```
>>> e1 = sqrt(x**2 + y**2) - 10
>>> e2 = sqrt(y**2 + (-x + 10)**2) - 3
>>> nonlinsolve((e1, e2), (x, y))
{(191/20, -3*sqrt(391)/20), (191/20, 3*sqrt(391)/20)}
>>> nonlinsolve([x**2 + 2/y - 2, x + y - 3], [x, y])
{(1, 2), (1 - sqrt(5), 2 + sqrt(5)), (1 + sqrt(5), 2 - sqrt(5))}
>>> nonlinsolve([x**2 + 2/y - 2, x + y - 3], [y, x])
{(2, 1), (2 - sqrt(5), 1 + sqrt(5)), (2 + sqrt(5), 1 - sqrt(5))}
```

6. It is better to use symbols instead of trigonometric functions or *Function* (page 1050). For example, replace $\sin(x)$ with a symbol, replace $f(x)$ with a symbol and so on. Get a solution from `nonlinsolve` and then use *solveset()* (page 866) to get the value of x.

How Nonlinsolve Is Better Than Old Solver `_solve_system` :

1. A positive dimensional system solver: nonlinsolve can return solution for positive dimensional system. It finds the Groebner Basis of the positive dimensional system(calling it as basis) then we can start solving equation(having least number of variable first in the basis) using solveset and substituting that solved solutions into other equation(of basis) to get solution in terms of minimum variables. Here the important thing is how we are substituting the known values and in which equations.

2. Real and complex solutions: nonlinsolve returns both real and complex solution. If all the equations in the system are polynomial then using *solve_poly_system()* (page 856) both real and complex solution is returned. If all the equations in the system are not polynomial equation then goes to `substitution` method with this polynomial and non polynomial equation(s), to solve for unsolved variables. Here to solve for particular variable solveset_real and solveset_complex is used. For both real and complex solution `_solve_using_known_values` is used inside `substitution` (`substitution` will be called when any non-polynomial equation is present). If a solution is valid its general solution is added to the final result.

3. *Complement* (page 1200) and *Intersection* (page 1198) will be added: nonlinsolve maintains dict for complements and intersections. If solveset find complements or/and intersections with any interval or set during the execution of `substitution` function, then complement or/and intersection for that variable is added before returning final solution.

transolve

`sympy.solvers.solveset._transolve`(*f*, *symbol*, *domain*)

Function to solve transcendental equations. It is a helper to `solveset` and should be used internally. `_transolve` currently supports the following class of equations:

- Exponential equations
- Logarithmic equations

Parameters

f : Any transcendental equation that needs to be solved.

This needs to be an expression, which is assumed to be equal to `0`.

symbol : The variable for which the equation is solved.

This needs to be of class `Symbol`.

domain : A set over which the equation is solved.

This needs to be of class `Set`.

Returns

Set

A set of values for `symbol` for which `f` is equal to zero. An `EmptySet` is returned if `f` does not have solutions in respective domain. A `ConditionSet` is returned as unsolved object if algorithms to evaluate complete solution are not yet implemented.

How To Use `_transolve`

`_transolve` should not be used as an independent function, because it assumes that the equation (`f`) and the `symbol` comes from `solveset` and might have undergone a few modification(s). To use `_transolve` as an independent function the equation (`f`) and the `symbol` should be passed as they would have been by `solveset`.

Examples

```
>>> from sympy.solvers.solveset import _transolve as transolve
>>> from sympy.solvers.solvers import _tsolve as tsolve
>>> from sympy import symbols, S, pprint
>>> x = symbols('x', real=True) # assumption added
>>> transolve(5**(x - 3) - 3**(2*x + 1), x, S.Reals)
{-(log(3) + 3*log(5))/(-log(5) + 2*log(3))}
```

How `_transolve` Works

`_transolve` uses two types of helper functions to solve equations of a particular class:

Identifying helpers: To determine whether a given equation belongs to a certain class of equation or not. Returns either `True` or `False`.

Solving helpers: Once an equation is identified, a corresponding helper either solves the equation or returns a form of the equation that `solveset` might better be able to handle.

- Philosophy behind the module

The purpose of `_transolve` is to take equations which are not already polynomial in their generator(s) and to either recast them as such through a valid transformation or to solve them outright. A pair of helper functions for each class of supported transcendental functions are employed for this purpose. One identifies the transcendental form of an equation and the other either solves it or recasts it into a tractable form that can be solved by `solveset`. For example, an equation in the form $ab^{f(x)} - cd^{g(x)} = 0$ can be transformed to $\log(a) + f(x)\log(b) - \log(c) - g(x)\log(d) = 0$ (under certain assumptions) and this can be solved with `solveset` if $f(x)$ and $g(x)$ are in polynomial form.

How _transolve Is Better Than _tsolve

1) Better output

_transolve provides expressions in a more simplified form.

Consider a simple exponential equation

```
>>> f = 3**(2*x) - 2**(x + 3)
>>> pprint(transolve(f, x, S.Reals), use_unicode=False)
    -3*log(2)
{------------------}
 -2*log(3) + log(2)
>>> pprint(tsolve(f, x), use_unicode=False)
     /   3     \
     | --------|
     | log(2/9)|
[-log\2         /]
```

2) Extensible

The API of _transolve is designed such that it is easily extensible, i.e. the code that solves a given class of equations is encapsulated in a helper and not mixed in with the code of _transolve itself.

3) Modular

_transolve is designed to be modular i.e, for every class of equation a separate helper for identification and solving is implemented. This makes it easy to change or modify any of the method implemented directly in the helpers without interfering with the actual structure of the API.

4) Faster Computation

Solving equation via _transolve is much faster as compared to _tsolve. In solve, attempts are made computing every possibility to get the solutions. This series of attempts makes solving a bit slow. In _transolve, computation begins only after a particular type of equation is identified.

How To Add New Class Of Equations

Adding a new class of equation solver is a three-step procedure:

- Identify the type of the equations

  Determine the type of the class of equations to which they belong: it could be of Add, Pow, etc. types. Separate internal functions are used for each type. Write identification and solving helpers and use them from within the routine for the given type of equation (after adding it, if necessary). Something like:

```
def add_type(lhs, rhs, x):
    ....
    if _is_exponential(lhs, x):
        new_eq = _solve_exponential(lhs, rhs, x)
....
```

(continues on next page)

(continued from previous page)

```
rhs, lhs = eq.as_independent(x)
if lhs.is_Add:
    result = add_type(lhs, rhs, x)
```

- Define the identification helper.
- Define the solving helper.

Apart from this, a few other things needs to be taken care while adding an equation solver:

- Naming conventions: Name of the identification helper should be as `_is_class` where class will be the name or abbreviation of the class of equation. The solving helper will be named as `_solve_class`. For example: for exponential equations it becomes `_is_exponential` and `_solve_expo`.
- The identifying helpers should take two input parameters, the equation to be checked and the variable for which a solution is being sought, while solving helpers would require an additional domain parameter.
- Be sure to consider corner cases.
- Add tests for each helper.
- Add a docstring to your helper that describes the method implemented. The documentation of the helpers should identify:
  - the purpose of the helper,
  - the method used to identify and solve the equation,
  - a proof of correctness
  - the return values of the helpers

`sympy.solvers.solveset._is_exponential`(*f*, *symbol*)

Return `True` if one or more terms contain `symbol` only in exponents, else `False`.

Parameters

f : Expr

The equation to be checked

symbol : Symbol

The variable in which the equation is checked

Examples

```
>>> from sympy import symbols, cos, exp
>>> from sympy.solvers.solveset import _is_exponential as check
>>> x, y = symbols('x y')
>>> check(y, y)
False
>>> check(x**y - 1, y)
True
>>> check(x**y*2**y - 1, y)
True
```

(continues on next page)

(continued from previous page)

```
>>> check(exp(x + 3) + 3**x, x)
True
>>> check(cos(2**x), x)
False
```

- Philosophy behind the helper

The function extracts each term of the equation and checks if it is of exponential form w.r.t `symbol`.

`sympy.solvers.solveset._solve_exponential`(*lhs*, *rhs*, *symbol*, *domain*)

Helper function for solving (supported) exponential equations.

Exponential equations are the sum of (currently) at most two terms with one or both of them having a power with a symbol-dependent exponent.

For example

$$5^{2x+3} - 5^{3x-1}$$

$$4^{5-9x} - e^{2-x}$$

Parameters

lhs, rhs : Expr

The exponential equation to be solved, $lhs = rhs$

symbol : Symbol

The variable in which the equation is solved

domain : Set

A set over which the equation is solved.

Returns

A set of solutions satisfying the given equation.

A `ConditionSet` if the equation is unsolvable or

if the assumptions are not properly defined, in that case

a different style of `ConditionSet` is returned having the

solution(s) of the equation with the desired assumptions.

Examples

```
>>> from sympy.solvers.solveset import _solve_exponential as solve_expo
>>> from sympy import symbols, S
>>> x = symbols('x', real=True)
>>> a, b = symbols('a b')
>>> solve_expo(2**x + 3**x - 5**x, 0, x, S.Reals)  # not solvable
ConditionSet(x, Eq(2**x + 3**x - 5**x, 0), Reals)
>>> solve_expo(a**x - b**x, 0, x, S.Reals)  # solvable but incorrect␣
↪assumptions
ConditionSet(x, (a > 0) & (b > 0), {0})
```

(continues on next page)

(continued from previous page)

```
>>> solve_expo(3**(2*x) - 2**(x + 3), 0, x, S.Reals)
{-3*log(2)/(-2*log(3) + log(2))}
>>> solve_expo(2**x - 4**x, 0, x, S.Reals)
{0}
```

- Proof of correctness of the method

The logarithm function is the inverse of the exponential function. The defining relation between exponentiation and logarithm is:

$$\log_b x = y \ \ if \ \ b^y = x$$

Therefore if we are given an equation with exponent terms, we can convert every term to its corresponding logarithmic form. This is achieved by taking logarithms and expanding the equation using logarithmic identities so that it can easily be handled by `solveset`.

For example:

$$3^{2x} = 2^{x+3}$$

Taking log both sides will reduce the equation to

$$(2x)\log(3) = (x+3)\log(2)$$

This form can be easily handed by `solveset`.

`sympy.solvers.solveset._solve_logarithm`(*lhs*, *rhs*, *symbol*, *domain*)

Helper to solve logarithmic equations which are reducible to a single instance of log.

Logarithmic equations are (currently) the equations that contains log terms which can be reduced to a single log term or a constant using various logarithmic identities.

For example:

$$\log(x) + \log(x-4)$$

can be reduced to:

$$\log(x(x-4))$$

Parameters

lhs, rhs : Expr

The logarithmic equation to be solved, $lhs = rhs$

symbol : Symbol

The variable in which the equation is solved

domain : Set

A set over which the equation is solved.

Returns

A set of solutions satisfying the given equation.

A `ConditionSet` if the equation is unsolvable.

Examples

```
>>> from sympy import symbols, log, S
>>> from sympy.solvers.solveset import _solve_logarithm as solve_log
>>> x = symbols('x')
>>> f = log(x - 3) + log(x + 3)
>>> solve_log(f, 0, x, S.Reals)
{-sqrt(10), sqrt(10)}
```

- Proof of correctness

A logarithm is another way to write exponent and is defined by

$$\log_b x = y \ \ if \ \ b^y = x$$

When one side of the equation contains a single logarithm, the equation can be solved by rewriting the equation as an equivalent exponential equation as defined above. But if one side contains more than one logarithm, we need to use the properties of logarithm to condense it into a single logarithm.

Take for example

$$\log(2x) - 15 = 0$$

contains single logarithm, therefore we can directly rewrite it to exponential form as

$$x = \frac{e^{15}}{2}$$

But if the equation has more than one logarithm as

$$\log(x - 3) + \log(x + 3) = 0$$

we use logarithmic identities to convert it into a reduced form

Using,

$$\log(a) + \log(b) = \log(ab)$$

the equation becomes,

$$\log((x - 3)(x + 3))$$

This equation contains one logarithm and can be solved by rewriting to exponents.

`sympy.solvers.solveset._is_logarithmic`(*f*, *symbol*)

Return `True` if the equation is in the form $a\log(f(x)) + b\log(g(x)) + ... + c$ else `False`.

Parameters

f : Expr

The equation to be checked

symbol : Symbol

The variable in which the equation is checked

Returns

`True` if the equation is logarithmic otherwise `False`.

Examples

```
>>> from sympy import symbols, tan, log
>>> from sympy.solvers.solveset import _is_logarithmic as check
>>> x, y = symbols('x y')
>>> check(log(x + 2) - log(x + 3), x)
True
>>> check(tan(log(2*x)), x)
False
>>> check(x*log(x), x)
False
>>> check(x + log(x), x)
False
>>> check(y + log(x), x)
True
```

- Philosophy behind the helper

The function extracts each term and checks whether it is logarithmic w.r.t `symbol`.

Diophantine Equations (DEs)

See *Diophantine* (page 714)

Inequalities

See *Inequality Solvers* (page 750)

Ordinary Differential equations (ODEs)

See *ODE* (page 755).

Partial Differential Equations (PDEs)

See *PDE* (page 826).

abc

This module exports all latin and greek letters as Symbols, so you can conveniently do

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

instead of the slightly more clunky-looking

```
>>> from sympy import symbols
>>> x, y = symbols('x y')
```

Caveats

1. As of the time of writing this, the names `O`, `S`, `I`, `N`, `E`, and `Q` are colliding with names defined in SymPy. If you import them from both `sympy.abc` and `sympy`, the second import will "win". This is an issue only for * imports, which should only be used for short-lived code such as interactive sessions and throwaway scripts that do not survive until the next SymPy upgrade, where `sympy` may contain a different set of names.

2. This module does not define symbol names on demand, i.e. `from sympy.abc import foo` will be reported as an error because `sympy.abc` does not contain the name `foo`. To get a symbol named `foo`, you still need to use `Symbol('foo')` or `symbols('foo')`. You can freely mix usage of `sympy.abc` and `Symbol`/`symbols`, though sticking with one and only one way to get the symbols does tend to make the code more readable.

The module also defines some special names to help detect which names clash with the default SymPy namespace.

`_clash1` defines all the single letter variables that clash with SymPy objects; `_clash2` defines the multi-letter clashing symbols; and `_clash` is the union of both. These can be passed for `locals` during sympification if one desires Symbols rather than the non-Symbol objects for those names.

Examples

```
>>> from sympy import S
>>> from sympy.abc import _clash1, _clash2, _clash
>>> S("Q & C", locals=_clash1)
C & Q
>>> S('pi(x)', locals=_clash2)
pi(x)
>>> S('pi(C, Q)', locals=_clash)
pi(C, Q)
```


Algebras

Introduction

The Algebras module for SymPy provides support for basic algebraic operations on Quaternions.

Quaternion Reference

This section lists the classes implemented by the Algebras module.

class sympy.algebras.**Quaternion**(*a=0*, *b=0*, *c=0*, *d=0*, *real_field=True*)

Provides basic quaternion operations. Quaternion objects can be instantiated as Quaternion(a, b, c, d) as in (a + b*i + c*j + d*k).

Examples

```
>>> from sympy import Quaternion
>>> q = Quaternion(1, 2, 3, 4)
>>> q
1 + 2*i + 3*j + 4*k
```

Quaternions over complex fields can be defined as :

```
>>> from sympy import Quaternion
>>> from sympy import symbols, I
>>> x = symbols('x')
>>> q1 = Quaternion(x, x**3, x, x**2, real_field = False)
>>> q2 = Quaternion(3 + 4*I, 2 + 5*I, 0, 7 + 8*I, real_field = False)
>>> q1
x + x**3*i + x*j + x**2*k
>>> q2
(3 + 4*I) + (2 + 5*I)*i + 0*j + (7 + 8*I)*k
```

References

[R1], [R2]

add(*other*)

Adds quaternions.

Parameters

other : Quaternion

The quaternion to add to current (self) quaternion.

Returns

Quaternion

The resultant quaternion after adding self to other

Examples

```
>>> from sympy import Quaternion
>>> from sympy import symbols
>>> q1 = Quaternion(1, 2, 3, 4)
>>> q2 = Quaternion(5, 6, 7, 8)
>>> q1.add(q2)
6 + 8*i + 10*j + 12*k
>>> q1 + 5
6 + 2*i + 3*j + 4*k
>>> x = symbols('x', real = True)
>>> q1.add(x)
(x + 1) + 2*i + 3*j + 4*k
```

Quaternions over complex fields :

```
>>> from sympy import Quaternion
>>> from sympy import I
>>> q3 = Quaternion(3 + 4*I, 2 + 5*I, 0, 7 + 8*I, real_field = False)
>>> q3.add(2 + 3*I)
(5 + 7*I) + (2 + 5*I)*i + 0*j + (7 + 8*I)*k
```

angle()

Returns the angle of the quaternion measured in the real-axis plane.

Explanation

Given a quaternion $q = a + bi + cj + dk$ where a, b, c and d are real numbers, returns the angle of the quaternion given by

$$angle := atan2(\sqrt{b^2 + c^2 + d^2}, a)$$

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(1, 4, 4, 4)
>>> q.angle()
atan(4*sqrt(3))
```

arc_coplanar(*other*)

Returns True if the transformation arcs represented by the input quaternions happen in the same plane.

Parameters

other : a Quaternion

Returns

True : if the planes of the two quaternions are the same, apart from its orientation/sign.

False : if the planes of the two quaternions are not the same, apart from its orientation/sign.

None : if plane of either of the quaternion is unknown.

Explanation

Two quaternions are said to be coplanar (in this arc sense) when their axes are parallel. The plane of a quaternion is the one normal to its axis.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q1 = Quaternion(1, 4, 4, 4)
>>> q2 = Quaternion(3, 8, 8, 8)
>>> Quaternion.arc_coplanar(q1, q2)
True
```

```
>>> q1 = Quaternion(2, 8, 13, 12)
>>> Quaternion.arc_coplanar(q1, q2)
False
```

See also:

vector_coplanar (page 897), *is_pure* (page 891)

axis()

Returns the axis($\mathbf{Ax}(q)$) of the quaternion.

Explanation

Given a quaternion $q = a + bi + cj + dk$, returns $\mathbf{Ax}(q)$ i.e., the versor of the vector part of that quaternion equal to $\mathbf{U}[\mathbf{V}(q)]$. The axis is always an imaginary unit with square equal to $-1 + 0i + 0j + 0k$.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(1, 1, 1, 1)
>>> q.axis()
0 + sqrt(3)/3*i + sqrt(3)/3*j + sqrt(3)/3*k
```

See also:

vector_part (page 898)

exp()

Returns the exponential of q (e^q).

Returns

Quaternion

Exponential of q (e^q).

Examples

```
>>> from sympy import Quaternion
>>> q = Quaternion(1, 2, 3, 4)
>>> q.exp()
E*cos(sqrt(29))
+ 2*sqrt(29)*E*sin(sqrt(29))/29*i
+ 3*sqrt(29)*E*sin(sqrt(29))/29*j
+ 4*sqrt(29)*E*sin(sqrt(29))/29*k
```

classmethod from_axis_angle(*vector*, *angle*)

Returns a rotation quaternion given the axis and the angle of rotation.

Parameters

vector : tuple of three numbers

The vector representation of the given axis.

angle : number

The angle by which axis is rotated (in radians).

Returns

Quaternion

The normalized rotation quaternion calculated from the given axis and the angle of rotation.

Examples

```
>>> from sympy import Quaternion
>>> from sympy import pi, sqrt
>>> q = Quaternion.from_axis_angle((sqrt(3)/3, sqrt(3)/3, sqrt(3)/3),␣
↪2*pi/3)
>>> q
1/2 + 1/2*i + 1/2*j + 1/2*k
```

classmethod from_rotation_matrix(*M*)

Returns the equivalent quaternion of a matrix. The quaternion will be normalized only if the matrix is special orthogonal (orthogonal and det(M) = 1).

Parameters

M : Matrix

Input matrix to be converted to equivalent quaternion. M must be special orthogonal (orthogonal and det(M) = 1) for the quaternion to be normalized.

Returns

Quaternion

The quaternion equivalent to given matrix.

Examples

```
>>> from sympy import Quaternion
>>> from sympy import Matrix, symbols, cos, sin, trigsimp
>>> x = symbols('x')
>>> M = Matrix([[cos(x), -sin(x), 0], [sin(x), cos(x), 0], [0, 0, 1]])
>>> q = trigsimp(Quaternion.from_rotation_matrix(M))
>>> q
sqrt(2)*sqrt(cos(x) + 1)/2 + 0*i + 0*j + sqrt(2 - ␣
↪2*cos(x))*sign(sin(x))/2*k
```

index_vector()

Returns the index vector of the quaternion.

Returns

Quaternion: representing index vector of the provided quaternion.

Explanation

Index vector is given by $\mathbf{T}(q)$ multiplied by $\mathbf{Ax}(q)$ where $\mathbf{Ax}(q)$ is the axis of the quaternion q, and mod(q) is the $\mathbf{T}(q)$ (magnitude) of the quaternion.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(2, 4, 2, 4)
>>> q.index_vector()
0 + 4*sqrt(10)/3*i + 2*sqrt(10)/3*j + 4*sqrt(10)/3*k
```

See also:

axis (page 888), *norm* (page 892)

integrate(**args*)

Computes integration of quaternion.

Returns

Quaternion

Integration of the quaternion(self) with the given variable.

Examples

Indefinite Integral of quaternion :

```
>>> from sympy import Quaternion
>>> from sympy.abc import x
>>> q = Quaternion(1, 2, 3, 4)
>>> q.integrate(x)
x + 2*x*i + 3*x*j + 4*x*k
```

Definite integral of quaternion :

```
>>> from sympy import Quaternion
>>> from sympy.abc import x
>>> q = Quaternion(1, 2, 3, 4)
>>> q.integrate((x, 1, 5))
4 + 8*i + 12*j + 16*k
```

inverse()

Returns the inverse of the quaternion.

is_pure()

Returns true if the quaternion is pure, false if the quaternion is not pure or returns none if it is unknown.

Explanation

A pure quaternion (also a vector quaternion) is a quaternion with scalar part equal to 0.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(0, 8, 13, 12)
>>> q.is_pure()
True
```

See also:

scalar_part (page 895)

is_zero_quaternion()

Returns true if the quaternion is a zero quaternion or false if it is not a zero quaternion and None if the value is unknown.

Explanation

A zero quaternion is a quaternion with both scalar part and vector part equal to 0.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(1, 0, 0, 0)
>>> q.is_zero_quaternion()
False
```

```
>>> q = Quaternion(0, 0, 0, 0)
>>> q.is_zero_quaternion()
True
```

See also:

scalar_part (page 895), *vector_part* (page 898)

mensor()

Returns the natural logarithm of the norm(magnitude) of the quaternion.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(2, 4, 2, 4)
>>> q.mensor()
log(2*sqrt(10))
>>> q.norm()
2*sqrt(10)
```

See also:

norm (page 892)

mul(*other*)

Multiplies quaternions.

Parameters

other : Quaternion or symbol

The quaternion to multiply to current (self) quaternion.

Returns

Quaternion

The resultant quaternion after multiplying self with other

Examples

```
>>> from sympy import Quaternion
>>> from sympy import symbols
>>> q1 = Quaternion(1, 2, 3, 4)
>>> q2 = Quaternion(5, 6, 7, 8)
>>> q1.mul(q2)
(-60) + 12*i + 30*j + 24*k
>>> q1.mul(2)
2 + 4*i + 6*j + 8*k
>>> x = symbols('x', real = True)
>>> q1.mul(x)
x + 2*x*i + 3*x*j + 4*x*k
```

Quaternions over complex fields :

```
>>> from sympy import Quaternion
>>> from sympy import I
>>> q3 = Quaternion(3 + 4*I, 2 + 5*I, 0, 7 + 8*I, real_field = False)
>>> q3.mul(2 + 3*I)
(2 + 3*I)*(3 + 4*I) + (2 + 3*I)*(2 + 5*I)*i + 0*j + (2 + 3*I)*(7 +␣
↪8*I)*k
```

norm()

Returns the norm of the quaternion.

normalize()

Returns the normalized form of the quaternion.

orthogonal(*other*)

Returns the orthogonality of two quaternions.

Parameters

other : a Quaternion

Returns

True : if the two pure quaternions seen as 3D vectors are orthogonal.

False : if the two pure quaternions seen as 3D vectors are not orthogonal.

None : if the two pure quaternions seen as 3D vectors are orthogonal is unknown.

Explanation

Two pure quaternions are called orthogonal when their product is anti-commutative.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(0, 4, 4, 4)
>>> q1 = Quaternion(0, 8, 8, 8)
>>> q.orthogonal(q1)
False
```

```
>>> q1 = Quaternion(0, 2, 2, 0)
>>> q = Quaternion(0, 2, -2, 0)
>>> q.orthogonal(q1)
True
```

parallel(*other*)

Returns True if the two pure quaternions seen as 3D vectors are parallel.

Parameters

other : a Quaternion

Returns

True : if the two pure quaternions seen as 3D vectors are parallel.

False : if the two pure quaternions seen as 3D vectors are not parallel.

None : if the two pure quaternions seen as 3D vectors are parallel is unknown.

Explanation

Two pure quaternions are called parallel when their vector product is commutative which implies that the quaternions seen as 3D vectors have same direction.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(0, 4, 4, 4)
>>> q1 = Quaternion(0, 8, 8, 8)
>>> q.parallel(q1)
True
```

```
>>> q1 = Quaternion(0, 8, 13, 12)
>>> q.parallel(q1)
False
```

pow(*p*)

Finds the pth power of the quaternion.

Parameters

p : int

Power to be applied on quaternion.

Returns

Quaternion

Returns the p-th power of the current quaternion. Returns the inverse if p = -1.

Examples

```
>>> from sympy import Quaternion
>>> q = Quaternion(1, 2, 3, 4)
>>> q.pow(4)
668 + (-224)*i + (-336)*j + (-448)*k
```

pow_cos_sin(*p*)

Computes the pth power in the cos-sin form.

Parameters

p : int

Power to be applied on quaternion.

Returns

Quaternion

The p-th power in the cos-sin form.

Examples

```
>>> from sympy import Quaternion
>>> q = Quaternion(1, 2, 3, 4)
>>> q.pow_cos_sin(4)
900*cos(4*acos(sqrt(30)/30))
+ 1800*sqrt(29)*sin(4*acos(sqrt(30)/30))/29*i
+ 2700*sqrt(29)*sin(4*acos(sqrt(30)/30))/29*j
+ 3600*sqrt(29)*sin(4*acos(sqrt(30)/30))/29*k
```

static rotate_point(*pin*, *r*)

Returns the coordinates of the point pin(a 3 tuple) after rotation.

Parameters

pin : tuple

A 3-element tuple of coordinates of a point which needs to be rotated.

r : Quaternion or tuple

Axis and angle of rotation.

It's important to note that when r is a tuple, it must be of the form (axis, angle)

Returns

tuple

The coordinates of the point after rotation.

Examples

```
>>> from sympy import Quaternion
>>> from sympy import symbols, trigsimp, cos, sin
>>> x = symbols('x')
>>> q = Quaternion(cos(x/2), 0, 0, sin(x/2))
>>> trigsimp(Quaternion.rotate_point((1, 1, 1), q))
(sqrt(2)*cos(x + pi/4), sqrt(2)*sin(x + pi/4), 1)
>>> (axis, angle) = q.to_axis_angle()
>>> trigsimp(Quaternion.rotate_point((1, 1, 1), (axis, angle)))
(sqrt(2)*cos(x + pi/4), sqrt(2)*sin(x + pi/4), 1)
```

scalar_part()

Returns scalar part($\mathbf{S}(q)$) of the quaternion q.

Explanation

Given a quaternion $q = a + bi + cj + dk$, returns $\mathbf{S}(q) = a$.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(4, 8, 13, 12)
>>> q.scalar_part()
4
```

to_axis_angle()

Returns the axis and angle of rotation of a quaternion

Returns

tuple

Tuple of (axis, angle)

Examples

```
>>> from sympy import Quaternion
>>> q = Quaternion(1, 1, 1, 1)
>>> (axis, angle) = q.to_axis_angle()
>>> axis
(sqrt(3)/3, sqrt(3)/3, sqrt(3)/3)
>>> angle
2*pi/3
```

to_rotation_matrix(*v=None*)

Returns the equivalent rotation transformation matrix of the quaternion which represents rotation about the origin if v is not passed.

Parameters

v : tuple or None

Default value: None

Returns

tuple

Returns the equivalent rotation transformation matrix of the quaternion which represents rotation about the origin if v is not passed.

Examples

```
>>> from sympy import Quaternion
>>> from sympy import symbols, trigsimp, cos, sin
>>> x = symbols('x')
>>> q = Quaternion(cos(x/2), 0, 0, sin(x/2))
>>> trigsimp(q.to_rotation_matrix((1, 1, 1)))
 Matrix([
[cos(x), -sin(x), 0,  sin(x) - cos(x) + 1],
[sin(x),  cos(x), 0, -sin(x) - cos(x) + 1],
[     0,       0, 1,                    0],
[     0,       0, 0,                    1]])
```

classmethod vector_coplanar(*q1*, *q2*, *q3*)

Returns True if the axis of the pure quaternions seen as 3D vectors q1, q2, and q3 are coplanar.

Parameters

q1 : a pure Quaternion.

q2 : a pure Quaternion.

q3 : a pure Quaternion.

Returns

True : if the axis of the pure quaternions seen as 3D vectors q1, q2, and q3 are coplanar.

False : if the axis of the pure quaternions seen as 3D vectors q1, q2, and q3 are not coplanar.

None : if the axis of the pure quaternions seen as 3D vectors q1, q2, and q3 are coplanar is unknown.

Explanation

Three pure quaternions are vector coplanar if the quaternions seen as 3D vectors are coplanar.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q1 = Quaternion(0, 4, 4, 4)
>>> q2 = Quaternion(0, 8, 8, 8)
>>> q3 = Quaternion(0, 24, 24, 24)
>>> Quaternion.vector_coplanar(q1, q2, q3)
True
```

```
>>> q1 = Quaternion(0, 8, 16, 8)
>>> q2 = Quaternion(0, 8, 3, 12)
>>> Quaternion.vector_coplanar(q1, q2, q3)
False
```

See also:

axis (page 888), *is_pure* (page 891)

vector_part()

Returns vector part($\mathbf{V}(q)$) of the quaternion q.

Explanation

Given a quaternion $q = a + bi + cj + dk$, returns $\mathbf{V}(q) = bi + cj + dk$.

Examples

```
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(1, 1, 1, 1)
>>> q.vector_part()
0 + 1*i + 1*j + 1*k
```

```
>>> q = Quaternion(4, 8, 13, 12)
>>> q.vector_part()
0 + 8*i + 13*j + 12*k
```

Concrete

Hypergeometric terms

The center stage, in recurrence solving and summations, play hypergeometric terms. Formally these are sequences annihilated by first order linear recurrence operators. In simple words if we are given term $a(n)$ then it is hypergeometric if its consecutive term ratio is a rational function in n.

To check if a sequence is of this type you can use the `is_hypergeometric` method which is available in Basic class. Here is simple example involving a polynomial:

```
>>> from sympy import *
>>> n, k = symbols('n,k')
>>> (n**2 + 1).is_hypergeometric(n)
True
```

Of course polynomials are hypergeometric but are there any more complicated sequences of this type? Here are some trivial examples:

```
>>> factorial(n).is_hypergeometric(n)
True
>>> binomial(n, k).is_hypergeometric(n)
True
>>> rf(n, k).is_hypergeometric(n)
True
>>> ff(n, k).is_hypergeometric(n)
True
```

(continues on next page)

(continued from previous page)

```
>>> gamma(n).is_hypergeometric(n)
True
>>> (2**n).is_hypergeometric(n)
True
```

We see that all species used in summations and other parts of concrete mathematics are hypergeometric. Note also that binomial coefficients and both rising and falling factorials are hypergeometric in both their arguments:

```
>>> binomial(n, k).is_hypergeometric(k)
True
>>> rf(n, k).is_hypergeometric(k)
True
>>> ff(n, k).is_hypergeometric(k)
True
```

To say more, all previously shown examples are valid for integer linear arguments:

```
>>> factorial(2*n).is_hypergeometric(n)
True
>>> binomial(3*n+1, k).is_hypergeometric(n)
True
>>> rf(n+1, k-1).is_hypergeometric(n)
True
>>> ff(n-1, k+1).is_hypergeometric(n)
True
>>> gamma(5*n).is_hypergeometric(n)
True
>>> (2**(n-7)).is_hypergeometric(n)
True
```

However nonlinear arguments make those sequences fail to be hypergeometric:

```
>>> factorial(n**2).is_hypergeometric(n)
False
>>> (2**(n**3 + 1)).is_hypergeometric(n)
False
```

If not only the knowledge of being hypergeometric or not is needed, you can use `hypersimp()` function. It will try to simplify combinatorial expression and if the term given is hypergeometric it will return a quotient of polynomials of minimal degree. Otherwise is will return $None$ to say that sequence is not hypergeometric:

```
>>> hypersimp(factorial(2*n), n)
2*(n + 1)*(2*n + 1)
>>> hypersimp(factorial(n**2), n)
```

Concrete Class Reference

class sympy.concrete.summations.**Sum**(*function*, **symbols*, ***assumptions*)

Represents unevaluated summation.

Explanation

`Sum` represents a finite or infinite series, with the first argument being the general form of terms in the series, and the second argument being `(dummy_variable, start, end)`, with `dummy_variable` taking all integer values from `start` through `end`. In accordance with long-standing mathematical convention, the end term is included in the summation.

Finite Sums

For finite sums (and sums with symbolic limits assumed to be finite) we follow the summation convention described by Karr [1], especially definition 3 of section 1.4. The sum:

$$\sum_{m \le i < n} f(i)$$

has *the obvious meaning* for $m < n$, namely:

$$\sum_{m \le i < n} f(i) = f(m) + f(m+1) + \ldots + f(n-2) + f(n-1)$$

with the upper limit value $f(n)$ excluded. The sum over an empty set is zero if and only if $m = n$:

$$\sum_{m \le i < n} f(i) = 0 \quad \text{for} \quad m = n$$

Finally, for all other sums over empty sets we assume the following definition:

$$\sum_{m \le i < n} f(i) = -\sum_{n \le i < m} f(i) \quad \text{for} \quad m > n$$

It is important to note that Karr defines all sums with the upper limit being exclusive. This is in contrast to the usual mathematical notation, but does not affect the summation convention. Indeed we have:

$$\sum_{m \le i < n} f(i) = \sum_{i=m}^{n-1} f(i)$$

where the difference in notation is intentional to emphasize the meaning, with limits typeset on the top being inclusive.

Examples

```
>>> from sympy.abc import i, k, m, n, x
>>> from sympy import Sum, factorial, oo, IndexedBase, Function
>>> Sum(k, (k, 1, m))
Sum(k, (k, 1, m))
>>> Sum(k, (k, 1, m)).doit()
m**2/2 + m/2
>>> Sum(k**2, (k, 1, m))
Sum(k**2, (k, 1, m))
>>> Sum(k**2, (k, 1, m)).doit()
m**3/3 + m**2/2 + m/6
>>> Sum(x**k, (k, 0, oo))
Sum(x**k, (k, 0, oo))
>>> Sum(x**k, (k, 0, oo)).doit()
Piecewise((1/(1 - x), Abs(x) < 1), (Sum(x**k, (k, 0, oo)), True))
>>> Sum(x**k/factorial(k), (k, 0, oo)).doit()
exp(x)
```

Here are examples to do summation with symbolic indices. You can use either Function of IndexedBase classes:

```
>>> f = Function('f')
>>> Sum(f(n), (n, 0, 3)).doit()
f(0) + f(1) + f(2) + f(3)
>>> Sum(f(n), (n, 0, oo)).doit()
Sum(f(n), (n, 0, oo))
>>> f = IndexedBase('f')
>>> Sum(f[n]**2, (n, 0, 3)).doit()
f[0]**2 + f[1]**2 + f[2]**2 + f[3]**2
```

An example showing that the symbolic result of a summation is still valid for seemingly nonsensical values of the limits. Then the Karr convention allows us to give a perfectly valid interpretation to those sums by interchanging the limits according to the above rules:

```
>>> S = Sum(i, (i, 1, n)).doit()
>>> S
n**2/2 + n/2
>>> S.subs(n, -4)
6
>>> Sum(i, (i, 1, -4)).doit()
6
>>> Sum(-i, (i, -3, 0)).doit()
6
```

An explicit example of the Karr summation convention:

```
>>> S1 = Sum(i**2, (i, m, m+n-1)).doit()
>>> S1
m**2*n + m*n**2 - m*n + n**3/3 - n**2/2 + n/6
>>> S2 = Sum(i**2, (i, m+n, m-1)).doit()
>>> S2
```

(continues on next page)

(continued from previous page)

```
-m**2*n - m*n**2 + m*n - n**3/3 + n**2/2 - n/6
>>> S1 + S2
0
>>> S3 = Sum(i, (i, m, m-1)).doit()
>>> S3
0
```

See also:

summation (page 915), *Product* (page 905), *sympy.concrete.products.product* (page 916)

References

[R87], [R88], [R89]

euler_maclaurin(*m=0*, *n=0*, *eps=0*, *eval_integral=True*)

Return an Euler-Maclaurin approximation of self, where m is the number of leading terms to sum directly and n is the number of terms in the tail.

With m = n = 0, this is simply the corresponding integral plus a first-order endpoint correction.

Returns (s, e) where s is the Euler-Maclaurin approximation and e is the estimated error (taken to be the magnitude of the first omitted term in the tail):

```
>>> from sympy.abc import k, a, b
>>> from sympy import Sum
>>> Sum(1/k, (k, 2, 5)).doit().evalf()
1.28333333333333
>>> s, e = Sum(1/k, (k, 2, 5)).euler_maclaurin()
>>> s
-log(2) + 7/20 + log(5)
>>> from sympy import sstr
>>> print(sstr((s.evalf(), e.evalf()), full_prec=True))
(1.26629073187415, 0.0175000000000000)
```

The endpoints may be symbolic:

```
>>> s, e = Sum(1/k, (k, a, b)).euler_maclaurin()
>>> s
-log(a) + log(b) + 1/(2*b) + 1/(2*a)
>>> e
Abs(1/(12*b**2) - 1/(12*a**2))
```

If the function is a polynomial of degree at most 2n+1, the Euler-Maclaurin formula becomes exact (and e = 0 is returned):

```
>>> Sum(k, (k, 2, b)).euler_maclaurin()
(b**2/2 + b/2 - 1, 0)
>>> Sum(k, (k, 2, b)).doit()
b**2/2 + b/2 - 1
```

With a nonzero eps specified, the summation is ended as soon as the remainder term is less than the epsilon.

eval_zeta_function(*f*, *limits*)

Check whether the function matches with the zeta function. If it matches, then return a *Piecewise* expression because zeta function does not converge unless $s > 1$ and $q > 0$

is_absolutely_convergent()

Checks for the absolute convergence of an infinite series.

Same as checking convergence of absolute value of sequence_term of an infinite series.

Examples

```
>>> from sympy import Sum, Symbol, oo
>>> n = Symbol('n', integer=True)
>>> Sum((-1)**n, (n, 1, oo)).is_absolutely_convergent()
False
>>> Sum((-1)**n/n**2, (n, 1, oo)).is_absolutely_convergent()
True
```

See also:

Sum.is_convergent (page 903)

References

[R90]

is_convergent()

Checks for the convergence of a Sum.

Explanation

We divide the study of convergence of infinite sums and products in two parts.

First Part: One part is the question whether all the terms are well defined, i.e., they are finite in a sum and also non-zero in a product. Zero is the analogy of (minus) infinity in products as $e^{-\infty} = 0$.

Second Part: The second part is the question of convergence after infinities, and zeros in products, have been omitted assuming that their number is finite. This means that we only consider the tail of the sum or product, starting from some point after which all terms are well defined.

For example, in a sum of the form:

$$\sum_{1 \le i < \infty} \frac{1}{n^2 + an + b}$$

where a and b are numbers. The routine will return true, even if there are infinities in the term sequence (at most two). An analogous product would be:

$$\prod_{1\leq i<\infty} e^{\frac{1}{n^2+an+b}}$$

This is how convergence is interpreted. It is concerned with what happens at the limit. Finding the bad terms is another independent matter.

Note: It is responsibility of user to see that the sum or product is well defined.

There are various tests employed to check the convergence like divergence test, root test, integral test, alternating series test, comparison tests, Dirichlet tests. It returns true if Sum is convergent and false if divergent and NotImplementedError if it cannot be checked.

Examples

```
>>> from sympy import factorial, S, Sum, Symbol, oo
>>> n = Symbol('n', integer=True)
>>> Sum(n/(n - 1), (n, 4, 7)).is_convergent()
True
>>> Sum(n/(2*n + 1), (n, 1, oo)).is_convergent()
False
>>> Sum(factorial(n)/5**n, (n, 1, oo)).is_convergent()
False
>>> Sum(1/n**(S(6)/5), (n, 1, oo)).is_convergent()
True
```

See also:

Sum.is_absolutely_convergent (page 903), *sympy.concrete.products.Product.is_convergent* (page 908)

References

[R91]

reverse_order(**indices*)

Reverse the order of a limit in a Sum.

Explanation

`reverse_order(self, *indices)` reverses some limits in the expression `self` which can be either a `Sum` or a `Product`. The selectors in the argument `indices` specify some indices whose limits get reversed. These selectors are either variable names or numerical indices counted starting from the inner-most limit tuple.

Examples

```
>>> from sympy import Sum
>>> from sympy.abc import x, y, a, b, c, d
```

```
>>> Sum(x, (x, 0, 3)).reverse_order(x)
Sum(-x, (x, 4, -1))
>>> Sum(x*y, (x, 1, 5), (y, 0, 6)).reverse_order(x, y)
Sum(x*y, (x, 6, 0), (y, 7, -1))
>>> Sum(x, (x, a, b)).reverse_order(x)
Sum(-x, (x, b + 1, a - 1))
>>> Sum(x, (x, a, b)).reverse_order(0)
Sum(-x, (x, b + 1, a - 1))
```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```
>>> S = Sum(x**2, (x, a, b), (x, c, d))
>>> S
Sum(x**2, (x, a, b), (x, c, d))
>>> S0 = S.reverse_order(0)
>>> S0
Sum(-x**2, (x, b + 1, a - 1), (x, c, d))
>>> S1 = S0.reverse_order(1)
>>> S1
Sum(x**2, (x, b + 1, a - 1), (x, d + 1, c - 1))
```

Of course we can mix both notations:

```
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(x, 1)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(y, x)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
```

See also:

sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index (page 913), *reorder_limit* (page 914), *sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder* (page 914)

References

[R92]

class sympy.concrete.products.**Product**(*function*, **symbols*, ***assumptions*)

Represents unevaluated products.

Explanation

`Product` represents a finite or infinite product, with the first argument being the general form of terms in the series, and the second argument being `(dummy_variable, start, end)`, with `dummy_variable` taking all integer values from `start` through `end`. In accordance with long-standing mathematical convention, the end term is included in the product.

Finite Products

For finite products (and products with symbolic limits assumed to be finite) we follow the analogue of the summation convention described by Karr [1], especially definition 3 of section 1.4. The product:

$$\prod_{m \le i < n} f(i)$$

has *the obvious meaning* for $m < n$, namely:

$$\prod_{m \le i < n} f(i) = f(m) f(m+1) \cdot \ldots \cdot f(n-2) f(n-1)$$

with the upper limit value $f(n)$ excluded. The product over an empty set is one if and only if $m = n$:

$$\prod_{m \le i < n} f(i) = 1 \quad \text{for} \quad m = n$$

Finally, for all other products over empty sets we assume the following definition:

$$\prod_{m \le i < n} f(i) = \frac{1}{\prod_{n \le i < m} f(i)} \quad \text{for} \quad m > n$$

It is important to note that above we define all products with the upper limit being exclusive. This is in contrast to the usual mathematical notation, but does not affect the product convention. Indeed we have:

$$\prod_{m \le i < n} f(i) = \prod_{i=m}^{n-1} f(i)$$

where the difference in notation is intentional to emphasize the meaning, with limits typeset on the top being inclusive.

Examples

```
>>> from sympy.abc import a, b, i, k, m, n, x
>>> from sympy import Product, oo
>>> Product(k, (k, 1, m))
Product(k, (k, 1, m))
>>> Product(k, (k, 1, m)).doit()
factorial(m)
>>> Product(k**2,(k, 1, m))
Product(k**2, (k, 1, m))
>>> Product(k**2,(k, 1, m)).doit()
factorial(m)**2
```

Wallis' product for pi:

```
>>> W = Product(2*i/(2*i-1) * 2*i/(2*i+1), (i, 1, oo))
>>> W
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, oo))
```

Direct computation currently fails:

```
>>> W.doit()
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, oo))
```

But we can approach the infinite product by a limit of finite products:

```
>>> from sympy import limit
>>> W2 = Product(2*i/(2*i-1)*2*i/(2*i+1), (i, 1, n))
>>> W2
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, n))
>>> W2e = W2.doit()
>>> W2e
4**n*factorial(n)**2/(2**(2*n)*RisingFactorial(1/2, n)*RisingFactorial(3/
↪2, n))
>>> limit(W2e, n, oo)
pi/2
```

By the same formula we can compute sin(pi/2):

```
>>> from sympy import combsimp, pi, gamma, simplify
>>> P = pi * x * Product(1 - x**2/k**2, (k, 1, n))
>>> P = P.subs(x, pi/2)
>>> P
pi**2*Product(1 - pi**2/(4*k**2), (k, 1, n))/2
>>> Pe = P.doit()
>>> Pe
pi**2*RisingFactorial(1 - pi/2, n)*RisingFactorial(1 + pi/2, n)/
↪(2*factorial(n)**2)
>>> limit(Pe, n, oo).gammasimp()
sin(pi**2/2)
>>> Pe.rewrite(gamma)
(-1)**n*pi**2*gamma(pi/2)*gamma(n + 1 + pi/2)/(2*gamma(1 + pi/2)*gamma(-
↪n + pi/2)*gamma(n + 1)**2)
```

Products with the lower limit being larger than the upper one:

```
>>> Product(1/i, (i, 6, 1)).doit()
120
>>> Product(i, (i, 2, 5)).doit()
120
```

The empty product:

```
>>> Product(i, (i, n, n-1)).doit()
1
```

An example showing that the symbolic result of a product is still valid for seemingly nonsensical values of the limits. Then the Karr convention allows us to give a perfectly

valid interpretation to those products by interchanging the limits according to the above rules:

```
>>> P = Product(2, (i, 10, n)).doit()
>>> P
2**(n - 9)
>>> P.subs(n, 5)
1/16
>>> Product(2, (i, 10, 5)).doit()
1/16
>>> 1/Product(2, (i, 6, 9)).doit()
1/16
```

An explicit example of the Karr summation convention applied to products:

```
>>> P1 = Product(x, (i, a, b)).doit()
>>> P1
x**(-a + b + 1)
>>> P2 = Product(x, (i, b+1, a-1)).doit()
>>> P2
x**(a - b - 1)
>>> simplify(P1 * P2)
1
```

And another one:

```
>>> P1 = Product(i, (i, b, a)).doit()
>>> P1
RisingFactorial(b, a - b + 1)
>>> P2 = Product(i, (i, a+1, b-1)).doit()
>>> P2
RisingFactorial(a + 1, -a + b - 1)
>>> P1 * P2
RisingFactorial(b, a - b + 1)*RisingFactorial(a + 1, -a + b - 1)
>>> combsimp(P1 * P2)
1
```

See also:

Sum (page 900), *summation* (page 915), *product* (page 916)

References

[R93], [R94], [R95]

is_convergent()

See docs of *Sum.is_convergent()* (page 903) for explanation of convergence in SymPy.

Explanation

The infinite product:

$$\prod_{1\leq i<\infty} f(i)$$

is defined by the sequence of partial products:

$$\prod_{i=1}^{n} f(i) = f(1)f(2)\cdots f(n)$$

as n increases without bound. The product converges to a non-zero value if and only if the sum:

$$\sum_{1\leq i<\infty} \log f(n)$$

converges.

Examples

```
>>> from sympy import Product, Symbol, cos, pi, exp, oo
>>> n = Symbol('n', integer=True)
>>> Product(n/(n + 1), (n, 1, oo)).is_convergent()
False
>>> Product(1/n**2, (n, 1, oo)).is_convergent()
False
>>> Product(cos(pi/n), (n, 1, oo)).is_convergent()
True
>>> Product(exp(-n**2), (n, 1, oo)).is_convergent()
False
```

References

[R96]

reverse_order(**indices*)

Reverse the order of a limit in a Product.

Explanation

`reverse_order(expr, *indices)` reverses some limits in the expression `expr` which can be either a `Sum` or a `Product`. The selectors in the argument `indices` specify some indices whose limits get reversed. These selectors are either variable names or numerical indices counted starting from the inner-most limit tuple.

Examples

```
>>> from sympy import gamma, Product, simplify, Sum
>>> from sympy.abc import x, y, a, b, c, d
>>> P = Product(x, (x, a, b))
>>> Pr = P.reverse_order(x)
>>> Pr
Product(1/x, (x, b + 1, a - 1))
>>> Pr = Pr.doit()
>>> Pr
1/RisingFactorial(b + 1, a - b - 1)
>>> simplify(Pr.rewrite(gamma))
Piecewise((gamma(b + 1)/gamma(a), b > -1), ((-1)**(-a + b +␣
↪1)*gamma(1 - a)/gamma(-b), True))
>>> P = P.doit()
>>> P
RisingFactorial(a, -a + b + 1)
>>> simplify(P.rewrite(gamma))
Piecewise((gamma(b + 1)/gamma(a), a > 0), ((-1)**(-a + b + 1)*gamma(1␣
↪- a)/gamma(-b), True))
```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```
>>> S = Sum(x*y, (x, a, b), (y, c, d))
>>> S
Sum(x*y, (x, a, b), (y, c, d))
>>> S0 = S.reverse_order(0)
>>> S0
Sum(-x*y, (x, b + 1, a - 1), (y, c, d))
>>> S1 = S0.reverse_order(1)
>>> S1
Sum(x*y, (x, b + 1, a - 1), (y, d + 1, c - 1))
```

Of course we can mix both notations:

```
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(x, 1)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(y, x)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
```

See also:

sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index (page 913), *reorder_limit* (page 914), *sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder* (page 914)

References

[R97]

class sympy.concrete.expr_with_intlimits.**ExprWithIntLimits**(*function*, **symbols*, ***assumptions*)

Superclass for Product and Sum.

See also:

sympy.concrete.expr_with_limits.ExprWithLimits (page 606), *sympy.concrete.products.Product* (page 905), *sympy.concrete.summations.Sum* (page 900)

change_index(*var*, *trafo*, *newvar=None*)

Change index of a Sum or Product.

Perform a linear transformation $x \mapsto ax + b$ on the index variable x. For a the only values allowed are ± 1. A new variable to be used after the change of index can also be specified.

Explanation

`change_index(expr, var, trafo, newvar=None)` where `var` specifies the index variable x to transform. The transformation `trafo` must be linear and given in terms of `var`. If the optional argument `newvar` is provided then `var` gets replaced by `newvar` in the final expression.

Examples

```
>>> from sympy import Sum, Product, simplify
>>> from sympy.abc import x, y, a, b, c, d, u, v, i, j, k, l
```

```
>>> S = Sum(x, (x, a, b))
>>> S.doit()
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> Sn = S.change_index(x, x + 1, y)
>>> Sn
Sum(y - 1, (y, a + 1, b + 1))
>>> Sn.doit()
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> Sn = S.change_index(x, -x, y)
>>> Sn
Sum(-y, (y, -b, -a))
>>> Sn.doit()
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> Sn = S.change_index(x, x+u)
>>> Sn
Sum(-u + x, (x, a + u, b + u))
```

(continues on next page)

(continued from previous page)

```
>>> Sn.doit()
-a**2/2 - a*u + a/2 + b**2/2 + b*u + b/2 - u*(-a + b + 1) + u
>>> simplify(Sn.doit())
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> Sn = S.change_index(x, -x - u, y)
>>> Sn
Sum(-u - y, (y, -b - u, -a - u))
>>> Sn.doit()
-a**2/2 - a*u + a/2 + b**2/2 + b*u + b/2 - u*(-a + b + 1) + u
>>> simplify(Sn.doit())
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> P = Product(i*j**2, (i, a, b), (j, c, d))
>>> P
Product(i*j**2, (i, a, b), (j, c, d))
>>> P2 = P.change_index(i, i+3, k)
>>> P2
Product(j**2*(k - 3), (k, a + 3, b + 3), (j, c, d))
>>> P3 = P2.change_index(j, -j, l)
>>> P3
Product(l**2*(k - 3), (k, a + 3, b + 3), (l, -d, -c))
```

When dealing with symbols only, we can make a general linear transformation:

```
>>> Sn = S.change_index(x, u*x+v, y)
>>> Sn
Sum((-v + y)/u, (y, b*u + v, a*u + v))
>>> Sn.doit()
-v*(a*u - b*u + 1)/u + (a**2*u**2/2 + a*u*v + a*u/2 - b**2*u**2/2 -␣
↪b*u*v + b*u/2 + v)/u
>>> simplify(Sn.doit())
a**2*u/2 + a/2 - b**2*u/2 + b/2
```

However, the last result can be inconsistent with usual summation where the index increment is always 1. This is obvious as we get back the original value only for u equal +1 or -1.

See also:

sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index (page 913), *reorder_limit* (page 914), *sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder* (page 914), *sympy.concrete.summations.Sum.reverse_order* (page 904), *sympy.concrete.products.Product.reverse_order* (page 909)

property has_empty_sequence

Returns True if the Sum or Product is computed for an empty sequence.

Examples

```
>>> from sympy import Sum, Product, Symbol
>>> m = Symbol('m')
>>> Sum(m, (m, 1, 0)).has_empty_sequence
True
```

```
>>> Sum(m, (m, 1, 1)).has_empty_sequence
False
```

```
>>> M = Symbol('M', integer=True, positive=True)
>>> Product(m, (m, 1, M)).has_empty_sequence
False
```

```
>>> Product(m, (m, 2, M)).has_empty_sequence
```

```
>>> Product(m, (m, M + 1, M)).has_empty_sequence
True
```

```
>>> N = Symbol('N', integer=True, positive=True)
>>> Sum(m, (m, N, M)).has_empty_sequence
```

```
>>> N = Symbol('N', integer=True, negative=True)
>>> Sum(m, (m, N, M)).has_empty_sequence
False
```

See also:

has_reversed_limits (page 607), *has_finite_limits* (page 607)

index(*x*)

Return the index of a dummy variable in the list of limits.

Explanation

`index(expr, x)` returns the index of the dummy variable `x` in the limits of `expr`. Note that we start counting with 0 at the inner-most limits tuple.

Examples

```
>>> from sympy.abc import x, y, a, b, c, d
>>> from sympy import Sum, Product
>>> Sum(x*y, (x, a, b), (y, c, d)).index(x)
0
>>> Sum(x*y, (x, a, b), (y, c, d)).index(y)
1
>>> Product(x*y, (x, a, b), (y, c, d)).index(x)
0
>>> Product(x*y, (x, a, b), (y, c, d)).index(y)
1
```

See also:

reorder_limit (page 914), *reorder* (page 914), *sympy.concrete.summations.Sum.reverse_order* (page 904), *sympy.concrete.products.Product.reverse_order* (page 909)

reorder(**arg*)

Reorder limits in a expression containing a Sum or a Product.

Explanation

`expr.reorder(*arg)` reorders the limits in the expression `expr` according to the list of tuples given by `arg`. These tuples can contain numerical indices or index variable names or involve both.

Examples

```
>>> from sympy import Sum, Product
>>> from sympy.abc import x, y, z, a, b, c, d, e, f
```

```
>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((x, y))
Sum(x*y, (y, c, d), (x, a, b))
```

```
>>> Sum(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder((x, y), (x,␣
↪z), (y, z))
Sum(x*y*z, (z, e, f), (y, c, d), (x, a, b))
```

```
>>> P = Product(x*y*z, (x, a, b), (y, c, d), (z, e, f))
>>> P.reorder((x, y), (x, z), (y, z))
Product(x*y*z, (z, e, f), (y, c, d), (x, a, b))
```

We can also select the index variables by counting them, starting with the inner-most one:

```
>>> Sum(x**2, (x, a, b), (x, c, d)).reorder((0, 1))
Sum(x**2, (x, c, d), (x, a, b))
```

And of course we can mix both schemes:

```
>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((y, x))
Sum(x*y, (y, c, d), (x, a, b))
>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((y, 0))
Sum(x*y, (y, c, d), (x, a, b))
```

See also:

reorder_limit (page 914), *index* (page 913), *sympy.concrete.summations.Sum.reverse_order* (page 904), *sympy.concrete.products.Product.reverse_order* (page 909)

reorder_limit(*x*, *y*)

Interchange two limit tuples of a Sum or Product expression.

Explanation

`expr.reorder_limit(x, y)` interchanges two limit tuples. The arguments `x` and `y` are integers corresponding to the index variables of the two limits which are to be interchanged. The expression `expr` has to be either a Sum or a Product.

Examples

```
>>> from sympy.abc import x, y, z, a, b, c, d, e, f
>>> from sympy import Sum, Product
```

```
>>> Sum(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder_limit(0, 2)
Sum(x*y*z, (z, e, f), (y, c, d), (x, a, b))
>>> Sum(x**2, (x, a, b), (x, c, d)).reorder_limit(1, 0)
Sum(x**2, (x, c, d), (x, a, b))
```

```
>>> Product(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder_limit(0,␣
↪2)
Product(x*y*z, (z, e, f), (y, c, d), (x, a, b))
```

See also:

index (page 913), *reorder* (page 914), *sympy.concrete.summations.Sum.reverse_order* (page 904), *sympy.concrete.products.Product.reverse_order* (page 909)

Concrete Functions Reference

`sympy.concrete.summations.`**`summation`**(*f*, **symbols*, ***kwargs*)

Compute the summation of f with respect to symbols.

Explanation

The notation for symbols is similar to the notation used in Integral. summation(f, (i, a, b)) computes the sum of f with respect to i from a to b, i.e.,

```
                                b
                              _____
                              \    `
summation(f, (i, a, b)) =      )     f
                              /____,
                              i = a
```

If it cannot compute the sum, it returns an unevaluated Sum object. Repeated sums can be computed by introducing additional symbols tuples:

```
.. rubric:: Examples
```

```
>>> from sympy import summation, oo, symbols, log
>>> i, n, m = symbols('i n m', integer=True)
```

```
>>> summation(2*i - 1, (i, 1, n))
n**2
>>> summation(1/2**i, (i, 0, oo))
2
>>> summation(1/log(n)**n, (n, 2, oo))
Sum(log(n)**(-n), (n, 2, oo))
>>> summation(i, (i, 0, n), (n, 0, m))
m**3/6 + m**2/2 + m/3
```

```
>>> from sympy.abc import x
>>> from sympy import factorial
>>> summation(x**n/factorial(n), (n, 0, oo))
exp(x)
```

See also:

Sum (page 900), *Product* (page 905), *sympy.concrete.products.product* (page 916)

sympy.concrete.products.**product**(**args*, ***kwargs*)

Compute the product.

Explanation

The notation for symbols is similar to the notation used in Sum or Integral. product(f, (i, a, b)) computes the product of f with respect to i from a to b, i.e.,

```
                                b
                              _____
product(f(n), (i, a, b)) = |   | f(n)
                           |   |
                           i = a
```

If it cannot compute the product, it returns an unevaluated Product object. Repeated products can be computed by introducing additional symbols tuples:

```
.. rubric:: Examples
```

```
>>> from sympy import product, symbols
>>> i, n, m, k = symbols('i n m k', integer=True)
```

```
>>> product(i, (i, 1, k))
factorial(k)
>>> product(m, (i, 1, k))
m**k
>>> product(i, (i, 1, k), (k, 1, n))
Product(factorial(k), (k, 1, n))
```

sympy.concrete.gosper.**gosper_normal**(*f*, *g*, *n*, *polys=True*)

Compute the Gosper's normal form of f and g.