy.holonomic import expr_to_holonomic
>>> from sympy import sin, cos, hyperexpand, log, symbols
>>> x = symbols('x')
>>> hyperexpand(expr_to_holonomic(cos(x) + sin(x)).to_meijerg())
sin(x) + cos(x)
>>> hyperexpand(expr_to_holonomic(log(x)).to_meijerg()).simplify()
log(x)
```

See also:

to_hyper (page 2321)

Convert to expressions

HolonomicFunction.**to_expr**()

Converts a Holonomic Function back to elementary functions.

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction,␣
↪DifferentialOperators
>>> from sympy import ZZ
>>> from sympy import symbols, S
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x),'Dx')
>>> HolonomicFunction(x**2*Dx**2 + x*Dx + (x**2 - 1), x, 0, [0, S(1)/2]).
↪to_expr()
besselj(1, x)
>>> HolonomicFunction((1 + x)*Dx**3 + Dx**2, x, 0, [1, 1, 1]).to_expr()
x*log(x + 1) + log(x + 1) + 1
```

Converting other representations to holonomic

Converting hypergeometric functions

sympy.holonomic.holonomic.**from_hyper**(*func*, *x0=0*, *evalf=False*)

Converts a hypergeometric function to holonomic. `func` is the Hypergeometric Function and `x0` is the point at which initial conditions are required.

Examples

```
>>> from sympy.holonomic.holonomic import from_hyper
>>> from sympy import symbols, hyper, S
>>> x = symbols('x')
>>> from_hyper(hyper([], [S(3)/2], x**2/4))
HolonomicFunction((-x) + (2)*Dx + (x)*Dx**2, x, 1, [sinh(1), -sinh(1) +␣
↪cosh(1)])
```

Converting Meijer G-functions

sympy.holonomic.holonomic.**from_meijerg**(*func*, *x0=0*, *evalf=False*, *initcond=True*, *domain=QQ*)

Converts a Meijer G-function to Holonomic. `func` is the G-Function and `x0` is the point at which initial conditions are required.

Examples

```
>>> from sympy.holonomic.holonomic import from_meijerg
>>> from sympy import symbols, meijerg, S
>>> x = symbols('x')
>>> from_meijerg(meijerg(([], []), ([S(1)/2], [0]), x**2/4))
HolonomicFunction((1) + (1)*Dx**2, x, 0, [0, 1/sqrt(pi)])
```

Converting symbolic expressions

sympy.holonomic.holonomic.**expr_to_holonomic**(*func*, *x=None*, *x0=0*, *y0=None*, *lenics=None*, *domain=None*, *initcond=True*)

Converts a function or an expression to a holonomic function.

Parameters

func:

The expression to be converted.

x:

variable for the function.

x0:

point at which initial condition must be computed.

y0:

One can optionally provide initial condition if the method is not able to do it automatically.

lenics:

Number of terms in the initial condition. By default it is equal to the order of the annihilator.

domain:

Ground domain for the polynomials in `x` appearing as coefficients in the annihilator.

initcond:

Set it false if you do not want the initial conditions to be computed.

Examples

```
>>> from sympy.holonomic.holonomic import expr_to_holonomic
>>> from sympy import sin, exp, symbols
>>> x = symbols('x')
>>> expr_to_holonomic(sin(x))
HolonomicFunction((1) + (1)*Dx**2, x, 0, [0, 1])
>>> expr_to_holonomic(exp(x))
HolonomicFunction((-1) + (1)*Dx, x, 0, [1])
```

See also:

sympy.integrals.meijerint._rewrite1 (page 572), *_convert_poly_rat_alg* (page 2326), *_create_table* (page 2326)

Uses and Current limitations

Integration

One can perform integrations using holonomic functions by following these steps:

1. Convert the integrand to a holonomic function.
2. Now integrate the holonomic representation of the function.
3. Convert the integral back to expressions.

Examples

```
>>> from sympy.abc import x, a
>>> from sympy import sin
>>> from sympy.holonomic import expr_to_holonomic
>>> expr_to_holonomic(1/(x**2+a), x).integrate(x).to_expr()
atan(x/sqrt(a))/sqrt(a)
>>> expr_to_holonomic(sin(x)/x).integrate(x).to_expr()
Si(x)
```

As you can see in the first example we converted the function to holonomic, integrated the result and then converted back to symbolic expression.

Limitations

1. Converting to expressions is not always possible. The holonomic function should have a hypergeometric series at `x0`.

2. Implementation of converting to holonomic sequence currently doesn't support `Frobenius method` when the solutions need to have log terms. This happens when at least one pair of the roots of the indicial equation differ by an integer and frobenius method yields linearly dependent series solutions. Since we use this while converting to expressions, sometimes *to_expr()* (page 2323) fails.

3. There doesn't seem to be a way for computing indefinite integrals, so *integrate()* (page 2318) basically computes $\int_{x_0}^{x} f(x)dx$ if no limits are given, where x_0 is the point at which initial conditions for the integrand are stored. Sometimes this gives an additional constant in the result. For instance:

```
>>> expr_to_holonomic(sin(x)).integrate(x).to_expr()
1 - cos(x)
>>> sin(x).integrate(x)
-cos(x)
```

The indefinite integral of $\sin(x)$ is $-\cos(x)$. But the output is $-\cos(x)+1$ which is $\int_0^x sin(x)dx$. Although both are considered correct but $-\cos(x)$ is simpler.

Internal API

sympy.holonomic.holonomic.**_create_table**(*table*, *domain=QQ*)

Creates the look-up table. For a similar implementation see meijerint._create_lookup_table.

sympy.holonomic.holonomic.**_convert_poly_rat_alg**(*func*, *x*, *x0=0*, *y0=None*, *lenics=None*, *domain=QQ*, *initcond=True*)

Converts polynomials, rationals and algebraic functions to holonomic.

Lie Algebra

class sympy.liealgebras.root_system.**RootSystem**(*cartantype*)

Represent the root system of a simple Lie algebra

Every simple Lie algebra has a unique root system. To find the root system, we first consider the Cartan subalgebra of g, which is the maximal abelian subalgebra, and consider the adjoint action of g on this subalgebra. There is a root system associated with this action. Now, a root system over a vector space V is a set of finite vectors Phi (called roots), which satisfy:

1. The roots span V
2. The only scalar multiples of x in Phi are x and -x
3. For every x in Phi, the set Phi is closed under reflection through the hyperplane perpendicular to x.
4. If x and y are roots in Phi, then the projection of y onto the line through x is a half-integral multiple of x.

Now, there is a subset of Phi, which we will call Delta, such that: 1. Delta is a basis of V 2. Each root x in Phi can be written x = sum k_y y for y in Delta

The elements of Delta are called the simple roots. Therefore, we see that the simple roots span the root space of a given simple Lie algebra.

References

[R555], [R556]

add_as_roots(*root1*, *root2*)

Add two roots together if and only if their sum is also a root

It takes as input two vectors which should be roots. It then computes their sum and checks if it is in the list of all possible roots. If it is, it returns the sum. Otherwise it returns a string saying that the sum is not a root.

Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> c.add_as_roots([1, 0, -1, 0], [0, 0, 1, -1])
[1, 0, 0, -1]
>>> c.add_as_roots([1, -1, 0, 0], [0, 0, -1, 1])
'The sum of these two roots is not a root'
```

add_simple_roots(*root1*, *root2*)

Add two simple roots together

The function takes as input two integers, root1 and root2. It then uses these integers as keys in the dictionary of simple roots, and gets the corresponding simple roots, and then adds them together.

Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> newroot = c.add_simple_roots(1, 2)
>>> newroot
[1, 0, -1, 0]
```

all_roots()

Generate all the roots of a given root system

The result is a dictionary where the keys are integer numbers. It generates the roots by getting the dictionary of all positive roots from the bases classes, and then taking each root, and multiplying it by -1 and adding it to the dictionary. In this way all the negative roots are generated.

cartan_matrix()

Cartan matrix of Lie algebra associated with this root system

Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> c.cartan_matrix()
Matrix([
    [ 2, -1,  0],
    [-1,  2, -1],
    [ 0, -1,  2]])
```

dynkin_diagram()

Dynkin diagram of the Lie algebra associated with this root system

Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> print(c.dynkin_diagram())
0---0---0
1   2   3
```

root_space()

Return the span of the simple roots

The root space is the vector space spanned by the simple roots, i.e. it is a vector space with a distinguished basis, the simple roots. This method returns a string that represents the root space as the span of the simple roots, alpha[1],...., alpha[n].

Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> c.root_space()
'alpha[1] + alpha[2] + alpha[3]'
```

simple_roots()

Generate the simple roots of the Lie algebra

The rank of the Lie algebra determines the number of simple roots that it has. This method obtains the rank of the Lie algebra, and then uses the simple_root method from the Lie algebra classes to generate all the simple roots.

Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> roots = c.simple_roots()
>>> roots
{1: [1, -1, 0, 0], 2: [0, 1, -1, 0], 3: [0, 0, 1, -1]}
```

class sympy.liealgebras.type_a.**TypeA**(*n*)

This class contains the information about the A series of simple Lie algebras. ====

basic_root(*i*, *j*)

This is a method just to generate roots with a 1 iin the ith position and a -1 in the jth position.

basis()

Returns the number of independent generators of A_n

cartan_matrix()

Returns the Cartan matrix for A_n. The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], ..., alpha[l]). Then the ijth entry of the Cartan matrix is (<alpha[i],alpha[j]>).

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A4')
>>> c.cartan_matrix()
Matrix([
[ 2, -1,  0,  0],
[-1,  2, -1,  0],
[ 0, -1,  2, -1],
[ 0,  0, -1,  2]])
```

dimension()

Dimension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A4")
>>> c.dimension()
5
```

highest_root()

Returns the highest weight root for A_n

lie_algebra()

Returns the Lie algebra associated with A_n

positive_roots()

This method generates all the positive roots of A_n. This is half of all of the roots of A_n; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1,␣
↪0],
        5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots for A_n

simple_root(*i*)

Every lie algebra has a unique root system. Given a root system Q, there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q. If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

In A_n the ith simple root is the root which has a 1 in the ith position, a -1 in the (i+1)th position, and zeroes elsewhere.

This method returns the ith simple root for the A series.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A4")
>>> c.simple_root(1)
[1, -1, 0, 0, 0]
```

class sympy.liealgebras.type_b.**TypeB**(*n*)

basic_root(*i*, *j*)

This is a method just to generate roots with a 1 iin the ith position and a -1 in the jth position.

basis()

Returns the number of independent generators of B_n

cartan_matrix()

Returns the Cartan matrix for B_n. The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], ..., alpha[l]). Then the ijth entry of the Cartan matrix is (<alpha[i],alpha[j]>).

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('B4')
>>> c.cartan_matrix()
Matrix([
[ 2, -1,  0,  0],
[-1,  2, -1,  0],
[ 0, -1,  2, -2],
[ 0,  0, -1,  2]])
```

dimension()

Dimension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("B3")
>>> c.dimension()
3
```

lie_algebra()

Returns the Lie algebra associated with B_n

positive_roots()

This method generates all the positive roots of A_n. This is half of all of the roots of B_n; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1,␣
↪0],
        5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots for B_n"

simple_root(*i*)

Every lie algebra has a unique root system. Given a root system Q, there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q. If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

In B_n the first n-1 simple roots are the same as the roots in A_(n-1) (a 1 in the ith position, a -1 in the (i+1)th position, and zeroes elsewhere). The n-th simple root is the root with a 1 in the nth position and zeroes elsewhere.

This method returns the ith simple root for the B series.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("B3")
>>> c.simple_root(2)
[0, 1, -1]
```

class sympy.liealgebras.type_c.**TypeC**(*n*)

basic_root(*i*, *j*)

Generate roots with 1 in ith position and a -1 in jth position

basis()

Returns the number of independent generators of C_n

cartan_matrix()

The Cartan matrix for C_n

The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], ..., alpha[l]). Then the ijth entry of the Cartan matrix is (<alpha[i],alpha[j]>).

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('C4')
>>> c.cartan_matrix()
Matrix([
[ 2, -1,  0,  0],
[-1,  2, -1,  0],
[ 0, -1,  2, -1],
[ 0,  0, -2,  2]])
```

dimension()

Dimension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("C3")
>>> c.dimension()
3
```

lie_algebra()

Returns the Lie algebra associated with C_n"

positive_roots()

Generates all the positive roots of A_n

This is half of all of the roots of C_n; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1,␣
↪0],
        5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots for C_n"

simple_root(*i*)

The ith simple root for the C series

Every lie algebra has a unique root system. Given a root system Q, there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q. If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

In C_n, the first n-1 simple roots are the same as the roots in A_(n-1) (a 1 in the ith position, a -1 in the (i+1)th position, and zeroes elsewhere). The nth simple root is the root in which there is a 2 in the nth position and zeroes elsewhere.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("C3")
>>> c.simple_root(2)
[0, 1, -1]
```

class sympy.liealgebras.type_d.**TypeD**(*n*)

basic_root(*i*, *j*)

This is a method just to generate roots with a 1 iin the ith position and a -1 in the jth position.

basis()

Returns the number of independent generators of D_n

cartan_matrix()

Returns the Cartan matrix for D_n. The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], …., alpha[l]). Then the ijth entry of the Cartan matrix is (<alpha[i],alpha[j]>).

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('D4')
>>> c.cartan_matrix()
    Matrix([
    [ 2, -1,  0,  0],
    [-1,  2, -1, -1],
    [ 0, -1,  2,  0],
    [ 0, -1,  0,  2]])
```

dimension()

Dmension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("D4")
>>> c.dimension()
4
```

lie_algebra()

Returns the Lie algebra associated with D_n"

positive_roots()

This method generates all the positive roots of A_n. This is half of all of the roots of D_n by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1,␣
↪0],
        5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots for D_n"

simple_root(*i*)

Every lie algebra has a unique root system. Given a root system Q, there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q. If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

In D_n, the first n-1 simple roots are the same as the roots in A_(n-1) (a 1 in the ith position, a -1 in the (i+1)th position, and zeroes elsewhere). The nth simple root is the root in which there 1s in the nth and (n-1)th positions, and zeroes elsewhere.

This method returns the ith simple root for the D series.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("D4")
>>> c.simple_root(2)
[0, 1, -1, 0]
```

class sympy.liealgebras.type_e.**TypeE**(*n*)

basic_root(*i*, *j*)

This is a method just to generate roots with a -1 in the ith position and a 1 in the jth position.

basis()

Returns the number of independent generators of E_n

cartan_matrix()

Returns the Cartan matrix for G_2 The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], ..., alpha[l]). Then the ijth entry of the Cartan matrix is (<alpha[i],alpha[j]>).

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A4')
>>> c.cartan_matrix()
Matrix([
[ 2, -1,  0,  0],
[-1,  2, -1,  0],
[ 0, -1,  2, -1],
[ 0,  0, -1,  2]])
```

dimension()

Dimension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("E6")
>>> c.dimension()
8
```

positive_roots()

This method generates all the positive roots of A_n. This is half of all of the roots of E_n; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1,␣
↪0],
        5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots of E_n

simple_root(*i*)

Every lie algebra has a unique root system. Given a root system Q, there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q. If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

This method returns the ith simple root for E_n.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("E6")
>>> c.simple_root(2)
[1, 1, 0, 0, 0, 0, 0, 0]
```

class sympy.liealgebras.type_f.**TypeF**(*n*)

basic_root(*i*, *j*)

Generate roots with 1 in ith position and -1 in jth position

basis()

Returns the number of independent generators of F_4

cartan_matrix()

The Cartan matrix for F_4

The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], ..., alpha[l]). Then the ijth entry of the Cartan matrix is (<alpha[i],alpha[j]>).

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A4')
>>> c.cartan_matrix()
Matrix([
[ 2, -1,  0,  0],
[-1,  2, -1,  0],
[ 0, -1,  2, -1],
[ 0,  0, -1,  2]])
```

dimension()

Dimension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("F4")
>>> c.dimension()
4
```

positive_roots()

Generate all the positive roots of A_n

This is half of all of the roots of F_4; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1,␣
↪0],
        5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots for F_4

simple_root(*i*)

The ith simple root of F_4

Every lie algebra has a unique root system. Given a root system Q, there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q. If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("F4")
>>> c.simple_root(3)
[0, 0, 0, 1]
```

class sympy.liealgebras.type_g.**TypeG**(*n*)

basis()

Returns the number of independent generators of G_2

cartan_matrix()

The Cartan matrix for G_2

The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], ..., alpha[l]). Then the ijth entry of the Cartan matrix is (<alpha[i],alpha[j]>).

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("G2")
>>> c.cartan_matrix()
Matrix([
    [ 2, -1],
    [-3,  2]])
```

dimension()

Dimension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("G2")
>>> c.dimension()
3
```

positive_roots()

Generate all the positive roots of A_n

This is half of all of the roots of A_n; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1,␣
↪0],
        5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots of G_2"

simple_root(*i*)

The ith simple root of G_2

Every lie algebra has a unique root system. Given a root system Q, there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q. If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("G2")
>>> c.simple_root(1)
[0, 1, -1]
```

class sympy.liealgebras.weyl_group.**WeylGroup**(*cartantype*)

For each semisimple Lie group, we have a Weyl group. It is a subgroup of the isometry group of the root system. Specifically, it's the subgroup that is generated by reflections through the hyperplanes orthogonal to the roots. Therefore, Weyl groups are reflection groups, and so a Weyl group is a finite Coxeter group.

coxeter_diagram()

This method returns the Coxeter diagram corresponding to a Weyl group. The Coxeter diagram can be obtained from a Lie algebra's Dynkin diagram by deleting all arrows; the Coxeter diagram is the undirected graph. The vertices of the Coxeter diagram represent the generating reflections of the Weyl group, s_i. An edge is drawn

between s_i and s_j if the order $m(i,j)$ of $s_i s_j$ is greater than two. If there is one edge, the order $m(i,j)$ is 3. If there are two edges, the order $m(i,j)$ is 4, and if there are three edges, the order $m(i,j)$ is 6.

Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> c = WeylGroup("B3")
>>> print(c.coxeter_diagram())
0---0===0
1   2   3
```

delete_doubles(*reflections*)

This is a helper method for determining the order of an element in the Weyl group of G2. It takes a Weyl element and if repeated simple reflections in it, it deletes them.

element_order(*weylelt*)

This method returns the order of a given Weyl group element, which should be specified by the user in the form of products of the generating reflections, i.e. of the form r1*r2 etc.

For types A-F, this method current works by taking the matrix form of the specified element, and then finding what power of the matrix is the identity. It then returns this power.

Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> b = WeylGroup("B4")
>>> b.element_order('r1*r4*r2')
4
```

generators()

This method creates the generating reflections of the Weyl group for a given Lie algebra. For a Lie algebra of rank n, there are n different generating reflections. This function returns them as a list.

Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> c = WeylGroup("F4")
>>> c.generators()
['r1', 'r2', 'r3', 'r4']
```

group_name()

This method returns some general information about the Weyl group for a given Lie algebra. It returns the name of the group and the elements it acts on, if relevant.

group_order()

This method returns the order of the Weyl group. For types A, B, C, D, and E the order depends on the rank of the Lie algebra. For types F and G, the order is fixed.

Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> c = WeylGroup("D4")
>>> c.group_order()
192.0
```

matrix_form(*weylelt*)

This method takes input from the user in the form of products of the generating reflections, and returns the matrix corresponding to the element of the Weyl group. Since each element of the Weyl group is a reflection of some type, there is a corresponding matrix representation. This method uses the standard representation for all the generating reflections.

Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> f = WeylGroup("F4")
>>> f.matrix_form('r2*r3')
Matrix([
[1, 0, 0,  0],
[0, 1, 0,  0],
[0, 0, 0, -1],
[0, 0, 1,  0]])
```

class sympy.liealgebras.cartan_type.**CartanType_generator**

Constructor for actually creating things

class sympy.liealgebras.cartan_type.**Standard_Cartan**(*series*, *n*)

Concrete base class for Cartan types such as A4, etc

rank()

Returns the rank of the Lie algebra

series()

Returns the type of the Lie algebra

sympy.liealgebras.dynkin_diagram.**DynkinDiagram**(*t*)

Display the Dynkin diagram of a given Lie algebra

Works by generating the CartanType for the input, t, and then returning the Dynkin diagram method from the individual classes.

Examples

```
>>> from sympy.liealgebras.dynkin_diagram import DynkinDiagram
>>> print(DynkinDiagram("A3"))
0---0---0
1   2   3
```

```
>>> print(DynkinDiagram("B4"))
0---0---0=>=0
1   2   3   4
```

sympy.liealgebras.cartan_matrix.**CartanMatrix**(*ct*)

Access the Cartan matrix of a specific Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_matrix import CartanMatrix
>>> CartanMatrix("A2")
Matrix([
[ 2, -1],
[-1,  2]])
```

```
>>> CartanMatrix(['C', 3])
Matrix([
[ 2, -1,  0],
[-1,  2, -1],
[ 0, -2,  2]])
```

This method works by returning the Cartan matrix which corresponds to Cartan type t.

Polynomial Manipulation

Computations with polynomials are at the core of computer algebra and having a fast and robust polynomials manipulation module is a key for building a powerful symbolic manipulation system. SymPy has a dedicated module *sympy.polys* (page 2360) for computing in polynomial algebras over various coefficient domains.

There is a vast number of methods implemented, ranging from simple tools like polynomial division, to advanced concepts including Gröbner bases and multivariate factorization over algebraic number domains.

Contents

Basic functionality of the module

Introduction

This tutorial tries to give an overview of the functionality concerning polynomials within SymPy. All code examples assume:

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

Basic concepts

Polynomials

Given a family (x_i) of symbols, or other suitable objects, including numbers, expressions derived from them by repeated addition, subtraction and multiplication are called *polynomial expressions in the generators* x_i.

By the distributive law it is possible to perform multiplications before additions and subtractions. The products of generators thus obtained are called *monomials*. They are usually written in the form $x_1^{\nu_1} x_2^{\nu_2} \cdots x_n^{\nu_n}$ where the exponents ν_i are nonnegative integers. It is often convenient to write this briefly as x^ν where $x = (x_1, x_2, \ldots, x_n)$ denotes the family of generators and $\nu = (\nu_1, \nu_2, \ldots, \nu_n)$ is the family of exponents.

When all monomials having the same exponents are combined, the polynomial expression becomes a sum of products $c_\nu x^\nu$, called the *terms* of the polynomial, where the *coefficients* c_ν are integers. If some of the x_i are manifest numbers, they are incorporated in the coefficients and not regarded as generators. Such coefficients are typically rational, real or complex numbers. Some symbolic numbers, e.g., `pi`, can be either coefficients or generators.

A polynomial expression that is a sum of terms with different monomials is uniquely determined by its family of coefficients (c_ν). Such an expression is customarily called a *polynomial*, though, more properly, that name does stand for the coefficient family once the generators are given. SymPy implements polynomials by default as dictionaries with monomials as keys and coefficients as values. Another implementation consists of nested lists of coefficients.

The set of all polynomials with integer coefficients in the generators x_i is a *ring*, i.e., the sums, differences and products of its elements are again polynomials in the same generators. This ring is denoted $\mathbb{Z}[x_1, x_2, \ldots, x_n]$, or $\mathbb{Z}[(x_i)]$, and called the *ring of polynomials in the* x_i *with integer coefficients*.

More generally, the coefficients of a polynomial can be elements of any commutative ring A, and the corresponding polynomial ring is then denoted $A[x_1, x_2, \ldots, x_n]$. The ring A can also be a polynomial ring. In SymPy, the coefficient ring is called the `domain` of the polynomial ring, and it can be given as a keyword parameter. By default, it is determined by the coefficients of the polynomial arguments.

Polynomial expressions can be transformed into polynomials by the method *sympy.core.expr.Expr.as_poly* (page 956):

```
>>> e = (x + y)*(y - 2*z)
>>> e.as_poly()
Poly(x*y - 2*x*z + y**2 - 2*y*z, x, y, z, domain='ZZ')
```

If a polynomial expression contains numbers that are not integers, they are regarded as coefficients and the coefficient ring is extended accordingly. In particular, division by integers leads to rational coefficients:

```
>>> e = (3*x/2 + y)*(z - 1)
>>> e.as_poly()
Poly(3/2*x*z - 3/2*x + y*z - y, x, y, z, domain='QQ')
```

Symbolic numbers are considered generators unless they are explicitly excluded, in which case they are adjoined to the coefficient ring:

```
>>> e = (x + 2*pi)*y
>>> e.as_poly()
Poly(x*y + 2*y*pi, x, y, pi, domain='ZZ')
>>> e.as_poly(x, y)
Poly(x*y + 2*pi*y, x, y, domain='ZZ[pi]')
```

Alternatively, the coefficient domain can be specified by means of a keyword argument:

```
>>> e = (x + 2*pi)*y
>>> e.as_poly(domain=ZZ[pi])
Poly(x*y + 2*pi*y, x, y, domain='ZZ[pi]')
```

Note that the ring $\mathbb{Z}[\pi][x, y]$ of polynomials in x and y with coefficients in $\mathbb{Z}[\pi]$ is mathematically equivalent to $\mathbb{Z}[\pi, x, y]$, only their implementations differ.

If an expression contains functions of the generators, other than their positive integer powers, these are interpreted as new generators:

```
>>> e = x*sin(y) - y
>>> e.as_poly()
Poly(x*(sin(y)) - y, x, y, sin(y), domain='ZZ')
```

Since y and $\sin(y)$ are algebraically independent they can both appear as generators in a polynomial. However, *polynomial expressions must not contain negative powers of generators*:

```
>>> e = x - 1/x
>>> e.as_poly()
Poly(x - (1/x), x, 1/x, domain='ZZ')
```

It is important to realize that the generators x and $1/x = x^{-1}$ are treated as algebraically independent variables. In particular, their product is not equal to 1. Hence *generators in denominators should be avoided even if they raise no error in the current implementation*. This behavior is undesirable and may change in the future. Similar problems emerge with rational powers of generators. So, for example, x and $\sqrt{x} = x^{1/2}$ are not recognized as algebraically dependent.

If there are algebraic numbers in an expression, it is possible to adjoin them to the coefficient ring by setting the keyword `extension`:

```
>>> e = x + sqrt(2)
>>> e.as_poly()
Poly(x + (sqrt(2)), x, sqrt(2), domain='ZZ')
>>> e.as_poly(extension=True)
Poly(x + sqrt(2), x, domain='QQ<sqrt(2)>')
```

With the default setting `extension=False`, both x and $\sqrt{2}$ are incorrectly considered algebraically independent variables. With coefficients in the extension field $\mathbb{Q}(\sqrt{2})$ the square root is treated properly as an algebraic number. Setting `extension=True` whenever algebraic numbers are involved is definitely recommended even though it is not forced in the current implementation.

Divisibility

The fourth rational operation, division, or inverted multiplication, is not generally possible in rings. If a and b are two elements of a ring A, then there may exist a third element q in A such that $a = bq$. In fact, there may exist several such elements.

If also $a = bq'$ for some q' in A, then $b(q - q') = 0$. Hence either b or $q - q'$ is zero, or they are both *zero divisors*, nonzero elements whose product is zero.

Integral domains

Commutative rings with no zero divisors are called *integral domains*. Most of the commonly encountered rings, the ring of integers, fields, and polynomial rings over integral domains are integral domains.

Assume now that A is an integral domain, and consider the set P of its nonzero elements, which is closed under multiplication. If a and b are in P, and there exists an element q in P such that $a = bq$, then q is unique and called the *quotient*, a/b, of a by b. Moreover, it is said that

- a is *divisible* by b,
- b is a *divisor* of a,
- a is a *multiple* of b,
- b is a *factor* of a.

An element a of P is a divisor of 1 if and only if it is *invertible* in A, with the inverse $a^{-1} = 1/a$. Such elements are called *units*. The units of the ring of integers are 1 and -1. The invertible elements in a polynomial ring over a field are the nonzero constant polynomials.

If two elements of P, a and b, are divisible by each other, then the quotient a/b is invertible with inverse b/a, or equivalently, $b = ua$ where u is a unit. Such elements are said to be *associated* with, or *associates* of, each other. The associates of an integer n are n and $-n$. In a polynomial ring over a field the associates of a polynomial are its constant multiples.

Each element of P is divisible by its associates and the units. An element is *irreducible* if it has no other divisors and is not a unit. The irreducible elements in the ring of integers are the prime numbers p and their opposites $-p$. In a field, every nonzero element is invertible and there are no irreducible elements.

Factorial domains

In the ring of integers, each nonzero element can be represented as a product of irreducible elements and optionally a unit ± 1. Moreover, any two such products have the same number of irreducible factors which are associated with each other in a suitable order. Integral domains having this property are called *factorial*, or *unique factorization domains*. In addition to the ring of integers, all polynomial rings over a field are factorial, and so are more generally polynomial rings over any factorial domain. Fields are trivially factorial since there are only units. The irreducible elements of a factorial domain are usually called *primes*.

A family of integers has only a finite number of common divisors and the greatest of them is divisible by all of them. More generally, given a family of nonzero elements (a_i) in an integral domain, a common divisor d of the elements is called a *greatest common divisor*, abbreviated *gcd*, of the family if it is a multiple of all common divisors. A greatest common divisor, if it exists, is not unique in general; all of its associates have the same property. It is denoted by $d = \gcd(a_1, \ldots, a_n)$ if there is no danger of confusion. A *least common multiple*, or *lcm*, of a family (a_i) is defined analogously as a common multiple m that divides all common multiples. It is denoted by $m = \operatorname{lcm}(a_1, \ldots, a_n)$.

In a factorial domain, greatest common divisors always exists. They can be found, at least in principle, by factoring each element of a family into a product of prime powers and an optional unit, and, for each prime, taking the least power that appears in the factorizations. The product of these prime powers is then a greatest common divisor. A least common multiple can be obtained from the same factorizations as the product of the greatest powers for each prime.

Euclidean domains

A practical algorithm for computing a greatest common divisor can be implemented in *Euclidean domains*. They are integral domains that can be endowed with a function w assigning a nonnegative integer to each nonzero element of the domain and having the following property:

> if a and b are nonzero, there are q and r that satisfy the *division identity*
>
> $$a = qb + r$$
>
> such that either $r = 0$ or $w(r) < w(b)$.

The ring of integers and all univariate polynomial rings over fields are Euclidean domains with $w(a) = |a|$ resp. $w(a) = \deg(a)$.

The division identity for integers is implemented in Python as the built-in function `divmod` that can also be applied to SymPy Integers:

```
>>> divmod(Integer(53), Integer(7))
(7, 4)
```

For polynomials the division identity is given in SymPy by the function *div()* (page 2363):

```
>>> f = 5*x**2 + 10*x + 3
>>> g = 2*x + 2

>>> q, r = div(f, g, domain='QQ')
>>> q
```

(continues on next page)

(continued from previous page)

```
5*x   5
--- + -
 2    2
>>> r
-2
>>> (q*g + r).expand()
   2
5*x  + 10*x + 3
```

The division identity can be used to determine the divisibility of elements in a Euclidean domain. If $r = 0$ in the division identity, then a is divisible by b. Conversely, if $a = cb$ for some element c, then $(c - q)b = r$. It follows that $c = q$ and $r = 0$ if w has the additional property:

> if a and b are nonzero, then $w(ab) \geq w(b)$.

This is satisfied by the functions given above. (And it is always possible to redefine $w(a)$ by taking the minimum of the values $w(xa)$ for $x \neq 0$.)

The principal application of the division identity is the efficient computation of a greatest common divisor by means of the Euclidean algorithm. It applies to two elements of a Euclidean domain. A gcd of several elements can be obtained by iteration.

The function for computing the greatest common divisor of integers in SymPy is currently *igcd()* (page 994):

```
>>> igcd(2, 4)
2
>>> igcd(5, 10, 15)
5
```

For univariate polynomials over a field the function has its common name *gcd()* (page 2368), and the returned polynomial is monic:

```
>>> f = 4*x**2 - 1
>>> g = 8*x**3 + 1
>>> gcd(f, g, domain=QQ)
x + 1/2
```

Divisibility of polynomials

The ring $A = \mathbb{Z}[x]$ of univariate polynomials over the ring of integers is not Euclidean but it is still factorial. To see this, consider the divisibility in A.

Let f and g be two nonzero polynomials in A. If f is divisible by g in A, then it is also divisible in the ring $B = \mathbb{Q}[x]$ of polynomials with rational coefficients. Since B is Euclidean, this can be determined by means of the division identity.

Assume, conversely, that $f = gh$ for some polynomial h in B. Then f is divisible by g in A if and only if the coefficients of h are integers. To find out when this is true it is necessary to consider the divisibility of the coefficients.

For a polynomial f in A, let c be the greatest common divisor of its coefficients. Then f is divisible by the constant polynomial c in A, and the quotient $f/c = p$ is a polynomial whose coefficients are integers that have no common divisor apart from the units. Such polynomials are called *primitive*. A polynomial with rational coefficients can also be written as $f = cp$,

where c is a rational number and p is a primitive polynomial. The constant c is called the *content* of f, and p is its *primitive part*. These components can be found by the method *sympy.core.expr.Expr.as_content_primitive* (page 951):

```
>>> f = 6*x**2 - 3*x + 9
>>> c, p = f.as_content_primitive()
>>> c, p
      2
(3, 2*x  - x + 3)
>>> f = x**2/3 - x/2 + 1
>>> c, p = f.as_content_primitive()
>>> c, p
        2
(1/6, 2*x  - 3*x + 6)
```

Let f, f' be polynomials with contents c, c' and primitive parts p, p'. Then $ff' = (cc')(pp')$ where the product pp' is primitive by Gauss's lemma. It follows that

> the content of a product of polynomials is the product of their contents and the primitive part of the product is the product of the primitive parts.

Returning to the divisibility in the ring $\mathbb{Z}[x]$, assume that f and g are two polynomials with integer coefficients such that the division identity in $\mathbb{Q}[x]$ yields the equality $f = gh$ for some polynomial h with rational coefficients. Then the content of f is equal to the content of g multiplied by the content of h. As h has integer coefficients if and only if its content is an integer, we get the following criterion:

> f is divisible by g in the ring $\mathbb{Z}[x]$ if and only if
>
> i. f is divisible by g in $\mathbb{Q}[x]$, and
>
> ii. the content of f is divisible by the content of g in $\mathbb{Z}$.

If $f = cp$ is irreducible in $\mathbb{Z}[x]$, then either c or p must be a unit. If p is not a unit, it must be irreducible also in $\mathbb{Q}[x]$. For if it is a product of two polynomials, it is also the product of their primitive parts, and one of them must be a unit. Hence there are two kinds of irreducible elements in $\mathbb{Z}[x]$:

i. prime numbers of $\mathbb{Z}$, and

ii. primitive polynomials that are irreducible in $\mathbb{Q}[x]$.

It follows that each polynomial in $\mathbb{Z}[x]$ is a product of irreducible elements. It suffices to factor its content and primitive part separately. These products are essentially unique; hence $\mathbb{Z}[x]$ is also factorial.

Another important consequence is that a greatest common divisor of two polynomials in $\mathbb{Z}[x]$ can be found efficiently by applying the Euclidean algorithm separately to their contents and primitive parts in the Euclidean domains $\mathbb{Z}$ and $\mathbb{Q}[x]$. This is also implemented in SymPy:

```
>>> f = 4*x**2 - 1
>>> g = 8*x**3 + 1
>>> gcd(f, g)
2*x + 1
>>> gcd(6*f, 3*g)
6*x + 3
```

Basic functionality

These functions provide different algorithms dealing with polynomials in the form of SymPy expression, like symbols, sums etc.

Division

The function *div()* (page 2363) provides division of polynomials with remainder. That is, for polynomials `f` and `g`, it computes `q` and `r`, such that $f = g \cdot q + r$ and $\deg(r) < \deg(q)$. For polynomials in one variables with coefficients in a field, say, the rational numbers, `q` and `r` are uniquely defined this way:

```
>>> f = 5*x**2 + 10*x + 3
>>> g = 2*x + 2

>>> q, r = div(f, g, domain='QQ')
>>> q
5*x   5
--- + -
 2    2
>>> r
-2
>>> (q*g + r).expand()
   2
5*x  + 10*x + 3
```

As you can see, `q` has a non-integer coefficient. If you want to do division only in the ring of polynomials with integer coefficients, you can specify an additional parameter:

```
>>> q, r = div(f, g, domain='ZZ')
>>> q
0
>>> r
   2
5*x  + 10*x + 3
```

But be warned, that this ring is no longer Euclidean and that the degree of the remainder doesn't need to be smaller than that of `f`. Since 2 doesn't divide 5, $2x$ doesn't divide $5x^2$, even if the degree is smaller. But:

```
>>> g = 5*x + 1

>>> q, r = div(f, g, domain='ZZ')
>>> q
x
>>> r
9*x + 3
>>> (q*g + r).expand()
   2
5*x  + 10*x + 3
```

This also works for polynomials with multiple variables:

```
>>> f = x*y + y*z
>>> g = 3*x + 3*z

>>> q, r = div(f, g, domain='QQ')
>>> q
y
-
3
>>> r
0
```

In the last examples, all of the three variables `x`, `y` and `z` are assumed to be variables of the polynomials. But if you have some unrelated constant as coefficient, you can specify the variables explicitly:

```
>>> a, b, c = symbols('a,b,c')
>>> f = a*x**2 + b*x + c
>>> g = 3*x + 2
>>> q, r = div(f, g, domain='QQ')
>>> q
a*x   2*a   b
--- - --- + -
 3     9    3

>>> r
4*a   2*b
--- - --- + c
 9     3
```


GCD and LCM

With division, there is also the computation of the greatest common divisor and the least common multiple.

When the polynomials have integer coefficients, the contents' gcd is also considered:

```
>>> f = (12*x + 12)*x
>>> g = 16*x**2
>>> gcd(f, g)
4*x
```

But if the polynomials have rational coefficients, then the returned polynomial is monic:

```
>>> f = 3*x**2/2
>>> g = 9*x/4
>>> gcd(f, g)
x
```

It also works with multiple variables. In this case, the variables are ordered alphabetically, be default, which has influence on the leading coefficient:

```
>>> f = x*y/2 + y**2
>>> g = 3*x + 6*y

>>> gcd(f, g)
x + 2*y
```

The lcm is connected with the gcd and one can be computed using the other:

```
>>> f = x*y**2 + x**2*y
>>> g = x**2*y**2
>>> gcd(f, g)
x*y
>>> lcm(f, g)
 3  2    2  3
x *y  + x *y
>>> (f*g).expand()
 4  3    3  4
x *y  + x *y
>>> (gcd(f, g, x, y)*lcm(f, g, x, y)).expand()
 4  3    3  4
x *y  + x *y
```

Square-free factorization

The square-free factorization of a univariate polynomial is the product of all factors (not necessarily irreducible) of degree 1, 2 etc.:

```
>>> f = 2*x**2 + 5*x**3 + 4*x**4 + x**5

>>> sqf_list(f)
                    2
(1, [(x + 2, 1), (x  + x, 2)])

>>> sqf(f)
                2
       / 2    \
(x + 2)*\x  + x/
```

Factorization

This function provides factorization of univariate and multivariate polynomials with rational coefficients:

```
>>> factor(x**4/2 + 5*x**3/12 - x**2/3)
 2
x *(2*x - 1)*(3*x + 4)
----------------------
          12

>>> factor(x**2 + 4*x*y + 4*y**2)
```

(continues on next page)

(continued from previous page)

```
         2
(x + 2*y)
```

Groebner bases

Buchberger's algorithm is implemented, supporting various monomial orders:

```
>>> groebner([x**2 + 1, y**4*x + x**3], x, y, order='lex')
             /[ 2       4    ]                              \
GroebnerBasis\[x  + 1, y  - 1], x, y, domain=ZZ, order=lex/


>>> groebner([x**2 + 1, y**4*x + x**3, x*y*z**3], x, y, z, order='grevlex')
             /[ 4       3   2    ]                                \
GroebnerBasis\[y  - 1, z , x  + 1], x, y, z, domain=ZZ, order=grevlex/
```

Solving Equations

We have (incomplete) methods to find the complex or even symbolic roots of polynomials and to solve some systems of polynomial equations:

```
>>> from sympy import roots, solve_poly_system

>>> solve(x**3 + 2*x + 3, x)

     1   \/ 11 *I  1   \/ 11 *I
[-1, - - --------, - + --------]
     2      2      2      2

>>> p = Symbol('p')
>>> q = Symbol('q')

>>> solve(x**2 + p*x + q, x)

         _________             _________
        /  2                  /  2
   p   \/  p  - 4*q      p   \/  p  - 4*q
[- - - -------------, - - + -------------]
   2         2           2         2

>>> solve_poly_system([y - x, x - 5], x, y)
[(5, 5)]

>>> solve_poly_system([y**2 - x**3 + 1, y*x], x, y)

                            1   \/ 3 *I         1   \/ 3 *I
[(0, -I), (0, I), (1, 0), (- - - -------, 0), (- - + -------, 0)]
                            2      2            2      2
```

Examples from Wester's Article

Introduction

In this tutorial we present examples from Wester's article concerning comparison and critique of mathematical abilities of several computer algebra systems (see [Wester1999]). All the examples are related to polynomial and algebraic computations and SymPy specific remarks were added to all of them.

Examples

All examples in this tutorial are computable, so one can just copy and paste them into a Python shell and do something useful with them. All computations were done using the following setup:

```
>>> from sympy import *

>>> init_printing(use_unicode=True, wrap_line=False)

>>> var('x,y,z,s,c,n')
(x, y, z, s, c, n)
```

Simple univariate polynomial factorization

To obtain a factorization of a polynomial use *factor()* (page 2373) function. By default *factor()* (page 2373) returns the result in unevaluated form, so the content of the input polynomial is left unexpanded, as in the following example:

```
>>> factor(6*x - 10)
2·(3·x - 5)
```

To achieve the same effect in a more systematic way use *primitive()* (page 2369) function, which returns the content and the primitive part of the input polynomial:

```
>>> primitive(6*x - 10)
(2, 3·x - 5)
```

Note: The content and the primitive part can be computed only over a ring. To simplify coefficients of a polynomial over a field use *monic()* (page 2369).

Univariate GCD, resultant and factorization

Consider univariate polynomials `f`, `g` and `h` over integers:

```
>>> f = 64*x**34 - 21*x**47 - 126*x**8 - 46*x**5 - 16*x**60 - 81
>>> g = 72*x**60 - 25*x**25 - 19*x**23 - 22*x**39 - 83*x**52 + 54*x**10 + 81
>>> h = 34*x**19 - 25*x**16 + 70*x**7 + 20*x**3 - 91*x - 86
```

We can compute the greatest common divisor (GCD) of two polynomials using *gcd()* (page 2368) function:

```
>>> gcd(f, g)
1
```

We see that `f` and `g` have no common factors. However, `f*h` and `g*h` have an obvious factor `h`:

```
>>> gcd(expand(f*h), expand(g*h)) - h
0
```

The same can be verified using the resultant of univariate polynomials:

```
>>> resultant(expand(f*h), expand(g*h))
0
```

Factorization of large univariate polynomials (of degree 120 in this case) over integers is also possible:

```
>>> factor(expand(f*g))
 ⎛    60       47       34       8       5     ⎞ ⎛    60       52     39     ␣
 ↪  25       23       10     ⎞
-⎝16⋅x   + 21⋅x   - 64⋅x   + 126⋅x  + 46⋅x  + 81⎠⋅⎝72⋅x   - 83⋅x - 22⋅x   -␣
 ↪25⋅x   - 19⋅x   + 54⋅x   + 81⎠
```

Multivariate GCD and factorization

What can be done in univariate case, can be also done for multivariate polynomials. Consider the following polynomials `f`, `g` and `h` in $\mathbb{Z}[x, y, z]$:

```
>>> f = 24*x*y**19*z**8 - 47*x**17*y**5*z**8 + 6*x**15*y**9*z**2 - 3*x**22 + 5
>>> g = 34*x**5*y**8*z**13 + 20*x**7*y**7*z**7 + 12*x**9*y**16*z**4 +␣
 ↪80*y**14*z
>>> h = 11*x**12*y**7*z**13 - 23*x**2*y**8*z**10 + 47*x**17*y**5*z**8
```

As previously, we can verify that `f` and `g` have no common factors:

```
>>> gcd(f, g)
1
```

However, `f*h` and `g*h` have an obvious factor `h`:

```
>>> gcd(expand(f*h), expand(g*h)) - h
0
```

Multivariate factorization of large polynomials is also possible:

```
>>> factor(expand(f*g))
    7   ⎛   9  9  3       7  6       5    12       7⎞ ⎛   22       17  5  8  ␣
↪    15  9  2         19  8    ⎞
-2⋅y ⋅z⋅⎝6⋅x ⋅y ⋅z  + 10⋅x ⋅z  + 17⋅x ⋅y⋅z   + 40⋅y ⎠⋅⎝3⋅x   + 47⋅x  ⋅y ⋅z  -␣
↪6⋅x  ⋅y ⋅z  - 24⋅x⋅y  ⋅z  - 5⎠
```

Support for symbols in exponents

Polynomial manipulation functions provided by *sympy.polys* (page 2360) are mostly used with integer exponents. However, it's perfectly valid to compute with symbolic exponents, e.g.:

```
>>> gcd(2*x**(n + 4) - x**(n + 2), 4*x**(n + 1) + 3*x**n)
 n
x
```

Testing if polynomials have common zeros

To test if two polynomials have a root in common we can use *resultant()* (page 2366) function. The theory says that the resultant of two polynomials vanishes if there is a common zero of those polynomials. For example:

```
>>> resultant(3*x**4 + 3*x**3 + x**2 - x - 2, x**3 - 3*x**2 + x + 5)
0
```

We can visualize this fact by factoring the polynomials:

```
>>> factor(3*x**4 + 3*x**3 + x**2 - x - 2)
       ⎛   3       ⎞
(x + 1)⋅⎝3⋅x  + x - 2⎠

>>> factor(x**3 - 3*x**2 + x + 5)
       ⎛ 2          ⎞
(x + 1)⋅⎝x  - 4⋅x + 5⎠
```

In both cases we obtained the factor $x + 1$ which tells us that the common root is $x = -1$.

Normalizing simple rational functions

To remove common factors from the numerator and the denominator of a rational function the elegant way, use *cancel()* (page 2376) function. For example:

```
>>> cancel((x**2 - 4)/(x**2 + 4*x + 4))
x - 2
─────
x + 2
```

Expanding expressions and factoring back

One can work easily we expressions in both expanded and factored forms. Consider a polynomial `f` in expanded form. We differentiate it and factor the result back:

```
>>> f = expand((x + 1)**20)

>>> g = diff(f, x)

>>> factor(g)
          19
20·(x + 1)
```

The same can be achieved in factored form:

```
>>> diff((x + 1)**20, x)
          19
20·(x + 1)
```

Factoring in terms of cyclotomic polynomials

SymPy can very efficiently decompose polynomials of the form $x^n \pm 1$ in terms of cyclotomic polynomials:

```
>>> factor(x**15 - 1)
        ⎛ 2          ⎞ ⎛ 4    3    2        ⎞ ⎛ 8    7    5    4    3        ⎞
(x - 1)·⎝x  + x + 1⎠·⎝x  + x  + x  + x + 1⎠·⎝x  - x  + x  - x  + x - x + 1⎠
```

The original Wester`s example was $x^{100} - 1$, but was truncated for readability purpose. Note that this is not a big struggle for *factor()* (page 2373) to decompose polynomials of degree 1000 or greater.

Univariate factoring over Gaussian numbers

Consider a univariate polynomial `f` with integer coefficients:

```
>>> f = 4*x**4 + 8*x**3 + 77*x**2 + 18*x + 153
```

We want to obtain a factorization of `f` over Gaussian numbers. To do this we use *factor()* (page 2373) as previously, but this time we set `gaussian` keyword to `True`:

```
>>> factor(f, gaussian=True)
  ⎛    3·i⎞ ⎛    3·i⎞
4·⎜x - ────⎟·⎜x + ────⎟·(x + 1 - 4·i)·(x + 1 + 4·i)
  ⎝     2 ⎠ ⎝     2 ⎠
```

As the result we got a splitting factorization of `f` with monic factors (this is a general rule when computing in a field with SymPy). The `gaussian` keyword is useful for improving code readability, however the same result can be computed using more general syntax:

```
>>> factor(f, extension=I)
  ⎛    3⋅i⎞ ⎛    3⋅i⎞
4⋅⎜x - ────⎟⋅⎜x + ────⎟⋅(x + 1 - 4⋅i)⋅(x + 1 + 4⋅i)
  ⎝     2 ⎠ ⎝     2 ⎠
```

Computing with automatic field extensions

Consider two univariate polynomials `f` and `g`:

```
>>> f = x**3 + (sqrt(2) - 2)*x**2 - (2*sqrt(2) + 3)*x - 3*sqrt(2)
>>> g = x**2 - 2
```

We would like to reduce degrees of the numerator and the denominator of a rational function `f/g`. To do this we employ *cancel()* (page 2376) function:

```
>>> cancel(f/g)
 3      2       2
x  - 2⋅x  + √2⋅x  - 3⋅x - 2⋅√2⋅x - 3⋅√2
──────────────────────────────────────
                 2
                x  - 2
```

Unfortunately nothing interesting happened. This is because by default SymPy treats $\sqrt{2}$ as a generator, obtaining a bivariate polynomial for the numerator. To make *cancel()* (page 2376) recognize algebraic properties of $\sqrt{2}$, one needs to use `extension` keyword:

```
>>> cancel(f/g, extension=True)
 2
x  - 2⋅x - 3
────────────
   x - √2
```

Setting `extension=True` tells *cancel()* (page 2376) to find minimal algebraic number domain for the coefficients of `f/g`. The automatically inferred domain is $\mathbb{Q}(\sqrt{2})$. If one doesn't want to rely on automatic inference, the same result can be obtained by setting the `extension` keyword with an explicit algebraic number:

```
>>> cancel(f/g, extension=sqrt(2))
 2
x  - 2⋅x - 3
────────────
   x - √2
```