Examples

```
>>> from sympy import EmptySequence, oo, SeqAdd, SeqPer, SeqFormula
>>> from sympy.abc import n
>>> SeqAdd(SeqPer((1, 2), (n, 0, oo)), EmptySequence)
SeqPer((1, 2), (n, 0, oo))
>>> SeqAdd(SeqPer((1, 2), (n, 0, 5)), SeqPer((1, 2), (n, 6, 10)))
EmptySequence
>>> SeqAdd(SeqPer((1, 2), (n, 0, oo)), SeqFormula(n**2, (n, 0, oo)))
SeqAdd(SeqFormula(n**2, (n, 0, oo)), SeqPer((1, 2), (n, 0, oo)))
>>> SeqAdd(SeqFormula(n**3), SeqFormula(n**2))
SeqFormula(n**3 + n**2, (n, 0, oo))
```

See also:

sympy.series.sequences.SeqMul (page 637)

static reduce(*args*)

Simplify *SeqAdd* (page 636) using known rules.

Iterates through all pairs and ask the constituent sequences if they can simplify themselves with any other constituent.

Notes

adapted from `Union.reduce`

class sympy.series.sequences.**SeqMul**(**args*, ***kwargs*)

Represents term-wise multiplication of sequences.

Explanation

Handles multiplication of sequences only. For multiplication with other objects see *SeqBase.coeff_mul()* (page 632).

Rules:

- The interval on which sequence is defined is the intersection of respective intervals of sequences.
- Anything * *EmptySequence* (page 636) returns *EmptySequence* (page 636).
- Other rules are defined in `_mul` methods of sequence classes.

Examples

```
>>> from sympy import EmptySequence, oo, SeqMul, SeqPer, SeqFormula
>>> from sympy.abc import n
>>> SeqMul(SeqPer((1, 2), (n, 0, oo)), EmptySequence)
EmptySequence
>>> SeqMul(SeqPer((1, 2), (n, 0, 5)), SeqPer((1, 2), (n, 6, 10)))
EmptySequence
>>> SeqMul(SeqPer((1, 2), (n, 0, oo)), SeqFormula(n**2))
```

(continues on next page)

(continued from previous page)

```
SeqMul(SeqFormula(n**2, (n, 0, oo)), SeqPer((1, 2), (n, 0, oo)))
>>> SeqMul(SeqFormula(n**3), SeqFormula(n**2))
SeqFormula(n**5, (n, 0, oo))
```

See also:

sympy.series.sequences.SeqAdd (page 636)

static reduce(*args*)

Simplify a *SeqMul* (page 637) using known rules.

Explanation

Iterates through all pairs and ask the constituent sequences if they can simplify themselves with any other constituent.

Notes

adapted from `Union.reduce`

Recursive Sequences

class sympy.series.sequences.**RecursiveSeq**(*recurrence*, *yn*, *n*, *initial=None*, *start=0*)

A finite degree recursive sequence.

Parameters

recurrence : SymPy expression defining recurrence

This is *not* an equality, only the expression that the nth term is equal to. For example, if `a(n) = f(a(n - 1), ..., a(n - d))`, then the expression should be `f(a(n - 1), ..., a(n - d))`.

yn : applied undefined function

Represents the nth term of the sequence as e.g. `y(n)` where `y` is an undefined function and n is the sequence index.

n : symbolic argument

The name of the variable that the recurrence is in, e.g., `n` if the recurrence function is `y(n)`.

initial : iterable with length equal to the degree of the recurrence

The initial values of the recurrence.

start : start value of sequence (inclusive)

Explanation

That is, a sequence a(n) that depends on a fixed, finite number of its previous values. The general form is

a(n) = f(a(n - 1), a(n - 2), ..., a(n - d))

for some fixed, positive integer d, where f is some function defined by a SymPy expression.

Examples

```
>>> from sympy import Function, symbols
>>> from sympy.series.sequences import RecursiveSeq
>>> y = Function("y")
>>> n = symbols("n")
>>> fib = RecursiveSeq(y(n - 1) + y(n - 2), y(n), n, [0, 1])
```

```
>>> fib.coeff(3) # Value at a particular point
2
```

```
>>> fib[:6] # supports slicing
[0, 1, 1, 2, 3, 5]
```

```
>>> fib.recurrence # inspect recurrence
Eq(y(n), y(n - 2) + y(n - 1))
```

```
>>> fib.degree # automatically determine degree
2
```

```
>>> for x in zip(range(10), fib): # supports iteration
...     print(x)
(0, 0)
(1, 1)
(2, 1)
(3, 2)
(4, 3)
(5, 5)
(6, 8)
(7, 13)
(8, 21)
(9, 34)
```

See also:

sympy.series.sequences.SeqFormula (page 634)

property initial

The initial values of the sequence

property interval

Interval on which sequence is defined.

property n
Sequence index symbol

property recurrence
Equation defining recurrence.

property start
The starting point of the sequence. This point is included

property stop
The ending point of the sequence. (oo)

property y
Undefined function for the nth term of the sequence

property yn
Applied function representing the nth term

Fourier Series

Provides methods to compute Fourier series.

class `sympy.series.fourier.`**`FourierSeries`**`(*args)`
Represents Fourier sine/cosine series.

Explanation

This class only represents a fourier series. No computation is performed.

For how to compute Fourier series, see the *fourier_series()* (page 643) docstring.

See also:

sympy.series.fourier.fourier_series (page 643)

scale(*s*)
Scale the function by a term independent of x.

Explanation

f(x) -> s * f(x)

This is fast, if Fourier series of f(x) is already computed.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.scale(2).truncate()
-8*cos(x) + 2*cos(2*x) + 2*pi**2/3
```

scalex(*s*)
Scale x by a term independent of x.

Explanation

f(x) -> f(s*x)

This is fast, if Fourier series of f(x) is already computed.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.scalex(2).truncate()
-4*cos(2*x) + cos(4*x) + pi**2/3
```

shift(*s*)

Shift the function by a term independent of x.

Explanation

f(x) -> f(x) + s

This is fast, if Fourier series of f(x) is already computed.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.shift(1).truncate()
-4*cos(x) + cos(2*x) + 1 + pi**2/3
```

shiftx(*s*)

Shift x by a term independent of x.

Explanation

f(x) -> f(x + s)

This is fast, if Fourier series of f(x) is already computed.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.shiftx(1).truncate()
-4*cos(x + 1) + cos(2*x + 2) + pi**2/3
```

sigma_approximation(*n=3*)

Return σ-approximation of Fourier series with respect to order n.

Parameters

n : int

Highest order of the terms taken into account in approximation.

Returns

Expr :

Sigma approximation of function expanded into Fourier series.

Explanation

Sigma approximation adjusts a Fourier summation to eliminate the Gibbs phenomenon which would otherwise occur at discontinuities. A sigma-approximated summation for a Fourier series of a T-periodical function can be written as

$$s(\theta) = \frac{1}{2}a_0 + \sum_{k=1}^{m-1} \operatorname{sinc}\Bigl(\frac{k}{m}\Bigr) \cdot \left[a_k \cos\Bigl(\frac{2\pi k}{T}\theta\Bigr) + b_k \sin\Bigl(\frac{2\pi k}{T}\theta\Bigr)\right],$$

where $a_0, a_k, b_k, k = 1, \ldots, m-1$ are standard Fourier series coefficients and $\operatorname{sinc}\left(\frac{k}{m}\right)$ is a Lanczos σ factor (expressed in terms of normalized sinc function).

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x, (x, -pi, pi))
>>> s.sigma_approximation(4)
2*sin(x)*sinc(pi/4) - 2*sin(2*x)/pi + 2*sin(3*x)*sinc(3*pi/4)/3
```

Notes

The behaviour of *sigma_approximation()* (page 642) is different from *truncate()* (page 643) - it takes all nonzero terms of degree smaller than n, rather than first n nonzero ones.

See also:

sympy.series.fourier.FourierSeries.truncate (page 643)

References

[R737], [R738]

truncate(*n=3*)

Return the first n nonzero terms of the series.

If n is None return an iterator.

Parameters

n : int or None

Amount of non-zero terms in approximation or None.

Returns

Expr or iterator :

Approximation of function expanded into Fourier series.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x, (x, -pi, pi))
>>> s.truncate(4)
2*sin(x) - sin(2*x) + 2*sin(3*x)/3 - sin(4*x)/2
```

See also:

sympy.series.fourier.FourierSeries.sigma_approximation (page 642)

sympy.series.fourier.**fourier_series**(*f*, *limits=None*, *finite=True*)

Computes the Fourier trigonometric series expansion.

Parameters

limits : (sym, start, end), optional

sym denotes the symbol the series is computed with respect to.

start and *end* denotes the start and the end of the interval where the fourier series converges to the given function.

Default range is specified as $-\pi$ and π.

Returns

FourierSeries

A symbolic object representing the Fourier trigonometric series.

Explanation

Fourier trigonometric series of $f(x)$ over the interval (a, b) is defined as:

$$\frac{a_0}{2} + \sum_{n=1}^{\infty}(a_n \cos(\frac{2n\pi x}{L}) + b_n \sin(\frac{2n\pi x}{L}))$$

where the coefficients are:

$$L = b - a$$

$$a_0 = \frac{2}{L}\int_a^b f(x)dx$$

$$a_n = \frac{2}{L}\int_a^b f(x)\cos(\frac{2n\pi x}{L})dx$$

$$b_n = \frac{2}{L}\int_a^b f(x)\sin(\frac{2n\pi x}{L})dx$$

The condition whether the function $f(x)$ given should be periodic or not is more than necessary, because it is sufficient to consider the series to be converging to $f(x)$ only in the given interval, not throughout the whole real line.

This also brings a lot of ease for the computation because you do not have to make $f(x)$ artificially periodic by wrapping it with piecewise, modulo operations, but you can shape the function to look like the desired periodic function only in the interval (a, b), and the computed series will automatically become the series of the periodic version of $f(x)$.

This property is illustrated in the examples section below.

Examples

Computing the Fourier series of $f(x) = x^2$:

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> f = x**2
>>> s = fourier_series(f, (x, -pi, pi))
>>> s1 = s.truncate(n=3)
>>> s1
-4*cos(x) + cos(2*x) + pi**2/3
```

Shifting of the Fourier series:

```
>>> s.shift(1).truncate()
-4*cos(x) + cos(2*x) + 1 + pi**2/3
>>> s.shiftx(1).truncate()
-4*cos(x + 1) + cos(2*x + 2) + pi**2/3
```

Scaling of the Fourier series:

```
>>> s.scale(2).truncate()
-8*cos(x) + 2*cos(2*x) + 2*pi**2/3
>>> s.scalex(2).truncate()
-4*cos(2*x) + cos(4*x) + pi**2/3
```

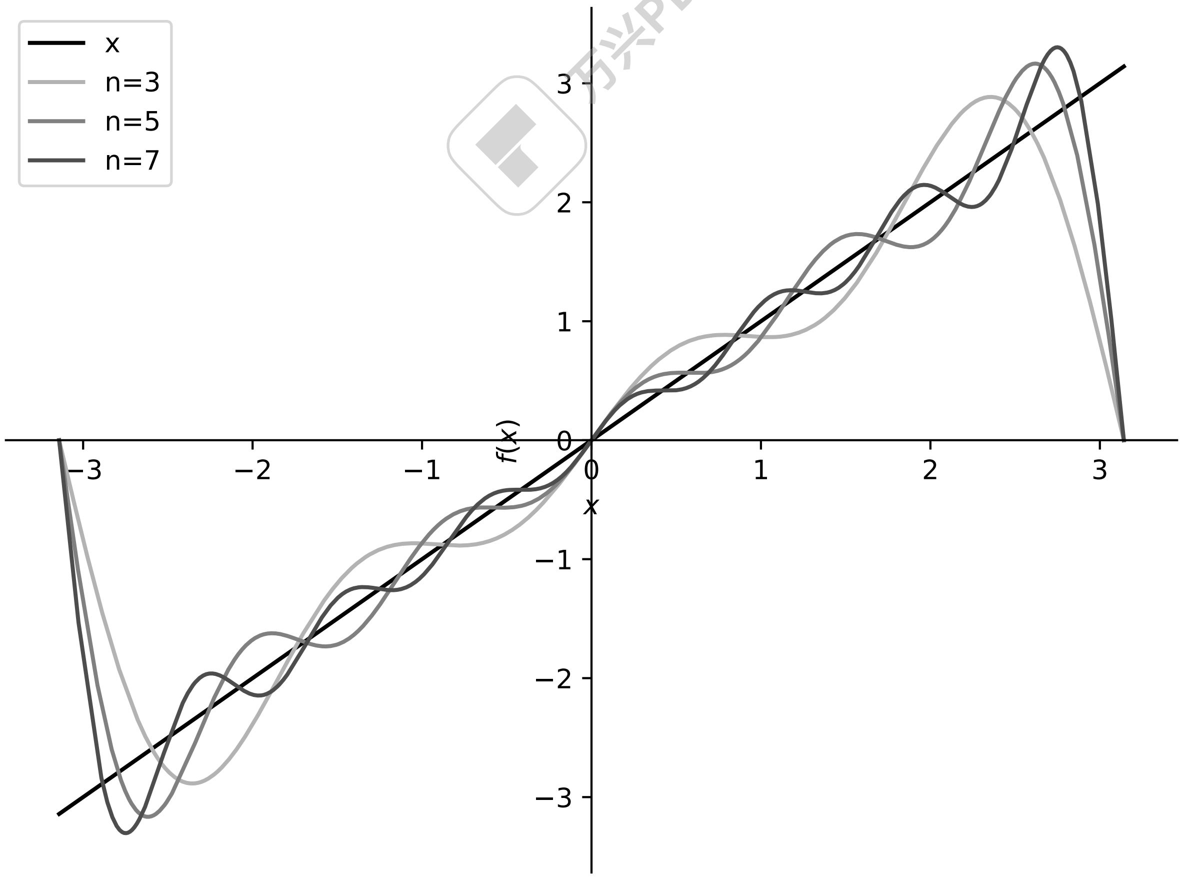
Computing the Fourier series of $f(x) = x$:

This illustrates how truncating to the higher order gives better convergence.

```
>>> from sympy import fourier_series, pi, plot
>>> from sympy.abc import x
>>> f = x
>>> s = fourier_series(f, (x, -pi, pi))
>>> s1 = s.truncate(n = 3)
>>> s2 = s.truncate(n = 5)
>>> s3 = s.truncate(n = 7)
>>> p = plot(f, s1, s2, s3, (x, -pi, pi), show=False, legend=True)
```

```
>>> p[0].line_color = (0, 0, 0)
>>> p[0].label = 'x'
>>> p[1].line_color = (0.7, 0.7, 0.7)
>>> p[1].label = 'n=3'
>>> p[2].line_color = (0.5, 0.5, 0.5)
>>> p[2].label = 'n=5'
>>> p[3].line_color = (0.3, 0.3, 0.3)
>>> p[3].label = 'n=7'
```

```
>>> p.show()
```

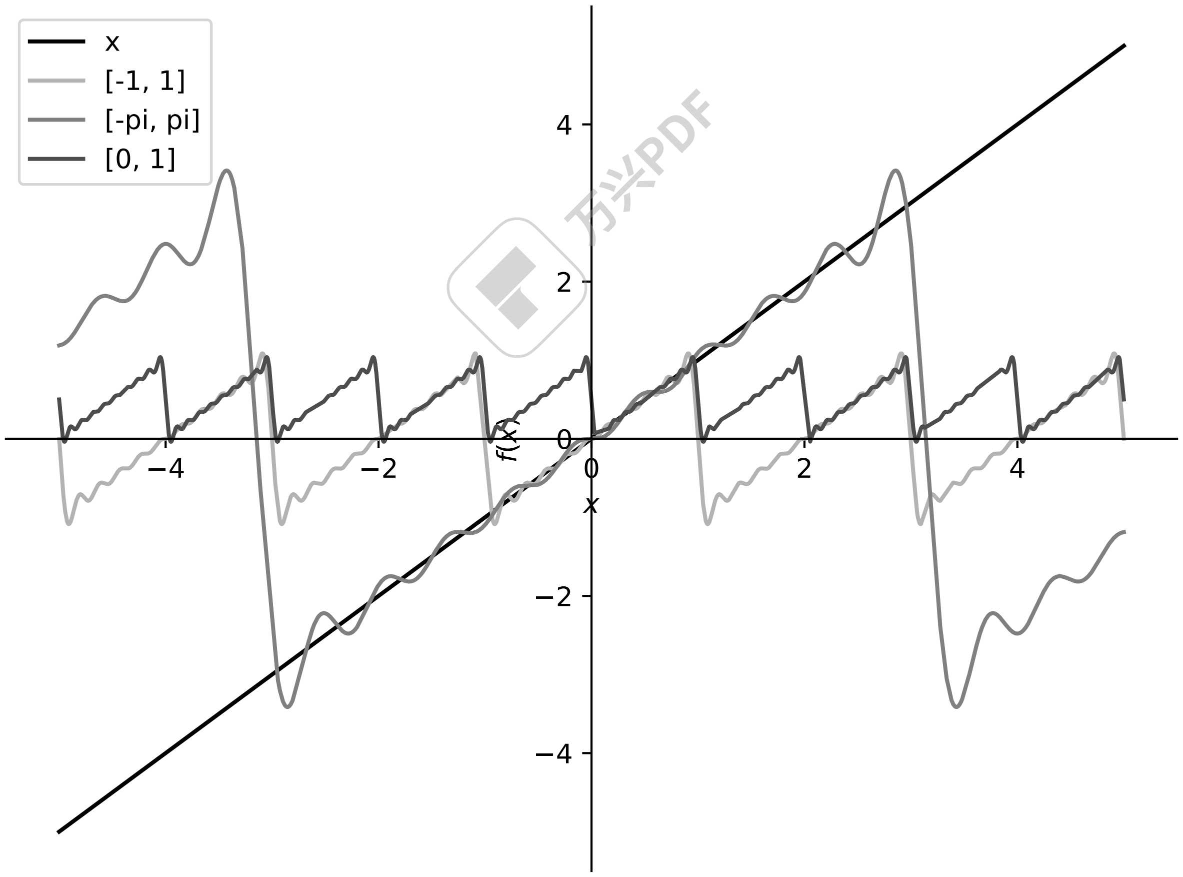


This illustrates how the series converges to different sawtooth waves if the different ranges are specified.

```
>>> s1 = fourier_series(x, (x, -1, 1)).truncate(10)
>>> s2 = fourier_series(x, (x, -pi, pi)).truncate(10)
>>> s3 = fourier_series(x, (x, 0, 1)).truncate(10)
>>> p = plot(x, s1, s2, s3, (x, -5, 5), show=False, legend=True)
```

```
>>> p[0].line_color = (0, 0, 0)
>>> p[0].label = 'x'
>>> p[1].line_color = (0.7, 0.7, 0.7)
>>> p[1].label = '[-1, 1]'
>>> p[2].line_color = (0.5, 0.5, 0.5)
>>> p[2].label = '[-pi, pi]'
>>> p[3].line_color = (0.3, 0.3, 0.3)
>>> p[3].label = '[0, 1]'
```

```
>>> p.show()
```



Notes

Computing Fourier series can be slow due to the integration required in computing an, bn.

It is faster to compute Fourier series of a function by using shifting and scaling on an already computed Fourier series rather than computing again.

e.g. If the Fourier series of `x**2` is known the Fourier series of `x**2 - 1` can be found by shifting by `-1`.

See also:

sympy.series.fourier.FourierSeries (page 640)

References

[R739]

Formal Power Series

Methods for computing and manipulating Formal Power Series.

`class sympy.series.formal.`**`FormalPowerSeries`**`(*args)`

Represents Formal Power Series of a function.

Explanation

No computation is performed. This class should only to be used to represent a series. No checks are performed.

For computing a series use *fps()* (page 650).

See also:

sympy.series.formal.fps (page 650)

`coeff_bell``(n)`

self.coeff_bell(n) returns a sequence of Bell polynomials of the second kind. Note that `n` should be a integer.

The second kind of Bell polynomials (are sometimes called "partial" Bell polynomials or incomplete Bell polynomials) are defined as

$$B_{n,k}(x_1, x_2, \dots x_{n-k+1}) = \sum_{\substack{j_1+j_2+j_2+\cdots=k \\ j_1+2j_2+3j_2+\cdots=n}} \frac{n!}{j_1!j_2!\cdots j_{n-k+1}!} \left(\frac{x_1}{1!}\right)^{j_1} \left(\frac{x_2}{2!}\right)^{j_2} \cdots \left(\frac{x_{n-k+1}}{(n-k+1)!}\right)^{j_{n-k+1}}.$$

- `bell(n, k, (x1, x2, ...))` gives Bell polynomials of the second kind, $B_{n,k}(x_1, x_2, \dots, x_{n-k+1})$.

See also:

sympy.functions.combinatorial.numbers.bell (page 427)

compose(*other*, *x=None*, *n=6*)

Returns the truncated terms of the formal power series of the composed function, up to specified n.

Parameters

n : Number, optional

Specifies the order of the term up to which the polynomial should be truncated.

Explanation

If f and g are two formal power series of two different functions, then the coefficient sequence ak of the composed formal power series fp will be as follows.

$$\sum_{k=0}^{n} b_k B_{n,k}(x_1, x_2, \dots, x_{n-k+1})$$

Examples

```
>>> from sympy import fps, sin, exp
>>> from sympy.abc import x
>>> f1 = fps(exp(x))
>>> f2 = fps(sin(x))
```

```
>>> f1.compose(f2, x).truncate()
1 + x + x**2/2 - x**4/8 - x**5/15 + O(x**6)
```

```
>>> f1.compose(f2, x).truncate(8)
1 + x + x**2/2 - x**4/8 - x**5/15 - x**6/240 + x**7/90 + O(x**8)
```

See also:

sympy.functions.combinatorial.numbers.bell (page 427), *sympy.series.formal.FormalPowerSeriesCompose* (page 653)

References

[R731]

property infinite

Returns an infinite representation of the series

integrate(*x=None*, ***kwargs*)

Integrate Formal Power Series.

Examples

```
>>> from sympy import fps, sin, integrate
>>> from sympy.abc import x
>>> f = fps(sin(x))
>>> f.integrate(x).truncate()
-1 + x**2/2 - x**4/24 + O(x**6)
>>> integrate(f, (x, 0, 1))
1 - cos(1)
```

inverse(*x=None*, *n=6*)

Returns the truncated terms of the inverse of the formal power series, up to specified n.

Parameters

n : Number, optional

Specifies the order of the term up to which the polynomial should be truncated.

Explanation

If f and g are two formal power series of two different functions, then the coefficient sequence ak of the composed formal power series fp will be as follows.

$$\sum_{k=0}^{n}(-1)^k x_0^{-k-1} B_{n,k}(x_1, x_2, \ldots, x_{n-k+1})$$

Examples

```
>>> from sympy import fps, exp, cos
>>> from sympy.abc import x
>>> f1 = fps(exp(x))
>>> f2 = fps(cos(x))
```

```
>>> f1.inverse(x).truncate()
1 - x + x**2/2 - x**3/6 + x**4/24 - x**5/120 + O(x**6)
```

```
>>> f2.inverse(x).truncate(8)
1 + x**2/2 + 5*x**4/24 + 61*x**6/720 + O(x**8)
```

See also:

sympy.functions.combinatorial.numbers.bell (page 427), *sympy.series.formal.FormalPowerSeriesInverse* (page 653)

References

[R732]

polynomial(*n=6*)

Truncated series as polynomial.

Explanation

Returns series expansion of `f` upto order `O(x**n)` as a polynomial(without `O` term).

product(*other*, *x=None*, *n=6*)

Multiplies two Formal Power Series, using discrete convolution and return the truncated terms upto specified order.

Parameters

n : Number, optional

Specifies the order of the term up to which the polynomial should be truncated.

Examples

```
>>> from sympy import fps, sin, exp
>>> from sympy.abc import x
>>> f1 = fps(sin(x))
>>> f2 = fps(exp(x))
```

```
>>> f1.product(f2, x).truncate(4)
x + x**2 + x**3/3 + O(x**4)
```

See also:

sympy.discrete.convolutions (page 1089), *sympy.series.formal.FormalPowerSeriesProduct* (page 653)

truncate(*n=6*)

Truncated series.

Explanation

Returns truncated series expansion of f upto order `O(x**n)`.

If n is `None`, returns an infinite iterator.

`sympy.series.formal.`**fps**(*f*, *x=None*, *x0=0*, *dir=1*, *hyper=True*, *order=4*, *rational=True*, *full=False*)

Generates Formal Power Series of `f`.

Parameters

x : Symbol, optional

If x is None and `f` is univariate, the univariate symbols will be supplied, otherwise an error will be raised.

x0 : number, optional

Point to perform series expansion about. Default is 0.

dir : {1, -1, '+', '-'}, optional

If dir is 1 or '+' the series is calculated from the right and for -1 or '-' the series is calculated from the left. For smooth functions this flag will not alter the results. Default is 1.

hyper : {True, False}, optional

Set hyper to False to skip the hypergeometric algorithm. By default it is set to False.

order : int, optional

Order of the derivative of `f`, Default is 4.

rational : {True, False}, optional

Set rational to False to skip rational algorithm. By default it is set to True.

full : {True, False}, optional

Set full to True to increase the range of rational algorithm. See *rational_algorithm()* (page 654) for details. By default it is set to False.

Explanation

Returns the formal series expansion of `f` around `x = x0` with respect to `x` in the form of a `FormalPowerSeries` object.

Formal Power Series is represented using an explicit formula computed using different algorithms.

See *compute_fps()* (page 652) for the more details regarding the computation of formula.

Examples

```
>>> from sympy import fps, ln, atan, sin
>>> from sympy.abc import x, n
```

Rational Functions

```
>>> fps(ln(1 + x)).truncate()
x - x**2/2 + x**3/3 - x**4/4 + x**5/5 + O(x**6)
```

```
>>> fps(atan(x), full=True).truncate()
x - x**3/3 + x**5/5 + O(x**6)
```

Symbolic Functions

```
>>> fps(x**n*sin(x**2), x).truncate(8)
-x**(n + 6)/6 + x**(n + 2) + O(x**(n + 8))
```

See also:

sympy.series.formal.FormalPowerSeries (page 647), *sympy.series.formal.compute_fps* (page 652)

`sympy.series.formal.`**`compute_fps`**(*f, x, x0=0, dir=1, hyper=True, order=4, rational=True, full=False*)

Computes the formula for Formal Power Series of a function.

Parameters

x : Symbol

x0 : number, optional

Point to perform series expansion about. Default is 0.

dir : {1, -1, '+', '-'}, optional

If dir is 1 or '+' the series is calculated from the right and for -1 or '-' the series is calculated from the left. For smooth functions this flag will not alter the results. Default is 1.

hyper : {True, False}, optional

Set hyper to False to skip the hypergeometric algorithm. By default it is set to False.

order : int, optional

Order of the derivative of `f`, Default is 4.

rational : {True, False}, optional

Set rational to False to skip rational algorithm. By default it is set to True.

full : {True, False}, optional

Set full to True to increase the range of rational algorithm. See *rational_algorithm()* (page 654) for details. By default it is set to False.

Returns

ak : sequence

Sequence of coefficients.

xk : sequence

Sequence of powers of x.

ind : Expr

Independent terms.

mul : Pow

Common terms.

Explanation

Tries to compute the formula by applying the following techniques (in order):

- rational_algorithm
- Hypergeometric algorithm

See also:

sympy.series.formal.rational_algorithm (page 654), *sympy.series.formal.hyper_algorithm* (page 658)

class sympy.series.formal.**FormalPowerSeriesCompose**(**args*)

Represents the composed formal power series of two functions.

Explanation

No computation is performed. Terms are calculated using a term by term logic, instead of a point by point logic.

There are two differences between a *FormalPowerSeries* (page 647) object and a *FormalPowerSeriesCompose* (page 653) object. The first argument contains the outer function and the inner function involved in the omposition. Also, the coefficient sequence contains the generic sequence which is to be multiplied by a custom `bell_seq` finite sequence. The finite terms will then be added up to get the final terms.

See also:

sympy.series.formal.FormalPowerSeries (page 647), *sympy.series.formal.FiniteFormalPowerSeries* (page 654)

property function

Function for the composed formal power series.

class sympy.series.formal.**FormalPowerSeriesInverse**(**args*)

Represents the Inverse of a formal power series.

Explanation

No computation is performed. Terms are calculated using a term by term logic, instead of a point by point logic.

There is a single difference between a *FormalPowerSeries* (page 647) object and a *FormalPowerSeriesInverse* (page 653) object. The coefficient sequence contains the generic sequence which is to be multiplied by a custom `bell_seq` finite sequence. The finite terms will then be added up to get the final terms.

See also:

sympy.series.formal.FormalPowerSeries (page 647), *sympy.series.formal.FiniteFormalPowerSeries* (page 654)

property function

Function for the inverse of a formal power series.

class sympy.series.formal.**FormalPowerSeriesProduct**(**args*)

Represents the product of two formal power series of two functions.

Explanation

No computation is performed. Terms are calculated using a term by term logic, instead of a point by point logic.

There are two differences between a *FormalPowerSeries* (page 647) object and a *FormalPowerSeriesProduct* (page 653) object. The first argument contains the two functions involved in the product. Also, the coefficient sequence contains both the coefficient sequence of the formal power series of the involved functions.

See also:

sympy.series.formal.FormalPowerSeries (page 647), *sympy.series.formal.FiniteFormalPowerSeries* (page 654)

property function

Function of the product of two formal power series.

class sympy.series.formal.**FiniteFormalPowerSeries**(**args*)

Base Class for Product, Compose and Inverse classes

Rational Algorithm

sympy.series.formal.**rational_independent**(*terms*, *x*)

Returns a list of all the rationally independent terms.

Examples

```
>>> from sympy import sin, cos
>>> from sympy.series.formal import rational_independent
>>> from sympy.abc import x
```

```
>>> rational_independent([cos(x), sin(x)], x)
[cos(x), sin(x)]
>>> rational_independent([x**2, sin(x), x*sin(x), x**3], x)
[x**3 + x**2, x*sin(x) + sin(x)]
```

sympy.series.formal.**rational_algorithm**(*f*, *x*, *k*, *order=4*, *full=False*)

Rational algorithm for computing formula of coefficients of Formal Power Series of a function.

Parameters

x : Symbol

order : int, optional

Order of the derivative of f, Default is 4.

full : bool

Returns

formula : Expr

ind : Expr

Independent terms.

order : int

full : bool

Explanation

Applicable when f(x) or some derivative of f(x) is a rational function in x.

rational_algorithm() (page 654) uses *apart()* (page 2443) function for partial fraction decomposition. *apart()* (page 2443) by default uses 'undetermined coefficients method'. By setting full=True, 'Bronstein's algorithm' can be used instead.

Looks for derivative of a function up to 4'th order (by default). This can be overridden using order option.

Examples

```
>>> from sympy import log, atan
>>> from sympy.series.formal import rational_algorithm as ra
>>> from sympy.abc import x, k
```

```
>>> ra(1 / (1 - x), x, k)
(1, 0, 0)
>>> ra(log(1 + x), x, k)
(-1/((-1)**k*k), 0, 1)
```

```
>>> ra(atan(x), x, k, full=True)
((-I/(2*(-I)**k) + I/(2*I**k))/k, 0, 1)
```

Notes

By setting full=True, range of admissible functions to be solved using rational_algorithm can be increased. This option should be used carefully as it can significantly slow down the computation as doit is performed on the *RootSum* (page 2436) object returned by the *apart()* (page 2443) function. Use full=False whenever possible.

See also:

sympy.polys.partfrac.apart (page 2443)

References

[R733], [R734]

Hypergeometric Algorithm

sympy.series.formal.**simpleDE**(*f*, *x*, *g*, *order=4*)

Generates simple DE.

Explanation

DE is of the form

$$f^k(x) + \sum_{j=0}^{k-1} A_j f^j(x) = 0$$

where A_j should be rational function in x.

Generates DE's upto order 4 (default). DE's can also have free parameters.

By increasing order, higher order DE's can be found.

Yields a tuple of (DE, order).

sympy.series.formal.**exp_re**(*DE*, *r*, *k*)

Converts a DE with constant coefficients (explike) into a RE.

Explanation

Performs the substitution:

$$f^j(x) \to r(k + j)$$

Normalises the terms so that lowest order of a term is always r(k).

Examples

```
>>> from sympy import Function, Derivative
>>> from sympy.series.formal import exp_re
>>> from sympy.abc import x, k
>>> f, r = Function('f'), Function('r')
```

```
>>> exp_re(-f(x) + Derivative(f(x)), r, k)
-r(k) + r(k + 1)
>>> exp_re(Derivative(f(x), x) + Derivative(f(x), (x, 2)), r, k)
r(k) + r(k + 1)
```

See also:

sympy.series.formal.hyper_re (page 656)

sympy.series.formal.**hyper_re**(*DE*, *r*, *k*)

Converts a DE into a RE.

Explanation

Performs the substitution:

$$x^l f^j(x) \to (k + 1 - l)_j . a_{k+j-l}$$

Normalises the terms so that lowest order of a term is always r(k).

Examples

```
>>> from sympy import Function, Derivative
>>> from sympy.series.formal import hyper_re
>>> from sympy.abc import x, k
>>> f, r = Function('f'), Function('r')
```

```
>>> hyper_re(-f(x) + Derivative(f(x)), r, k)
(k + 1)*r(k + 1) - r(k)
>>> hyper_re(-x*f(x) + Derivative(f(x), (x, 2)), r, k)
(k + 2)*(k + 3)*r(k + 3) - r(k)
```

See also:

sympy.series.formal.exp_re (page 656)

sympy.series.formal.**rsolve_hypergeometric**(*f*, *x*, *P*, *Q*, *k*, *m*)

Solves RE of hypergeometric type.

Returns

formula : Expr

ind : Expr

Independent terms.

order : int

Explanation

Attempts to solve RE of the form

Q(k)*a(k + m) - P(k)*a(k)

Transformations that preserve Hypergeometric type:

a. x**n*f(x): b(k + m) = R(k - n)*b(k)
b. f(A*x): b(k + m) = A**m*R(k)*b(k)
c. f(x**n): b(k + n*m) = R(k/n)*b(k)
d. f(x**(1/m)): b(k + 1) = R(k*m)*b(k)
e. f'(x): b(k + m) = ((k + m + 1)/(k + 1))*R(k + 1)*b(k)

Some of these transformations have been used to solve the RE.

Examples

```
>>> from sympy import exp, ln, S
>>> from sympy.series.formal import rsolve_hypergeometric as rh
>>> from sympy.abc import x, k
```

```
>>> rh(exp(x), x, -S.One, (k + 1), k, 1)
(Piecewise((1/factorial(k), Eq(Mod(k, 1), 0)), (0, True)), 1, 1)
```

```
>>> rh(ln(1 + x), x, k**2, k*(k + 1), k, 1)
(Piecewise(((-1)**(k - 1)*factorial(k - 1)/RisingFactorial(2, k - 1),
 Eq(Mod(k, 1), 0)), (0, True)), x, 2)
```

References

[R735], [R736]

`sympy.series.formal.`**`solve_de`**(*f, x, DE, order, g, k*)

Solves the DE.

Returns

formula : Expr

ind : Expr

Independent terms.

order : int

Explanation

Tries to solve DE by either converting into a RE containing two terms or converting into a DE having constant coefficients.

Examples

```
>>> from sympy import Derivative as D, Function
>>> from sympy import exp, ln
>>> from sympy.series.formal import solve_de
>>> from sympy.abc import x, k
>>> f = Function('f')
```

```
>>> solve_de(exp(x), x, D(f(x), x) - f(x), 1, f, k)
(Piecewise((1/factorial(k), Eq(Mod(k, 1), 0)), (0, True)), 1, 1)
```

```
>>> solve_de(ln(1 + x), x, (x + 1)*D(f(x), x, 2) + D(f(x)), 2, f, k)
(Piecewise(((-1)**(k - 1)*factorial(k - 1)/RisingFactorial(2, k - 1),
 Eq(Mod(k, 1), 0)), (0, True)), x, 2)
```

`sympy.series.formal.`**`hyper_algorithm`**(*f, x, k, order=4*)

Hypergeometric algorithm for computing Formal Power Series.

Explanation

Steps:

- Generates DE
- Convert the DE into RE
- Solves the RE

Examples

```
>>> from sympy import exp, ln
>>> from sympy.series.formal import hyper_algorithm
```

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

```
>>> hyper_algorithm(exp(x), x, k)
(Piecewise((1/factorial(k), Eq(Mod(k, 1), 0)), (0, True)), 1, 1)
```

```
>>> hyper_algorithm(ln(1 + x), x, k)
(Piecewise(((-1)**(k - 1)*factorial(k - 1)/RisingFactorial(2, k - 1),
 Eq(Mod(k, 1), 0)), (0, True)), x, 2)
```

See also:

sympy.series.formal.simpleDE (page 656), *sympy.series.formal.solve_de* (page 658)

Limits of Sequences

Provides methods to compute limit of terms having sequences at infinity.

sympy.series.limitseq.**difference_delta**(*expr*, *n=None*, *step=1*)

Difference Operator.

Explanation

Discrete analog of differential operator. Given a sequence x[n], returns the sequence x[n + step] - x[n].

Examples

```
>>> from sympy import difference_delta as dd
>>> from sympy.abc import n
>>> dd(n*(n + 1), n)
2*n + 2
>>> dd(n*(n + 1), n, 2)
4*n + 6
```

References

[R740]

`sympy.series.limitseq.`**`dominant`**`(`*expr*, *n*`)`

Finds the dominant term in a sum, that is a term that dominates every other term.

Explanation

If limit(a/b, n, oo) is oo then a dominates b. If limit(a/b, n, oo) is 0 then b dominates a. Otherwise, a and b are comparable.

If there is no unique dominant term, then returns `None`.

Examples

```
>>> from sympy import Sum
>>> from sympy.series.limitseq import dominant
>>> from sympy.abc import n, k
>>> dominant(5*n**3 + 4*n**2 + n + 1, n)
5*n**3
>>> dominant(2**n + Sum(k, (k, 0, n)), n)
2**n
```

See also:

`sympy.series.limitseq.dominant` (page 660)

`sympy.series.limitseq.`**`limit_seq`**`(`*expr*, *n=None*, *trials=5*`)`

Finds the limit of a sequence as index `n` tends to infinity.

Parameters

expr : Expr

SymPy expression for the `n-th` term of the sequence

n : Symbol, optional

The index of the sequence, an integer that tends to positive infinity. If None, inferred from the expression unless it has multiple symbols.

trials: int, optional

The algorithm is highly recursive. `trials` is a safeguard from infinite recursion in case the limit is not easily computed by the algorithm. Try increasing `trials` if the algorithm returns `None`.

Admissible Terms

The algorithm is designed for sequences built from rational functions, indefinite sums, and indefinite products over an indeterminate n. Terms of alternating sign are also allowed, but more complex oscillatory behavior is not supported.

Examples

```
>>> from sympy import limit_seq, Sum, binomial
>>> from sympy.abc import n, k, m
>>> limit_seq((5*n**3 + 3*n**2 + 4) / (3*n**3 + 4*n - 5), n)
5/3
>>> limit_seq(binomial(2*n, n) / Sum(binomial(2*k, k), (k, 1, n)), n)
3/4
>>> limit_seq(Sum(k**2 * Sum(2**m/m, (m, 1, k)), (k, 1, n)) / (2**n*n),␣
↪n)
4
```

See also:

sympy.series.limitseq.dominant (page 660)

References

[R741]

Simplify

Simplify

simplify

sympy.simplify.simplify.**simplify**(*expr, ratio=1.7, measure=<function count_ops>, rational=False, inverse=False, doit=True, **kwargs*)

Simplifies the given expression.

Explanation

Simplification is not a well defined term and the exact strategies this function tries can change in the future versions of SymPy. If your algorithm relies on “simplification” (whatever it is), try to determine what you need exactly - is it powsimp()?, radsimp()?, together()?, logcombine()?, or something else? And use this particular function directly, because those are well defined and thus your algorithm will be robust.

Nonetheless, especially for interactive use, or when you do not know anything about the structure of the expression, simplify() tries to apply intelligent heuristics to make the input expression “simpler”. For example:

```
>>> from sympy import simplify, cos, sin
>>> from sympy.abc import x, y
>>> a = (x + x**2)/(x*sin(y)**2 + x*cos(y)**2)
>>> a
(x**2 + x)/(x*sin(y)**2 + x*cos(y)**2)
>>> simplify(a)
x + 1
```

Note that we could have obtained the same result by using specific simplification functions:

```
>>> from sympy import trigsimp, cancel
>>> trigsimp(a)
(x**2 + x)/x
>>> cancel(_)
x + 1
```

In some cases, applying *simplify()* (page 661) may actually result in some more complicated expression. The default `ratio=1.7` prevents more extreme cases: if (result length)/(input length) > ratio, then input is returned unmodified. The `measure` parameter lets you specify the function used to determine how complex an expression is. The function should take a single argument as an expression and return a number such that if expression `a` is more complex than expression `b`, then `measure(a) > measure(b)`. The default measure function is *count_ops()* (page 1059), which returns the total number of operations in the expression.

For example, if `ratio=1`, `simplify` output cannot be longer than input.

```
>>> from sympy import sqrt, simplify, count_ops, oo
>>> root = 1/(sqrt(2)+3)
```

Since `simplify(root)` would result in a slightly longer expression, root is returned unchanged instead:

```
>>> simplify(root, ratio=1) == root
True
```

If `ratio=oo`, simplify will be applied anyway:

```
>>> count_ops(simplify(root, ratio=oo)) > count_ops(root)
True
```

Note that the shortest expression is not necessary the simplest, so setting `ratio` to 1 may not be a good idea. Heuristically, the default value `ratio=1.7` seems like a reasonable choice.

You can easily define your own measure function based on what you feel should represent the "size" or "complexity" of the input expression. Note that some choices, such as `lambda expr: len(str(expr))` may appear to be good metrics, but have other problems (in this case, the measure function may slow down simplify too much for very large expressions). If you do not know what a good metric would be, the default, `count_ops`, is a good one.

For example:

```
>>> from sympy import symbols, log
>>> a, b = symbols('a b', positive=True)
>>> g = log(a) + log(b) + log(a)*log(1/b)
>>> h = simplify(g)
>>> h
log(a*b**(1 - log(a)))
>>> count_ops(g)
8
>>> count_ops(h)
5
```

So you can see that h is simpler than g using the count_ops metric. However, we may not like how `simplify` (in this case, using `logcombine`) has created the `b**(log(1/a) + 1)` term. A simple way to reduce this would be to give more weight to powers as operations in `count_ops`. We can do this by using the `visual=True` option:

```
>>> print(count_ops(g, visual=True))
2*ADD + DIV + 4*LOG + MUL
>>> print(count_ops(h, visual=True))
2*LOG + MUL + POW + SUB
```

```
>>> from sympy import Symbol, S
>>> def my_measure(expr):
...     POW = Symbol('POW')
...     # Discourage powers by giving POW a weight of 10
...     count = count_ops(expr, visual=True).subs(POW, 10)
...     # Every other operation gets a weight of 1 (the default)
...     count = count.replace(Symbol, type(S.One))
...     return count
>>> my_measure(g)
8
>>> my_measure(h)
14
>>> 15./8 > 1.7 # 1.7 is the default ratio
True
>>> simplify(g, measure=my_measure)
-log(a)*log(b) + log(a) + log(b)
```

Note that because `simplify()` internally tries many different simplification strategies and then compares them using the measure function, we get a completely different result that is still different from the input expression by doing this.

If `rational=True`, Floats will be recast as Rationals before simplification. If `rational=None`, Floats will be recast as Rationals but the result will be recast as Floats. If rational=False(default) then nothing will be done to the Floats.

If `inverse=True`, it will be assumed that a composition of inverse functions, such as sin and asin, can be cancelled in any order. For example, `asin(sin(x))` will yield x without checking whether x belongs to the set where this relation is true. The default is False.

Note that `simplify()` automatically calls `doit()` on the final expression. You can avoid this behavior by passing `doit=False` as an argument.

Also, it should be noted that simplifying the boolian expression is not well defined. If the expression prefers automatic evaluation (such as *Eq()* (page 1022) or *Or()* (page 1167)), simplification will return True or False if truth value can be determined. If the expression is not evaluated by default (such as *Predicate()* (page 195)), simplification will not reduce it and you should use *refine()* (page 197) or *ask()* (page 191) function. This inconsistency will be resolved in future version.

See also:

sympy.assumptions.refine.refine **(page 197)**
Simplification using assumptions.

sympy.assumptions.ask.ask **(page 191)**
Query for boolean expressions using assumptions.

separatevars

sympy.simplify.simplify.**separatevars**(*expr*, *symbols=[]*, *dict=False*, *force=False*)

Separates variables in an expression, if possible. By default, it separates with respect to all symbols in an expression and collects constant coefficients that are independent of symbols.

Explanation

If dict=True then the separated terms will be returned in a dictionary keyed to their corresponding symbols. By default, all symbols in the expression will appear as keys; if symbols are provided, then all those symbols will be used as keys, and any terms in the expression containing other symbols or non-symbols will be returned keyed to the string 'coeff'. (Passing None for symbols will return the expression in a dictionary keyed to 'coeff'.)

If force=True, then bases of powers will be separated regardless of assumptions on the symbols involved.

Notes

The order of the factors is determined by Mul, so that the separated expressions may not necessarily be grouped together.

Although factoring is necessary to separate variables in some expressions, it is not necessary in all cases, so one should not count on the returned factors being factored.

Examples

```
>>> from sympy.abc import x, y, z, alpha
>>> from sympy import separatevars, sin
>>> separatevars((x*y)**y)
(x*y)**y
>>> separatevars((x*y)**y, force=True)
x**y*y**y
```

```
>>> e = 2*x**2*z*sin(y)+2*z*x**2
>>> separatevars(e)
2*x**2*z*(sin(y) + 1)
>>> separatevars(e, symbols=(x, y), dict=True)
{'coeff': 2*z, x: x**2, y: sin(y) + 1}
>>> separatevars(e, [x, y, alpha], dict=True)
{'coeff': 2*z, alpha: 1, x: x**2, y: sin(y) + 1}
```

If the expression is not really separable, or is only partially separable, separatevars will do the best it can to separate it by using factoring.

```
>>> separatevars(x + x*y - 3*x**2)
-x*(3*x - y - 1)
```

If the expression is not separable then expr is returned unchanged or (if dict=True) then None is returned.

```
>>> eq = 2*x + y*sin(x)
>>> separatevars(eq) == eq
True
>>> separatevars(2*x + y*sin(x), symbols=(x, y), dict=True) is None
True
```

nthroot

sympy.simplify.simplify.**nthroot**(*expr*, *n*, *max_len=4*, *prec=15*)

Compute a real nth-root of a sum of surds.

Parameters

expr : sum of surds

n : integer

max_len : maximum number of surds passed as constants to `nsimplify`

Algorithm

First `nsimplify` is used to get a candidate root; if it is not a root the minimal polynomial is computed; the answer is one of its roots.

Examples

```
>>> from sympy.simplify.simplify import nthroot
>>> from sympy import sqrt
>>> nthroot(90 + 34*sqrt(7), 3)
sqrt(7) + 3
```

kroneckersimp

`sympy.simplify.simplify.`**`kroneckersimp`**(*expr*)

Simplify expressions with KroneckerDelta.

The only simplification currently attempted is to identify multiplicative cancellation:

Examples

```
>>> from sympy import KroneckerDelta, kroneckersimp
>>> from sympy.abc import i
>>> kroneckersimp(1 + KroneckerDelta(0, i) * KroneckerDelta(1, i))
1
```

besselsimp

`sympy.simplify.simplify.`**`besselsimp`**(*expr*)

Simplify bessel-type functions.

Explanation

This routine tries to simplify bessel-type functions. Currently it only works on the Bessel J and I functions, however. It works by looking at all such functions in turn, and eliminating factors of "I" and "-1" (actually their polar equivalents) in front of the argument. Then, functions of half-integer order are rewritten using strigonometric functions and functions of integer order (> 1) are rewritten using functions of low order. Finally, if the expression was changed, compute factorization of the result with factor().

```
>>> from sympy import besselj, besseli, besselsimp, polar_lift, I, S
>>> from sympy.abc import z, nu
>>> besselsimp(besselj(nu, z*polar_lift(-1)))
exp(I*pi*nu)*besselj(nu, z)
>>> besselsimp(besseli(nu, z*polar_lift(-I)))
exp(-I*pi*nu/2)*besselj(nu, z)
>>> besselsimp(besseli(S(-1)/2, z))
sqrt(2)*cosh(z)/(sqrt(pi)*sqrt(z))
>>> besselsimp(z*besseli(0, z) + z*(besseli(2, z))/2 + besseli(1, z))
3*z*besseli(0, z)/2
```

hypersimp

`sympy.simplify.simplify.`**`hypersimp`**(*f*, *k*)

Given combinatorial term f(k) simplify its consecutive term ratio i.e. f(k+1)/f(k). The input term can be composed of functions and integer sequences which have equivalent representation in terms of gamma special function.

Explanation

The algorithm performs three basic steps:

1. Rewrite all functions in terms of gamma, if possible.
2. Rewrite all occurrences of gamma in terms of products of gamma and rising factorial with integer, absolute constant exponent.
3. Perform simplification of nested fractions, powers and if the resulting expression is a quotient of polynomials, reduce their total degree.

If f(k) is hypergeometric then as result we arrive with a quotient of polynomials of minimal degree. Otherwise None is returned.

For more information on the implemented algorithm refer to:

1. W. Koepf, Algorithms for m-fold Hypergeometric Summation, Journal of Symbolic Computation (1995) 20, 399-417

hypersimilar

`sympy.simplify.simplify.`**`hypersimilar`**(*f*, *g*, *k*)

Returns True if `f` and `g` are hyper-similar.

Explanation

Similarity in hypergeometric sense means that a quotient of f(k) and g(k) is a rational function in `k`. This procedure is useful in solving recurrence relations.

For more information see hypersimp().

nsimplify

`sympy.simplify.simplify.`**`nsimplify`**(*expr*, *constants=()*, *tolerance=None*, *full=False*, *rational=None*, *rational_conversion='base10'*)

Find a simple representation for a number or, if there are free symbols or if `rational=True`, then replace Floats with their Rational equivalents. If no change is made and rational is not False then Floats will at least be converted to Rationals.

Explanation

For numerical expressions, a simple formula that numerically matches the given numerical expression is sought (and the input should be possible to evalf to a precision of at least 30 digits).

Optionally, a list of (rationally independent) constants to include in the formula may be given.

A lower tolerance may be set to find less exact matches. If no tolerance is given then the least precise value will set the tolerance (e.g. Floats default to 15 digits of precision, so would be tolerance=10**-15).

With `full=True`, a more extensive search is performed (this is useful to find simpler numbers when the tolerance is set low).

When converting to rational, if rational_conversion='base10' (the default), then convert floats to rationals using their base-10 (string) representation. When rational_conversion='exact' it uses the exact, base-2 representation.

Examples

```
>>> from sympy import nsimplify, sqrt, GoldenRatio, exp, I, pi
>>> nsimplify(4/(1+sqrt(5)), [GoldenRatio])
-2 + 2*GoldenRatio
>>> nsimplify((1/(exp(3*pi*I/5)+1)))
1/2 - I*sqrt(sqrt(5)/10 + 1/4)
>>> nsimplify(I**I, [pi])
exp(-pi/2)
>>> nsimplify(pi, tolerance=0.01)
22/7
```

```
>>> nsimplify(0.333333333333333, rational=True, rational_conversion=
↪'exact')
6004799503160655/18014398509481984
>>> nsimplify(0.333333333333333, rational=True)
1/3
```

See also:

sympy.core.function.nfloat (page 1064)

posify

sympy.simplify.simplify.**posify**(*eq*)

Return eq (with generic symbols made positive) and a dictionary containing the mapping between the old and new symbols.

Explanation

Any symbol that has positive=None will be replaced with a positive dummy symbol having the same name. This replacement will allow more symbolic processing of expressions, especially those involving powers and logarithms.

A dictionary that can be sent to subs to restore eq to its original symbols is also returned.

```
>>> from sympy import posify, Symbol, log, solve
>>> from sympy.abc import x
>>> posify(x + Symbol('p', positive=True) + Symbol('n', negative=True))
(_x + n + p, {_x: x})
```

```
>>> eq = 1/x
>>> log(eq).expand()
log(1/x)
>>> log(posify(eq)[0]).expand()
-log(_x)
>>> p, rep = posify(eq)
>>> log(p).expand().subs(rep)
-log(x)
```

It is possible to apply the same transformations to an iterable of expressions:

```
>>> eq = x**2 - 4
>>> solve(eq, x)
[-2, 2]
>>> eq_x, reps = posify([eq, x]); eq_x
[_x**2 - 4, _x]
>>> solve(*eq_x)
[2]
```

logcombine

`sympy.simplify.simplify.`**`logcombine`**`(`*expr, force=False*`)`

Takes logarithms and combines them using the following rules:

- log(x) + log(y) == log(x*y) if both are positive
- a*log(x) == log(x**a) if x is positive and a is real

If `force` is `True` then the assumptions above will be assumed to hold if there is no assumption already in place on a quantity. For example, if `a` is imaginary or the argument negative, force will not perform a combination but if `a` is a symbol with no assumptions the change will take place.

Examples

```
>>> from sympy import Symbol, symbols, log, logcombine, I
>>> from sympy.abc import a, x, y, z
>>> logcombine(a*log(x) + log(y) - log(z))
a*log(x) + log(y) - log(z)
>>> logcombine(a*log(x) + log(y) - log(z), force=True)
log(x**a*y/z)
>>> x,y,z = symbols('x,y,z', positive=True)
>>> a = Symbol('a', real=True)
>>> logcombine(a*log(x) + log(y) - log(z))
log(x**a*y/z)
```

The transformation is limited to factors and/or terms that contain logs, so the result depends on the initial state of expansion:

```
>>> eq = (2 + 3*I)*log(x)
>>> logcombine(eq, force=True) == eq
True
>>> logcombine(eq.expand(), force=True)
log(x**2) + I*log(x**3)
```

See also:

posify **(page 668)**
replace all symbols with symbols having positive assumptions

sympy.core.function.expand_log **(page 1061)**
expand the logarithms of products and powers; the opposite of logcombine

Radsimp

radsimp

sympy.simplify.radsimp.**radsimp**(*expr, symbolic=True, max_terms=4*)
Rationalize the denominator by removing square roots.

Explanation

The expression returned from radsimp must be used with caution since if the denominator contains symbols, it will be possible to make substitutions that violate the assumptions of the simplification process: that for a denominator matching a + b*sqrt(c), a != +/-b*sqrt(c). (If there are no symbols, this assumptions is made valid by collecting terms of sqrt(c) so the match variable a does not contain sqrt(c).) If you do not want the simplification to occur for symbolic denominators, set symbolic to False.

If there are more than max_terms radical terms then the expression is returned unchanged.

Examples

```
>>> from sympy import radsimp, sqrt, Symbol, pprint
>>> from sympy import factor_terms, fraction, signsimp
>>> from sympy.simplify.radsimp import collect_sqrt
>>> from sympy.abc import a, b, c
```

```
>>> radsimp(1/(2 + sqrt(2)))
(2 - sqrt(2))/2
>>> x,y = map(Symbol, 'xy')
>>> e = ((2 + 2*sqrt(2))*x + (2 + sqrt(8))*y)/(2 + sqrt(2))
>>> radsimp(e)
sqrt(2)*(x + y)
```

No simplification beyond removal of the gcd is done. One might want to polish the result a little, however, by collecting square root terms:

```
>>> r2 = sqrt(2)
>>> r5 = sqrt(5)
>>> ans = radsimp(1/(y*r2 + x*r2 + a*r5 + b*r5)); pprint(ans)
    ___       ___       ___       ___
  \/ 5 *a + \/ 5 *b - \/ 2 *x - \/ 2 *y
------------------------------------------
   2               2      2              2
5*a  + 10*a*b + 5*b  - 2*x  - 4*x*y - 2*y
```

```
>>> n, d = fraction(ans)
>>> pprint(factor_terms(signsimp(collect_sqrt(n))/d, radical=True))
        ___             ___
      \/ 5 *(a + b) - \/ 2 *(x + y)
------------------------------------------
   2               2      2              2
5*a  + 10*a*b + 5*b  - 2*x  - 4*x*y - 2*y
```

If radicals in the denominator cannot be removed or there is no denominator, the original expression will be returned.

```
>>> radsimp(sqrt(2)*x + sqrt(2))
sqrt(2)*x + sqrt(2)
```

Results with symbols will not always be valid for all substitutions:

```
>>> eq = 1/(a + b*sqrt(c))
>>> eq.subs(a, b*sqrt(c))
1/(2*b*sqrt(c))
>>> radsimp(eq).subs(a, b*sqrt(c))
nan
```

If `symbolic=False`, symbolic denominators will not be transformed (but numeric denominators will still be processed):

```
>>> radsimp(eq, symbolic=False)
1/(a + b*sqrt(c))
```

rad_rationalize

`sympy.simplify.radsimp.`**`rad_rationalize`**`(`*num, den*`)`

Rationalize `num/den` by removing square roots in the denominator; num and den are sum of terms whose squares are positive rationals.

Examples

```
>>> from sympy import sqrt
>>> from sympy.simplify.radsimp import rad_rationalize
>>> rad_rationalize(sqrt(3), 1 + sqrt(2)/3)
(-sqrt(3) + sqrt(6)/3, -7/9)
```

collect

`sympy.simplify.radsimp.`**`collect`**`(`*expr, syms, func=None, evaluate=None, exact=False, distribute_order_term=True*`)`

Collect additive terms of an expression.

Explanation

This function collects additive terms of an expression with respect to a list of expression up to powers with rational exponents. By the term symbol here are meant arbitrary expressions, which can contain powers, products, sums etc. In other words symbol is a pattern which will be searched for in the expression's terms.

The input expression is not expanded by *`collect()`* (page 672), so user is expected to provide an expression in an appropriate form. This makes *`collect()`* (page 672) more predictable as there is no magic happening behind the scenes. However, it is important to note, that powers of products are converted to products of powers using the *`expand_power_base()`* (page 1063) function.

There are two possible types of output. First, if `evaluate` flag is set, this function will return an expression with collected terms or else it will return a dictionary with expressions up to rational powers as keys and collected coefficients as values.

Examples

```
>>> from sympy import S, collect, expand, factor, Wild
>>> from sympy.abc import a, b, c, x, y
```

This function can collect symbolic coefficients in polynomials or rational expressions. It will manage to find all integer or rational powers of collection variable:

```
>>> collect(a*x**2 + b*x**2 + a*x - b*x + c, x)
c + x**2*(a + b) + x*(a - b)
```

The same result can be achieved in dictionary form:

```
>>> d = collect(a*x**2 + b*x**2 + a*x - b*x + c, x, evaluate=False)
>>> d[x**2]
a + b
>>> d[x]
a - b
>>> d[S.One]
c
```

You can also work with multivariate polynomials. However, remember that this function is greedy so it will care only about a single symbol at time, in specification order:

```
>>> collect(x**2 + y*x**2 + x*y + y + a*y, [x, y])
x**2*(y + 1) + x*y + y*(a + 1)
```

Also more complicated expressions can be used as patterns:

```
>>> from sympy import sin, log
>>> collect(a*sin(2*x) + b*sin(2*x), sin(2*x))
(a + b)*sin(2*x)

>>> collect(a*x*log(x) + b*(x*log(x)), x*log(x))
x*(a + b)*log(x)
```

You can use wildcards in the pattern:

```
>>> w = Wild('w1')
>>> collect(a*x**y - b*x**y, w**y)
x**y*(a - b)
```

It is also possible to work with symbolic powers, although it has more complicated behavior, because in this case power's base and symbolic part of the exponent are treated as a single symbol:

```
>>> collect(a*x**c + b*x**c, x)
a*x**c + b*x**c
>>> collect(a*x**c + b*x**c, x**c)
x**c*(a + b)
```

However if you incorporate rationals to the exponents, then you will get well known behavior:

```
>>> collect(a*x**(2*c) + b*x**(2*c), x**c)
x**(2*c)*(a + b)
```

Note also that all previously stated facts about *collect()* (page 672) function apply to the exponential function, so you can get:

```
>>> from sympy import exp
>>> collect(a*exp(2*x) + b*exp(2*x), exp(x))
(a + b)*exp(2*x)
```

If you are interested only in collecting specific powers of some symbols then set exact flag to True:

```
>>> collect(a*x**7 + b*x**7, x, exact=True)
a*x**7 + b*x**7
>>> collect(a*x**7 + b*x**7, x**7, exact=True)
x**7*(a + b)
```

If you want to collect on any object containing symbols, set exact to None:

```
>>> collect(x*exp(x) + sin(x)*y + sin(x)*2 + 3*x, x, exact=None)
x*exp(x) + 3*x + (y + 2)*sin(x)
>>> collect(a*x*y + x*y + b*x + x, [x, y], exact=None)
x*y*(a + 1) + x*(b + 1)
```

You can also apply this function to differential equations, where derivatives of arbitrary order can be collected. Note that if you collect with respect to a function or a derivative of a function, all derivatives of that function will also be collected. Use `exact=True` to prevent this from happening:

```
>>> from sympy import Derivative as D, collect, Function
>>> f = Function('f') (x)

>>> collect(a*D(f,x) + b*D(f,x), D(f,x))
(a + b)*Derivative(f(x), x)

>>> collect(a*D(D(f,x),x) + b*D(D(f,x),x), f)
(a + b)*Derivative(f(x), (x, 2))

>>> collect(a*D(D(f,x),x) + b*D(D(f,x),x), D(f,x), exact=True)
a*Derivative(f(x), (x, 2)) + b*Derivative(f(x), (x, 2))

>>> collect(a*D(f,x) + b*D(f,x) + a*f + b*f, f)
(a + b)*f(x) + (a + b)*Derivative(f(x), x)
```

Or you can even match both derivative order and exponent at the same time:

```
>>> collect(a*D(D(f,x),x)**2 + b*D(D(f,x),x)**2, D(f,x))
(a + b)*Derivative(f(x), (x, 2))**2
```

Finally, you can apply a function to each of the collected coefficients. For example you can factorize symbolic coefficients of polynomial:

```
>>> f = expand((x + a + 1)**3)

>>> collect(f, x, factor)
x**3 + 3*x**2*(a + 1) + 3*x*(a + 1)**2 + (a + 1)**3
```

Note: Arguments are expected to be in expanded form, so you might have to call *expand()* (page 1053) prior to calling this function.

See also:

collect_const (page 676), *collect_sqrt* (page 675), *rcollect* (page 674)

`sympy.simplify.radsimp.`**`rcollect`**`(`*expr*, **vars*`)`

Recursively collect sums in an expression.

Examples

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

```
>>> expr = (x**2*y + x*y + x + y)/(x + y)
```

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

See also:

collect (page 672), *collect_const* (page 676), *collect_sqrt* (page 675)

collect_sqrt

sympy.simplify.radsimp.**collect_sqrt**(*expr*, *evaluate=None*)

Return expr with terms having common square roots collected together. If `evaluate` is False a count indicating the number of sqrt-containing terms will be returned and, if non-zero, the terms of the Add will be returned, else the expression itself will be returned as a single term. If `evaluate` is True, the expression with any collected terms will be returned.

Note: since I = sqrt(-1), it is collected, too.

Examples

```
>>> from sympy import sqrt
>>> from sympy.simplify.radsimp import collect_sqrt
>>> from sympy.abc import a, b
```

```
>>> r2, r3, r5 = [sqrt(i) for i in [2, 3, 5]]
>>> collect_sqrt(a*r2 + b*r2)
sqrt(2)*(a + b)
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r3)
sqrt(2)*(a + b) + sqrt(3)*(a + b)
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r5)
sqrt(3)*a + sqrt(5)*b + sqrt(2)*(a + b)
```

If evaluate is False then the arguments will be sorted and returned as a list and a count of the number of sqrt-containing terms will be returned:

```
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r5, evaluate=False)
((sqrt(3)*a, sqrt(5)*b, sqrt(2)*(a + b)), 3)
>>> collect_sqrt(a*sqrt(2) + b, evaluate=False)
((b, sqrt(2)*a), 1)
>>> collect_sqrt(a + b, evaluate=False)
((a + b,), 0)
```

See also:

collect (page 672), *collect_const* (page 676), *rcollect* (page 674)

collect_const

`sympy.simplify.radsimp.collect_const(expr, *vars, Numbers=True)`

A non-greedy collection of terms with similar number coefficients in an Add expr. If `vars` is given then only those constants will be targeted. Although any Number can also be targeted, if this is not desired set `Numbers=False` and no Float or Rational will be collected.

Parameters

expr : SymPy expression

This parameter defines the expression the expression from which terms with similar coefficients are to be collected. A non-Add expression is returned as it is.

vars : variable length collection of Numbers, optional

Specifies the constants to target for collection. Can be multiple in number.

Numbers : bool

Specifies to target all instance of *sympy.core.numbers.Number* (page 981) class. If `Numbers=False`, then no Float or Rational will be collected.

Returns

expr : Expr

Returns an expression with similar coefficient terms collected.

Examples

```
>>> from sympy import sqrt
>>> from sympy.abc import s, x, y, z
>>> from sympy.simplify.radsimp import collect_const
>>> collect_const(sqrt(3) + sqrt(3)*(1 + sqrt(2)))
sqrt(3)*(sqrt(2) + 2)
>>> collect_const(sqrt(3)*s + sqrt(7)*s + sqrt(3) + sqrt(7))
(sqrt(3) + sqrt(7))*(s + 1)
>>> s = sqrt(2) + 2
>>> collect_const(sqrt(3)*s + sqrt(3) + sqrt(7)*s + sqrt(7))
(sqrt(2) + 3)*(sqrt(3) + sqrt(7))
>>> collect_const(sqrt(3)*s + sqrt(3) + sqrt(7)*s + sqrt(7), sqrt(3))
sqrt(7) + sqrt(3)*(sqrt(2) + 3) + sqrt(7)*(sqrt(2) + 2)
```

The collection is sign-sensitive, giving higher precedence to the unsigned values:

```
>>> collect_const(x - y - z)
x - (y + z)
>>> collect_const(-y - z)
-(y + z)
>>> collect_const(2*x - 2*y - 2*z, 2)
2*(x - y - z)
>>> collect_const(2*x - 2*y - 2*z, -2)
2*x - 2*(y + z)
```