`sympy.integrals.heurisch.`**`heurisch`**(*f*, *x*, *rewrite=False*, *hints=None*, *mappings=None*, *retries=3*, *degree_offset=0*, *unnecessary_permutations=None*, *_try_heurisch=None*)

Compute indefinite integral using heuristic Risch algorithm.

Explanation

This is a heuristic approach to indefinite integration in finite terms using the extended heuristic (parallel) Risch algorithm, based on Manuel Bronstein's "Poor Man's Integrator".

The algorithm supports various classes of functions including transcendental elementary or special functions like Airy, Bessel, Whittaker and Lambert.

Note that this algorithm is not a decision procedure. If it isn't able to compute the antiderivative for a given function, then this is not a proof that such a functions does not exist. One should use recursive Risch algorithm in such case. It's an open question if this algorithm can be made a full decision procedure.

This is an internal integrator procedure. You should use top level 'integrate' function in most cases, as this procedure needs some preprocessing steps and otherwise may fail.

Specification

heurisch(f, x, rewrite=False, hints=None)

where

f : expression x : symbol

rewrite -> force rewrite 'f' in terms of 'tan' and 'tanh' hints -> a list of functions that may appear in anti-derivate

- hints = None –> no suggestions at all
- hints = [ ] –> try to figure out
- hints = [f1, …, fn] –> we know better

Examples

```
>>> from sympy import tan
>>> from sympy.integrals.heurisch import heurisch
>>> from sympy.abc import x, y
```

```
>>> heurisch(y*tan(x), x)
y*log(tan(x)**2 + 1)/2
```

See Manuel Bronstein's "Poor Man's Integrator":

See also:

sympy.integrals.integrals.Integral.doit (page 603), *sympy.integrals.integrals.Integral* (page 601), *sympy.integrals.heurisch.components* (page 598)

References

For more information on the implemented algorithm refer to:

[R527], [R528], [R529], [R530], [R531]

sympy.integrals.heurisch.**components**(*f*, *x*)

Returns a set of all functional components of the given expression which includes symbols, function applications and compositions and non-integer powers. Fractional powers are collected with minimal, positive exponents.

Examples

```
>>> from sympy import cos, sin
>>> from sympy.abc import x
>>> from sympy.integrals.heurisch import components
```

```
>>> components(sin(x)*cos(x)**2, x)
{x, sin(x), cos(x)}
```

See also:

heurisch (page 596)

API reference

sympy.integrals.integrals.**integrate**(*f*, *var*, *...*)

Deprecated since version 1.6: Using `integrate()` with *Poly* (page 2378) is deprecated. Use *Poly.integrate()* (page 2398) instead. See *Using integrate with Poly* (page 176).

Explanation

Compute definite or indefinite integral of one or more variables using Risch-Norman algorithm and table lookup. This procedure is able to handle elementary algebraic and transcendental functions and also a huge class of special functions, including Airy, Bessel, Whittaker and Lambert.

var can be:

- a symbol – indefinite integration
- **a tuple (symbol, a) – indefinite integration with result**
  given with a replacing `symbol`
- a tuple (symbol, a, b) – definite integration

Several variables can be specified, in which case the result is multiple integration. (If var is omitted and the integrand is univariate, the indefinite integral in that variable will be performed.)

Indefinite integrals are returned without terms that are independent of the integration variables. (see examples)

Definite improper integrals often entail delicate convergence conditions. Pass conds='piecewise', 'separate' or 'none' to have these returned, respectively, as a Piecewise function, as a separate result (i.e. result will be a tuple), or not at all (default is 'piecewise').

Strategy

SymPy uses various approaches to definite integration. One method is to find an antiderivative for the integrand, and then use the fundamental theorem of calculus. Various functions are implemented to integrate polynomial, rational and trigonometric functions, and integrands containing DiracDelta terms.

SymPy also implements the part of the Risch algorithm, which is a decision procedure for integrating elementary functions, i.e., the algorithm can either find an elementary antiderivative, or prove that one does not exist. There is also a (very successful, albeit somewhat slow) general implementation of the heuristic Risch algorithm. This algorithm will eventually be phased out as more of the full Risch algorithm is implemented. See the docstring of Integral._eval_integral() for more details on computing the antiderivative using algebraic methods.

The option risch=True can be used to use only the (full) Risch algorithm. This is useful if you want to know if an elementary function has an elementary antiderivative. If the indefinite Integral returned by this function is an instance of NonElementaryIntegral, that means that the Risch algorithm has proven that integral to be non-elementary. Note that by default, additional methods (such as the Meijer G method outlined below) are tried on these integrals, as they may be expressible in terms of special functions, so if you only care about elementary answers, use risch=True. Also note that an unevaluated Integral returned by this function is not necessarily a NonElementaryIntegral, even with risch=True, as it may just be an indication that the particular part of the Risch algorithm needed to integrate that function is not yet implemented.

Another family of strategies comes from re-writing the integrand in terms of so-called Meijer G-functions. Indefinite integrals of a single G-function can always be computed, and the definite integral of a product of two G-functions can be computed from zero to infinity. Various strategies are implemented to rewrite integrands as G-functions, and use this information to compute integrals (see the `meijerint` module).

The option manual=True can be used to use only an algorithm that tries to mimic integration by hand. This algorithm does not handle as many integrands as the other algorithms implemented but may return results in a more familiar form. The `manualintegrate` module has functions that return the steps used (see the module docstring for more information).

In general, the algebraic methods work best for computing antiderivatives of (possibly complicated) combinations of elementary functions. The G-function methods work best for computing definite integrals from zero to infinity of moderately complicated combinations of special functions, or indefinite integrals of very simple combinations of special functions.

The strategy employed by the integration code is as follows:

- If computing a definite integral, and both limits are real, and at least one limit is +- oo, try the G-function method of definite integration first.
- Try to find an antiderivative, using all available methods, ordered by performance (that is try fastest method first, slowest last; in particular polynomial integration is tried first, Meijer G-functions second to last, and heuristic Risch last).
- If still not successful, try G-functions irrespective of the limits.

The option meijerg=True, False, None can be used to, respectively: always use G-function methods and no others, never use G-function methods, or use all available methods (in order as described above). It defaults to None.

Examples

```
>>> from sympy import integrate, log, exp, oo
>>> from sympy.abc import a, x, y
```

```
>>> integrate(x*y, x)
x**2*y/2
```

```
>>> integrate(log(x), x)
x*log(x) - x
```

```
>>> integrate(log(x), (x, 1, a))
a*log(a) - a + 1
```

```
>>> integrate(x)
x**2/2
```

Terms that are independent of x are dropped by indefinite integration:

```
>>> from sympy import sqrt
>>> integrate(sqrt(1 + x), (x, 0, x))
2*(x + 1)**(3/2)/3 - 2/3
>>> integrate(sqrt(1 + x), x)
2*(x + 1)**(3/2)/3
```

```
>>> integrate(x*y)
Traceback (most recent call last):
...
ValueError: specify integration variables to integrate x*y
```

Note that integrate(x) syntax is meant only for convenience in interactive sessions and should be avoided in library code.

```
>>> integrate(x**a*exp(-x), (x, 0, oo)) # same as conds='piecewise'
Piecewise((gamma(a + 1), re(a) > -1),
    (Integral(x**a*exp(-x), (x, 0, oo)), True))
```

```
>>> integrate(x**a*exp(-x), (x, 0, oo), conds='none')
gamma(a + 1)
```

```
>>> integrate(x**a*exp(-x), (x, 0, oo), conds='separate')
(gamma(a + 1), re(a) > -1)
```

See also:

Integral (page 601), *Integral.doit* (page 603)

`sympy.integrals.integrals.`**`line_integrate`**(*field*, *Curve*, *variables*)

Compute the line integral.

Examples

```
>>> from sympy import Curve, line_integrate, E, ln
>>> from sympy.abc import x, y, t
>>> C = Curve([E**t + 1, E**t - 1], (t, 0, ln(2)))
>>> line_integrate(x + y, C, [x, y])
3*sqrt(2)
```

See also:

sympy.integrals.integrals.integrate (page 598), *Integral* (page 601)

The class *Integral* (page 601) represents an unevaluated integral and has some methods that help in the integration of an expression.

class `sympy.integrals.integrals.`**`Integral`**(*function*, **symbols*, ***assumptions*)

Represents unevaluated integral.

`is_commutative`

Returns whether all the free symbols in the integral are commutative.

`as_sum`(*n=None*, *method='midpoint'*, *evaluate=True*)

Approximates a definite integral by a sum.

Parameters

n :

The number of subintervals to use, optional.

method :

One of: ‘left’, ‘right’, ‘midpoint’, ‘trapezoid’.

evaluate : bool

If False, returns an unevaluated Sum expression. The default is True, evaluate the sum.

Notes

These methods of approximate integration are described in [1].

Examples

```
>>> from sympy import Integral, sin, sqrt
>>> from sympy.abc import x, n
>>> e = Integral(sin(x), (x, 3, 7))
>>> e
Integral(sin(x), (x, 3, 7))
```

For demonstration purposes, this interval will only be split into 2 regions, bounded by [3, 5] and [5, 7].

The left-hand rule uses function evaluations at the left of each interval:

```
>>> e.as_sum(2, 'left')
2*sin(5) + 2*sin(3)
```

The midpoint rule uses evaluations at the center of each interval:

```
>>> e.as_sum(2, 'midpoint')
2*sin(4) + 2*sin(6)
```

The right-hand rule uses function evaluations at the right of each interval:

```
>>> e.as_sum(2, 'right')
2*sin(5) + 2*sin(7)
```

The trapezoid rule uses function evaluations on both sides of the intervals. This is equivalent to taking the average of the left and right hand rule results:

```
>>> e.as_sum(2, 'trapezoid')
2*sin(5) + sin(3) + sin(7)
>>> (e.as_sum(2, 'left') + e.as_sum(2, 'right'))/2 == _
True
```

Here, the discontinuity at x = 0 can be avoided by using the midpoint or right-hand method:

```
>>> e = Integral(1/sqrt(x), (x, 0, 1))
>>> e.as_sum(5).n(4)
1.730
>>> e.as_sum(10).n(4)
1.809
>>> e.doit().n(4)  # the actual value is 2
2.000
```

The left- or trapezoid method will encounter the discontinuity and return infinity:

```
>>> e.as_sum(5, 'left')
zoo
```

The number of intervals can be symbolic. If omitted, a dummy symbol will be used for it.

```
>>> e = Integral(x**2, (x, 0, 2))
>>> e.as_sum(n, 'right').expand()
8/3 + 4/n + 4/(3*n**2)
```

This shows that the midpoint rule is more accurate, as its error term decays as the square of n:

```
>>> e.as_sum(method='midpoint').expand()
8/3 - 2/(3*_n**2)
```

A symbolic sum is returned with evaluate=False:

```
>>> e.as_sum(n, 'midpoint', evaluate=False)
2*Sum((2*_k/n - 1/n)**2, (_k, 1, n))/n
```

See also:

Integral.doit **(page 603)**
Perform the integration using any hints

References

[R532]

doit(***hints*)

Perform the integration using any hints given.

Examples

```
>>> from sympy import Piecewise, S
>>> from sympy.abc import x, t
>>> p = x**2 + Piecewise((0, x/t < 0), (1, True))
>>> p.integrate((t, S(4)/5, 1), (x, -1, 1))
1/3
```

See also:

sympy.integrals.trigonometry.trigintegrate (page 589), *sympy.integrals.heurisch.heurisch* (page 596), *sympy.integrals.rationaltools.ratint* (page 587)

as_sum **(page 601)**
Approximate the integral using a sum

property free_symbols

This method returns the symbols that will exist when the integral is evaluated. This is useful if one is trying to determine whether an integral depends on a certain symbol or not.

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, y
>>> Integral(x, (x, y, 1)).free_symbols
{y}
```

See also:

sympy.concrete.expr_with_limits.ExprWithLimits.function (page 607), *sympy.concrete.expr_with_limits.ExprWithLimits.limits* (page 608), *sympy.concrete.expr_with_limits.ExprWithLimits.variables* (page 608)

principal_value(***kwargs*)

Compute the Cauchy Principal Value of the definite integral of a real function in the given interval on the real axis.

Explanation

In mathematics, the Cauchy principal value, is a method for assigning values to certain improper integrals which would otherwise be undefined.

Examples

```
>>> from sympy import Integral, oo
>>> from sympy.abc import x
>>> Integral(x+1, (x, -oo, oo)).principal_value()
oo
>>> f = 1 / (x**3)
>>> Integral(f, (x, -oo, oo)).principal_value()
0
>>> Integral(f, (x, -10, 10)).principal_value()
0
>>> Integral(f, (x, -10, oo)).principal_value() + Integral(f, (x, -oo,
↪ 10)).principal_value()
0
```

References

[R533], [R534]

transform(x, u)

Performs a change of variables from x to u using the relationship given by x and u which will define the transformations f and F (which are inverses of each other) as follows:

1) If x is a Symbol (which is a variable of integration) then u will be interpreted as some function, f(u), with inverse F(u). This, in effect, just makes the substitution of x with f(x).

2) If u is a Symbol then x will be interpreted as some function, F(x), with inverse f(u). This is commonly referred to as u-substitution.

Once f and F have been identified, the transformation is made as follows:

$$\int_a^b x \mathrm{d}x \rightarrow \int_{F(a)}^{F(b)} f(x) \frac{\mathrm{d}}{\mathrm{d}x}$$

where $F(x)$ is the inverse of $f(x)$ and the limits and integrand have been corrected so as to retain the same value after integration.

Notes

The mappings, F(x) or f(u), must lead to a unique integral. Linear or rational linear expression, 2*x, 1/x and sqrt(x), will always work; quadratic expressions like x**2 - 1 are acceptable as long as the resulting integrand does not depend on the sign of the solutions (see examples).

The integral will be returned unchanged if x is not a variable of integration.

x must be (or contain) only one of of the integration variables. If u has more than one free symbol then it should be sent as a tuple (u, uvar) where uvar identifies which variable is replacing the integration variable. XXX can it contain another integration variable?

Examples

```
>>> from sympy.abc import a, x, u
>>> from sympy import Integral, cos, sqrt
```

```
>>> i = Integral(x*cos(x**2 - 1), (x, 0, 1))
```

transform can change the variable of integration

```
>>> i.transform(x, u)
Integral(u*cos(u**2 - 1), (u, 0, 1))
```

transform can perform u-substitution as long as a unique integrand is obtained:

```
>>> i.transform(x**2 - 1, u)
Integral(cos(u)/2, (u, -1, 0))
```

This attempt fails because x = +/-sqrt(u + 1) and the sign does not cancel out of the integrand:

```
>>> Integral(cos(x**2 - 1), (x, 0, 1)).transform(x**2 - 1, u)
Traceback (most recent call last):
...
ValueError:
The mapping between F(x) and f(u) did not give a unique integrand.
```

transform can do a substitution. Here, the previous result is transformed back into the original expression using "u-substitution":

```
>>> ui = _
>>> _.transform(sqrt(u + 1), x) == i
True
```

We can accomplish the same with a regular substitution:

```
>>> ui.transform(u, x**2 - 1) == i
True
```

If the x does not contain a symbol of integration then the integral will be returned unchanged. Integral i does not have an integration variable a so no change is made:

```
>>> i.transform(a, x) == i
True
```

When u has more than one free symbol the symbol that is replacing x must be identified by passing u as a tuple:

```
>>> Integral(x, (x, 0, 1)).transform(x, (u + a, u))
Integral(a + u, (u, -a, 1 - a))
>>> Integral(x, (x, 0, 1)).transform(x, (u + a, a))
Integral(a + u, (a, -u, 1 - u))
```

See also:

sympy.concrete.expr_with_limits.ExprWithLimits.variables **(page 608)**
Lists the integration variables

as_dummy **(page 929)**
Replace integration variables with dummy ones

Integral (page 601) subclasses from *ExprWithLimits* (page 606), which is a common superclass of *Integral* (page 601) and *Sum* (page 900).

class sympy.concrete.expr_with_limits.**ExprWithLimits**(*function*, **symbols*, ***assumptions*)

property bound_symbols

Return only variables that are dummy variables.

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, i, j, k
>>> Integral(x**i, (i, 1, 3), (j, 2), k).bound_symbols
[i, j]
```

See also:

function (page 607), *limits* (page 608), *free_symbols* (page 606)

as_dummy **(page 929)**
Rename dummy variables

sympy.integrals.integrals.Integral.transform **(page 604)**
Perform mapping on the dummy variable

property free_symbols

This method returns the symbols in the object, excluding those that take on a specific value (i.e. the dummy symbols).

Examples

```
>>> from sympy import Sum
>>> from sympy.abc import x, y
>>> Sum(x, (x, y, 1)).free_symbols
{y}
```

property function

Return the function applied across limits.

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x
>>> Integral(x**2, (x,)).function
x**2
```

See also:

limits (page 608), *variables* (page 608), *free_symbols* (page 606)

property has_finite_limits

Returns True if the limits are known to be finite, either by the explicit bounds, assumptions on the bounds, or assumptions on the variables. False if known to be infinite, based on the bounds. None if not enough information is available to determine.

Examples

```
>>> from sympy import Sum, Integral, Product, oo, Symbol
>>> x = Symbol('x')
>>> Sum(x, (x, 1, 8)).has_finite_limits
True
```

```
>>> Integral(x, (x, 1, oo)).has_finite_limits
False
```

```
>>> M = Symbol('M')
>>> Sum(x, (x, 1, M)).has_finite_limits
```

```
>>> N = Symbol('N', integer=True)
>>> Product(x, (x, 1, N)).has_finite_limits
True
```

See also:

has_reversed_limits (page 607)

property has_reversed_limits

Returns True if the limits are known to be in reversed order, either by the explicit bounds, assumptions on the bounds, or assumptions on the variables. False if known to be in normal order, based on the bounds. None if not enough information is available to determine.

Examples

```
>>> from sympy import Sum, Integral, Product, oo, Symbol
>>> x = Symbol('x')
>>> Sum(x, (x, 8, 1)).has_reversed_limits
True
```

```
>>> Sum(x, (x, 1, oo)).has_reversed_limits
False
```

```
>>> M = Symbol('M')
>>> Integral(x, (x, 1, M)).has_reversed_limits
```

```
>>> N = Symbol('N', integer=True, positive=True)
>>> Sum(x, (x, 1, N)).has_reversed_limits
False
```

```
>>> Product(x, (x, 2, N)).has_reversed_limits
```

```
>>> Product(x, (x, 2, N)).subs(N, N + 2).has_reversed_limits
False
```

See also:

sympy.concrete.expr_with_intlimits.ExprWithIntLimits.has_empty_sequence (page 912)

property is_number

Return True if the Sum has no free symbols, else False.

property limits

Return the limits of expression.

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, i
>>> Integral(x**i, (i, 1, 3)).limits
((i, 1, 3),)
```

See also:

function (page 607), *variables* (page 608), *free_symbols* (page 606)

property variables

Return a list of the limit variables.

```
>>> from sympy import Sum
>>> from sympy.abc import x, i
>>> Sum(x**i, (i, 1, 3)).variables
[i]
```

See also:

function (page 607), *limits* (page 608), *free_symbols* (page 606)

as_dummy **(page 929)**
Rename dummy variables

sympy.integrals.integrals.Integral.transform **(page 604)**
Perform mapping on the dummy variable

TODO and Bugs

There are still lots of functions that SymPy does not know how to integrate. For bugs related to this module, see https://github.com/sympy/sympy/issues?q=is%3Aissue+is%3Aopen+label%3Aintegrals

Numeric Integrals

SymPy has functions to calculate points and weights for Gaussian quadrature of any order and any precision:

sympy.integrals.quadrature.**gauss_legendre**(*n*, *n_digits*)

Computes the Gauss-Legendre quadrature [R535] points and weights.

Parameters

n :

The order of quadrature.

n_digits :

Number of significant digits of the points and weights to return.

Returns

(x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

Explanation

The Gauss-Legendre quadrature approximates the integral:

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of P_n and the weights w_i are given by:

$$w_i = \frac{2}{\left(1 - x_i^2\right)\left(P'_n(x_i)\right)^2}$$

Examples

```
>>> from sympy.integrals.quadrature import gauss_legendre
>>> x, w = gauss_legendre(3, 5)
>>> x
[-0.7746, 0, 0.7746]
>>> w
[0.55556, 0.88889, 0.55556]
>>> x, w = gauss_legendre(4, 5)
>>> x
[-0.86114, -0.33998, 0.33998, 0.86114]
>>> w
[0.34785, 0.65215, 0.65215, 0.34785]
```

See also:

gauss_laguerre (page 610), *gauss_gen_laguerre* (page 612), *gauss_hermite* (page 611), *gauss_chebyshev_t* (page 613), *gauss_chebyshev_u* (page 614), *gauss_jacobi* (page 615), *gauss_lobatto* (page 616)

References

[R535], [R536]

sympy.integrals.quadrature.**gauss_laguerre**(*n*, *n_digits*)

Computes the Gauss-Laguerre quadrature [R537] points and weights.

Parameters

n :

The order of quadrature.

n_digits :

Number of significant digits of the points and weights to return.

Returns

(x, w) : The `x` and `w` are lists of points and weights as Floats.

The points x_i and weights w_i are returned as `(x, w)` tuple of lists.

Explanation

The Gauss-Laguerre quadrature approximates the integral:

$$\int_0^{\infty} e^{-x} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of L_n and the weights w_i are given by:

$$w_i = \frac{x_i}{(n+1)^2 \left(L_{n+1}(x_i)\right)^2}$$

Examples

```
>>> from sympy.integrals.quadrature import gauss_laguerre
>>> x, w = gauss_laguerre(3, 5)
>>> x
[0.41577, 2.2943, 6.2899]
>>> w
[0.71109, 0.27852, 0.010389]
>>> x, w = gauss_laguerre(6, 5)
>>> x
[0.22285, 1.1889, 2.9927, 5.7751, 9.8375, 15.983]
>>> w
[0.45896, 0.417, 0.11337, 0.010399, 0.00026102, 8.9855e-7]
```

See also:

gauss_legendre (page 609), *gauss_gen_laguerre* (page 612), *gauss_hermite* (page 611), *gauss_chebyshev_t* (page 613), *gauss_chebyshev_u* (page 614), *gauss_jacobi* (page 615), *gauss_lobatto* (page 616)

References

[R537], [R538]

`sympy.integrals.quadrature.`**`gauss_hermite`**`(`*n*, *n_digits*`)`

Computes the Gauss-Hermite quadrature [R539] points and weights.

Parameters

n :

The order of quadrature.

n_digits :

Number of significant digits of the points and weights to return.

Returns

(x, w) : The `x` and `w` are lists of points and weights as Floats.

The points x_i and weights w_i are returned as `(x, w)` tuple of lists.

Explanation

The Gauss-Hermite quadrature approximates the integral:

$$\int_{-\infty}^{\infty} e^{-x^2} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of H_n and the weights w_i are given by:

$$w_i = \frac{2^{n-1} n! \sqrt{\pi}}{n^2 \left(H_{n-1}(x_i)\right)^2}$$

Examples

```
>>> from sympy.integrals.quadrature import gauss_hermite
>>> x, w = gauss_hermite(3, 5)
>>> x
[-1.2247, 0, 1.2247]
>>> w
[0.29541, 1.1816, 0.29541]
```

```
>>> x, w = gauss_hermite(6, 5)
>>> x
[-2.3506, -1.3358, -0.43608, 0.43608, 1.3358, 2.3506]
>>> w
[0.00453, 0.15707, 0.72463, 0.72463, 0.15707, 0.00453]
```

See also:

gauss_legendre (page 609), *gauss_laguerre* (page 610), *gauss_gen_laguerre* (page 612), *gauss_chebyshev_t* (page 613), *gauss_chebyshev_u* (page 614), *gauss_jacobi* (page 615), *gauss_lobatto* (page 616)

References

[R539], [R540], [R541]

sympy.integrals.quadrature.**gauss_gen_laguerre**(*n*, *alpha*, *n_digits*)

Computes the generalized Gauss-Laguerre quadrature [R542] points and weights.

Parameters

n :

The order of quadrature.

alpha :

The exponent of the singularity, $\alpha > -1$.

n_digits :

Number of significant digits of the points and weights to return.

Returns

(x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

Explanation

The generalized Gauss-Laguerre quadrature approximates the integral:

$$\int_0^{\infty} x^{\alpha} e^{-x} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of L_n^{α} and the weights w_i are given by:

$$w_i = \frac{\Gamma(\alpha+n)}{n\Gamma(n)L_{n-1}^{\alpha}(x_i)L_{n-1}^{\alpha+1}(x_i)}$$

Examples

```
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_gen_laguerre
>>> x, w = gauss_gen_laguerre(3, -S.Half, 5)
>>> x
[0.19016, 1.7845, 5.5253]
>>> w
[1.4493, 0.31413, 0.00906]
```

```
>>> x, w = gauss_gen_laguerre(4, 3*S.Half, 5)
>>> x
[0.97851, 2.9904, 6.3193, 11.712]
>>> w
[0.53087, 0.67721, 0.11895, 0.0023152]
```

See also:

gauss_legendre (page 609), *gauss_laguerre* (page 610), *gauss_hermite* (page 611), *gauss_chebyshev_t* (page 613), *gauss_chebyshev_u* (page 614), *gauss_jacobi* (page 615), *gauss_lobatto* (page 616)

References

[R542], [R543]

`sympy.integrals.quadrature.`**`gauss_chebyshev_t`**`(`*n*, *n_digits*`)`

Computes the Gauss-Chebyshev quadrature [R544] points and weights of the first kind.

Parameters

n :

The order of quadrature.

n_digits :

Number of significant digits of the points and weights to return.

Returns

(x, w) : the `x` and `w` are lists of points and weights as Floats.

The points x_i and weights w_i are returned as `(x, w)` tuple of lists.

Explanation

The Gauss-Chebyshev quadrature of the first kind approximates the integral:

$$\int_{-1}^{1} \frac{1}{\sqrt{1-x^2}} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of T_n and the weights w_i are given by:

$$w_i = \frac{\pi}{n}$$

Examples

```
>>> from sympy.integrals.quadrature import gauss_chebyshev_t
>>> x, w = gauss_chebyshev_t(3, 5)
>>> x
[0.86602, 0, -0.86602]
>>> w
[1.0472, 1.0472, 1.0472]
```

```
>>> x, w = gauss_chebyshev_t(6, 5)
>>> x
[0.96593, 0.70711, 0.25882, -0.25882, -0.70711, -0.96593]
>>> w
[0.5236, 0.5236, 0.5236, 0.5236, 0.5236, 0.5236]
```

See also:

gauss_legendre (page 609), *gauss_laguerre* (page 610), *gauss_hermite* (page 611), *gauss_gen_laguerre* (page 612), *gauss_chebyshev_u* (page 614), *gauss_jacobi* (page 615), *gauss_lobatto* (page 616)

References

[R544], [R545]

sympy.integrals.quadrature.**gauss_chebyshev_u**(*n*, *n_digits*)

Computes the Gauss-Chebyshev quadrature [R546] points and weights of the second kind.

Parameters

n : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns

(x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

Explanation

The Gauss-Chebyshev quadrature of the second kind approximates the integral:

$$\int_{-1}^{1} \sqrt{1-x^2} f(x)\, dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of U_n and the weights w_i are given by:

$$w_i = \frac{\pi}{n+1} \sin^2 \left(\frac{i}{n+1} \pi \right)$$

Examples

```
>>> from sympy.integrals.quadrature import gauss_chebyshev_u
>>> x, w = gauss_chebyshev_u(3, 5)
>>> x
[0.70711, 0, -0.70711]
>>> w
[0.3927, 0.7854, 0.3927]
```

```
>>> x, w = gauss_chebyshev_u(6, 5)
>>> x
[0.90097, 0.62349, 0.22252, -0.22252, -0.62349, -0.90097]
>>> w
[0.084489, 0.27433, 0.42658, 0.42658, 0.27433, 0.084489]
```

See also:

gauss_legendre (page 609), *gauss_laguerre* (page 610), *gauss_hermite* (page 611), *gauss_gen_laguerre* (page 612), *gauss_chebyshev_t* (page 613), *gauss_jacobi* (page 615), *gauss_lobatto* (page 616)

References

[R546], [R547]

sympy.integrals.quadrature.**gauss_jacobi**(*n, alpha, beta, n_digits*)

Computes the Gauss-Jacobi quadrature [R548] points and weights.

Parameters

n : the order of quadrature

alpha : the first parameter of the Jacobi Polynomial, $\alpha > -1$

beta : the second parameter of the Jacobi Polynomial, $\beta > -1$

n_digits : number of significant digits of the points and weights to return

Returns

(x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

Explanation

The Gauss-Jacobi quadrature of the first kind approximates the integral:

$$\int_{-1}^{1} (1-x)^\alpha (1+x)^\beta f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of $P_n^{(\alpha,\beta)}$ and the weights w_i are given by:

$$w_i = -\frac{2n+\alpha+\beta+2}{n+\alpha+\beta+1} \frac{\Gamma(n+\alpha+1)\Gamma(n+\beta+1)}{\Gamma(n+\alpha+\beta+1)(n+1)!} \frac{2^{\alpha+\beta}}{P'_n(x_i) P^{(\alpha,\beta)}_{n+1}(x_i)}$$

Examples

```
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_jacobi
>>> x, w = gauss_jacobi(3, S.Half, -S.Half, 5)
>>> x
[-0.90097, -0.22252, 0.62349]
>>> w
[1.7063, 1.0973, 0.33795]
```

```
>>> x, w = gauss_jacobi(6, 1, 1, 5)
>>> x
[-0.87174, -0.5917, -0.2093, 0.2093, 0.5917, 0.87174]
>>> w
[0.050584, 0.22169, 0.39439, 0.39439, 0.22169, 0.050584]
```

See also:

gauss_legendre (page 609), *gauss_laguerre* (page 610), *gauss_hermite* (page 611), *gauss_gen_laguerre* (page 612), *gauss_chebyshev_t* (page 613), *gauss_chebyshev_u* (page 614), *gauss_lobatto* (page 616)

References

[R548], [R549], [R550]

sympy.integrals.quadrature.**gauss_lobatto**(*n*, *n_digits*)

Computes the Gauss-Lobatto quadrature [R551] points and weights.

Parameters

n : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns

(x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

Explanation

The Gauss-Lobatto quadrature approximates the integral:

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of $P'_(n-1)$ and the weights w_i are given by:

$$w_i = \frac{2}{n(n-1)\left[P_{n-1}(x_i)\right]^2}, \quad x \neq \pm 1$$
$$w_i = \frac{2}{n(n-1)}, \quad x = \pm 1$$

Examples

```
>>> from sympy.integrals.quadrature import gauss_lobatto
>>> x, w = gauss_lobatto(3, 5)
>>> x
[-1, 0, 1]
>>> w
[0.33333, 1.3333, 0.33333]
>>> x, w = gauss_lobatto(4, 5)
>>> x
[-1, -0.44721, 0.44721, 1]
>>> w
[0.16667, 0.83333, 0.83333, 0.16667]
```

See also:

gauss_legendre (page 609), *gauss_laguerre* (page 610), *gauss_gen_laguerre* (page 612), *gauss_hermite* (page 611), *gauss_chebyshev_t* (page 613), *gauss_chebyshev_u* (page 614), *gauss_jacobi* (page 615)

References

[R551], [R552]

Integration over Polytopes

The intpoly module in SymPy implements methods to calculate the integral of a polynomial over 2/3-Polytopes. Uses evaluation techniques as described in Chin et al. (2015) [1].

The input for 2-Polytope or Polygon uses the already existing Polygon data structure in SymPy. See *sympy.geometry.polygon* (page 2273) for how to create a polygon.

For the 3-Polytope or Polyhedron, the most economical representation is to specify a list of vertices and then to provide each constituting face(Polygon) as a list of vertex indices.

For example, consider the unit cube. Here is how it would be represented.

```
unit_cube = [[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0),(1, 0, 1),
(1, 1, 0), (1, 1, 1)],
    [3, 7, 6, 2], [1, 5, 7, 3], [5, 4, 6, 7], [0, 4, 5, 1], [2, 0, 1, 3], [2,
    6, 4, 0]]
```

Here, the first sublist is the list of vertices. The other smaller lists such as [3, 7, 6, 2] represent a 2D face of the polyhedra with vertices having index 3, 7, 6 and 2 in the first sublist(in that order).

Principal method in this module is *polytope_integrate()* (page 619)

- polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), x) returns the integral of x over the triangle with vertices (0, 0), (0, 1) and (1, 0)
- polytope_integrate(unit_cube, x + y + z) returns the integral of $x + y + z$ over the unit cube.

References

[1] : Chin, Eric B., Jean B. Lasserre, and N. Sukumar. “Numerical integration of homogeneous functions on convex and nonconvex polygons and polyhedra.” Computational Mechanics 56.6 (2015): 967-981

PDF link : http://dilbert.engr.ucdavis.edu/~suku/quadrature/cls-integration.pdf

Examples

For 2D Polygons

Single Polynomial:

```
>>> from sympy.integrals.intpoly import *
>>> init_printing(use_unicode=False, wrap_line=False)
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), x)
1/6
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), x + x*y + y**2)
7/24
```

List of specified polynomials:

```
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [3, x*y + y**2, x**4],
 ↪ max_degree=4)
          4               2
{3: 3/2, x : 1/30, x*y + y : 1/8}
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [1.125, x, x**2, 6.
 ↪89*x**3, x*y + y**2, x**4], max_degree=4)
                       2              3  689    4               2
{1.125: 9/16, x: 1/6, x : 1/12, 6.89*x : ----, x : 1/30, x*y + y : 1/8}
                                         2000
```

Computing all monomials up to a maximum degree:

```
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)),max_degree=3)
                        2         3                 2         3                 ␣
 ↪       2         2
{0: 0, 1: 1/2, x: 1/6, x : 1/12, x : 1/20, y: 1/6, y : 1/12, y : 1/20, x*y: 1/
 ↪24, x*y : 1/60, x *y: 1/60}
```

For 3-Polytopes/Polyhedra

Single Polynomial:

```
>>> from sympy.integrals.intpoly import *
>>> cube = [[(0, 0, 0), (0, 0, 5), (0, 5, 0), (0, 5, 5), (5, 0, 0), (5, 0, 5),
 ↪ (5, 5, 0), (5, 5, 5)], [2, 6, 7, 3], [3, 7, 5, 1], [7, 6, 4, 5], [1, 5, 4,␣
 ↪0], [3, 1, 0, 2], [0, 4, 6, 2]]
>>> polytope_integrate(cube, x**2 + y**2 + z**2 + x*y + y*z + x*z)
-21875/4
```

(continues on next page)

(continued from previous page)

```
>>> octahedron = [[(S(-1) / sqrt(2), 0, 0), (0, S(1) / sqrt(2), 0), (0, 0, S(-
↪1) / sqrt(2)), (0, 0, S(1) / sqrt(2)), (0, S(-1) / sqrt(2), 0), (S(1) /␣
↪sqrt(2), 0, 0)], [3, 4, 5], [3, 5, 1], [3, 1, 0], [3, 0, 4], [4, 0, 2], [4,␣
↪2, 5], [2, 0, 1], [5, 2, 1]]
>>> polytope_integrate(octahedron, x**2 + y**2 + z**2 + x*y + y*z + x*z)
  ___
\/ 2
-----
  20
```

List of specified polynomials:

```
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [3, x*y + y**2, x**4],
↪ max_degree=4)
          4               2
{3: 3/2, x : 1/30, x*y + y : 1/8}
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [1.125, x, x**2, 6.
↪89*x**3, x*y + y**2, x**4], max_degree=4)
                       2              3  689    4               2
{1.125: 9/16, x: 1/6, x : 1/12, 6.89*x : ----, x : 1/30, x*y + y : 1/8}
                                         2000
```

Computing all monomials up to a maximum degree:

```
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)),max_degree=3)
                        2         3                 2         3               ␣
↪       2         2
{0: 0, 1: 1/2, x: 1/6, x : 1/12, x : 1/20, y: 1/6, y : 1/12, y : 1/20, x*y: 1/
↪24, x*y : 1/60, x *y: 1/60}
```

API reference

sympy.integrals.intpoly.**polytope_integrate**(*poly, expr=None, *, clockwise=False, max_degree=None*)

Integrates polynomials over 2/3-Polytopes.

Parameters

poly : The input Polygon.

expr : The input polynomial.

clockwise : Binary value to sort input points of 2-Polytope clockwise.(Optional)

max_degree : The maximum degree of any monomial of the input polynomial.(Optional)

Explanation

This function accepts the polytope in `poly` and the function in `expr` (uni/bi/trivariate polynomials are implemented) and returns the exact integral of `expr` over `poly`.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import Point, Polygon
>>> from sympy.integrals.intpoly import polytope_integrate
>>> polygon = Polygon(Point(0, 0), Point(0, 1), Point(1, 1), Point(1, 0))
>>> polys = [1, x, y, x*y, x**2*y, x*y**2]
>>> expr = x*y
>>> polytope_integrate(polygon, expr)
1/4
>>> polytope_integrate(polygon, polys, max_degree=3)
{1: 1, x: 1/2, y: 1/2, x*y: 1/4, x*y**2: 1/6, x**2*y: 1/6}
```

Series

The series module implements series expansions as a function and many related functions.

Contents

Series Expansions

Limits

The main purpose of this module is the computation of limits.

`sympy.series.limits.`**`limit`**(*e, z, z0, dir*='+')

Computes the limit of `e(z)` at the point `z0`.

Parameters

e : expression, the limit of which is to be taken

z : symbol representing the variable in the limit.

Other symbols are treated as constants. Multivariate limits are not supported.

z0 : the value toward which `z` tends. Can be any expression,

including `oo` and `-oo`.

dir : string, optional (default: "+")

The limit is bi-directional if `dir=”+-”`, from the right (z->z0+) if `dir=”+”`, and from the left (z->z0-) if `dir=”-”`. For infinite `z0` (`oo` or `-oo`), the `dir` argument is determined from the direction of the infinity (i.e., `dir=”-”` for `oo`).

Examples

```
>>> from sympy import limit, sin, oo
>>> from sympy.abc import x
>>> limit(sin(x)/x, x, 0)
1
>>> limit(1/x, x, 0) # default dir='+'
oo
>>> limit(1/x, x, 0, dir=”-”)
-oo
>>> limit(1/x, x, 0, dir='+-')
zoo
>>> limit(1/x, x, oo)
0
```

Notes

First we try some heuristics for easy and frequent cases like “x”, “1/x”, “x**2” and similar, so that it’s fast. For all other cases, we use the Gruntz algorithm (see the gruntz() function).

See also:

limit_seq **(page 660)**
returns the limit of a sequence.

class sympy.series.limits.**Limit**(*e*, *z*, *z0*, *dir='+'*)

Represents an unevaluated limit.

Examples

```
>>> from sympy import Limit, sin
>>> from sympy.abc import x
>>> Limit(sin(x)/x, x, 0)
Limit(sin(x)/x, x, 0)
>>> Limit(1/x, x, 0, dir=”-”)
Limit(1/x, x, 0, dir='-')
```

doit(***hints*)

Evaluates the limit.

Parameters

deep : bool, optional (default: True)

Invoke the `doit` method of the expressions involved before taking the limit.

hints : optional keyword arguments

To be passed to `doit` methods; only used if deep is True.

As is explained above, the workhorse for limit computations is the function gruntz() which implements Gruntz’ algorithm for computing limits.

The Gruntz Algorithm

This section explains the basics of the algorithm used for computing limits. Most of the time the limit() function should just work. However it is still useful to keep in mind how it is implemented in case something does not work as expected.

First we define an ordering on functions. Suppose $f(x)$ and $g(x)$ are two real-valued functions such that $\lim_{x \to \infty} f(x) = \infty$ and similarly $\lim_{x \to \infty} g(x) = \infty$. We shall say that $f(x)$ *dominates* $g(x)$, written $f(x) \succ g(x)$, if for all $a, b \in \mathbb{R}_{>0}$ we have $\lim_{x \to \infty} \frac{f(x)^a}{g(x)^b} = \infty$. We also say that $f(x)$ and $g(x)$ are *of the same comparability class* if neither $f(x) \succ g(x)$ nor $g(x) \succ f(x)$ and shall denote it as $f(x) \asymp g(x)$.

Note that whenever $a, b \in \mathbb{R}_{>0}$ then $af(x)^b \asymp f(x)$, and we shall use this to extend the definition of $\succ$ to all functions which tend to 0 or $\pm\infty$ as $x \to \infty$. Thus we declare that $f(x) \asymp 1/f(x)$ and $f(x) \asymp -f(x)$.

It is easy to show the following examples:

- $e^x \succ x^m$
- $e^{x^2} \succ e^{mx}$
- $e^{e^x} \succ e^{x^m}$
- $x^m \asymp x^n$
- $e^{x + \frac{1}{x}} \asymp e^{x + \log x} \asymp e^x$.

From the above definition, it is possible to prove the following property:

> Suppose ω, $g_1, g_2, \ldots$ are functions of x, $\lim_{x \to \infty} \omega = 0$ and $\omega \succ g_i$ for all i. Let $c_1, c_2, \ldots \in \mathbb{R}$ with $c_1 < c_2 < \cdots$.
>
> Then $\lim_{x \to \infty} \sum_i g_i \omega^{c_i} = \lim_{x \to \infty} g_1 \omega^{c_1}$.

For $g_1 = g$ and ω as above we also have the following easy result:

- $\lim_{x \to \infty} g\omega^c = 0$ for $c > 0$
- $\lim_{x \to \infty} g\omega^c = \pm\infty$ for $c < 0$, where the sign is determined by the (eventual) sign of g
- $\lim_{x \to \infty} g\omega^0 = \lim_{x \to \infty} g$.

Using these results yields the following strategy for computing $\lim_{x \to \infty} f(x)$:

1. Find the set of *most rapidly varying subexpressions* (MRV set) of $f(x)$. That is, from the set of all subexpressions of $f(x)$, find the elements that are maximal under the relation $\succ$.
2. Choose a function ω that is in the same comparability class as the elements in the MRV set, such that $\lim_{x \to \infty} \omega = 0$.
3. Expand $f(x)$ as a series in ω in such a way that the antecedents of the above theorem are satisfied.
4. Apply the theorem and conclude the computation of $\lim_{x \to \infty} f(x)$, possibly by recursively working on $g_1(x)$.

Notes

This exposition glossed over several details. Many are described in the file gruntz.py, and all can be found in Gruntz' very readable thesis. The most important points that have not been explained are:

1. Given f(x) and g(x), how do we determine if $f(x) \succ g(x)$, $g(x) \succ f(x)$ or $g(x) \asymp f(x)$?
2. How do we find the MRV set of an expression?
3. How do we compute series expansions?
4. Why does the algorithm terminate?

If you are interested, be sure to take a look at Gruntz Thesis.

Reference

`sympy.series.gruntz.`**`gruntz`**`(`*e, z, z0, dir='+'*`)`

Compute the limit of e(z) at the point z0 using the Gruntz algorithm.

Explanation

`z0` can be any expression, including oo and -oo.

For `dir="+"` (default) it calculates the limit from the right (z->z0+) and for `dir="-"` the limit from the left (z->z0-). For infinite z0 (oo or -oo), the dir argument does not matter.

This algorithm is fully described in the module docstring in the gruntz.py file. It relies heavily on the series expansion. Most frequently, gruntz() is only used if the faster limit() function (which uses heuristics) fails.

`sympy.series.gruntz.`**`compare`**`(`*a, b, x*`)`

Returns "<" if a<b, "=" for a == b, ">" for a>b

`sympy.series.gruntz.`**`rewrite`**`(`*e, Omega, x, wsym*`)`

e(x) … the function Omega … the mrv set wsym … the symbol which is going to be used for w

Returns the rewritten e in terms of w and log(w). See test_rewrite1() for examples and correct results.

`sympy.series.gruntz.`**`build_expression_tree`**`(`*Omega, rewrites*`)`

Helper function for rewrite.

We need to sort Omega (mrv set) so that we replace an expression before we replace any expression in terms of which it has to be rewritten:

```
e1 ---> e2 ---> e3
         \
          -> e4
```

Here we can do e1, e2, e3, e4 or e1, e2, e4, e3. To do this we assemble the nodes into a tree, and sort them by height.

This function builds the tree, rewrites then sorts the nodes.

sympy.series.gruntz.**mrv_leadterm**(*e*, *x*)

Returns (c0, e0) for e.

sympy.series.gruntz.**calculate_series**(*e*, *x*, *logx=None*)

Calculates at least one term of the series of `e` in `x`.

This is a place that fails most often, so it is in its own function.

sympy.series.gruntz.**limitinf**(*e*, *x*, *leadsimp=False*)

Limit e(x) for x-> oo.

Explanation

If `leadsimp` is True, an attempt is made to simplify the leading term of the series expansion of `e`. That may succeed even if `e` cannot be simplified.

sympy.series.gruntz.**sign**(*e*, *x*)

Returns a sign of an expression e(x) for x->oo.

```
e >  0 for x sufficiently large ...  1
e == 0 for x sufficiently large ...  0
e <  0 for x sufficiently large ... -1
```

The result of this function is currently undefined if e changes sign arbitrarily often for arbitrarily large x (e.g. sin(x)).

Note that this returns zero only if e is *constantly* zero for x sufficiently large. [If e is constant, of course, this is just the same thing as the sign of e.]

sympy.series.gruntz.**mrv**(*e*, *x*)

Returns a SubsSet of most rapidly varying (mrv) subexpressions of 'e', and e rewritten in terms of these

sympy.series.gruntz.**mrv_max1**(*f*, *g*, *exps*, *x*)

Computes the maximum of two sets of expressions f and g, which are in the same comparability class, i.e. mrv_max1() compares (two elements of) f and g and returns the set, which is in the higher comparability class of the union of both, if they have the same order of variation. Also returns exps, with the appropriate substitutions made.

sympy.series.gruntz.**mrv_max3**(*f*, *expsf*, *g*, *expsg*, *union*, *expsboth*, *x*)

Computes the maximum of two sets of expressions f and g, which are in the same comparability class, i.e. max() compares (two elements of) f and g and returns either (f, expsf) [if f is larger], (g, expsg) [if g is larger] or (union, expsboth) [if f, g are of the same class].

class sympy.series.gruntz.**SubsSet**

Stores (expr, dummy) pairs, and how to rewrite expr-s.

Explanation

The gruntz algorithm needs to rewrite certain expressions in term of a new variable w. We cannot use subs, because it is just too smart for us. For example:

```
> Omega=[exp(exp(_p - exp(-_p))/(1 - 1/_p)), exp(exp(_p))]
> O2=[exp(-exp(_p) + exp(-exp(-_p))*exp(_p)/(1 - 1/_p))/_w, 1/_w]
> e = exp(exp(_p - exp(-_p))/(1 - 1/_p)) - exp(exp(_p))
> e.subs(Omega[0],O2[0]).subs(Omega[1],O2[1])
-1/w + exp(exp(p)*exp(-exp(-p))/(1 - 1/p))
```

is really not what we want!

So we do it the hard way and keep track of all the things we potentially want to substitute by dummy variables. Consider the expression:

```
exp(x - exp(-x)) + exp(x) + x.
```

The mrv set is {exp(x), exp(-x), exp(x - exp(-x))}. We introduce corresponding dummy variables d1, d2, d3 and rewrite:

```
d3 + d1 + x.
```

This class first of all keeps track of the mapping expr->variable, i.e. will at this stage be a dictionary:

```
{exp(x): d1, exp(-x): d2, exp(x - exp(-x)): d3}.
```

[It turns out to be more convenient this way round.] But sometimes expressions in the mrv set have other expressions from the mrv set as subexpressions, and we need to keep track of that as well. In this case, d3 is really exp(x - d2), so rewrites at this stage is:

```
{d3: exp(x-d2)}.
```

The function rewrite uses all this information to correctly rewrite our expression in terms of w. In this case w can be chosen to be exp(-x), i.e. d2. The correct rewriting then is:

```
exp(-w)/w + 1/w + x.
```

copy()

Create a shallow copy of SubsSet

do_subs(*e*)

Substitute the variables with expressions

meets(*s2*)

Tell whether or not self and s2 have non-empty intersection

union(*s2*, *exps=None*)

Compute the union of self and s2, adjusting exps

More Intuitive Series Expansion

This is achieved by creating a wrapper around Basic.series(). This allows for the use of series(x*cos(x),x), which is possibly more intuitive than (x*cos(x)).series(x).

Examples

```
>>> from sympy import Symbol, cos, series
>>> x = Symbol('x')
>>> series(cos(x),x)
1 - x**2/2 + x**4/24 + O(x**6)
```

Reference

sympy.series.series.**series**(*expr*, *x=None*, *x0=0*, *n=6*, *dir='+'*)

Series expansion of expr around point $x = x0$.

Parameters

expr : Expression

The expression whose series is to be expanded.

x : Symbol

It is the variable of the expression to be calculated.

x0 : Value

The value around which x is calculated. Can be any value from -oo to oo.

n : Value

The number of terms upto which the series is to be expanded.

dir : String, optional

The series-expansion can be bi-directional. If dir="+", then (x->x0+). If dir="-", then (x->x0-). For infinite ``x0 (oo or -oo), the dir argument is determined from the direction of the infinity (i.e., dir="-" for oo).

Returns

Expr

Series expansion of the expression about x0

Examples

```
>>> from sympy import series, tan, oo
>>> from sympy.abc import x
>>> f = tan(x)
>>> series(f, x, 2, 6, ”+”)
tan(2) + (1 + tan(2)**2)*(x - 2) + (x - 2)**2*(tan(2)**3 + tan(2)) +
(x - 2)**3*(1/3 + 4*tan(2)**2/3 + tan(2)**4) + (x - 2)**4*(tan(2)**5 +
5*tan(2)**3/3 + 2*tan(2)/3) + (x - 2)**5*(2/15 + 17*tan(2)**2/15 +
2*tan(2)**4 + tan(2)**6) + O((x - 2)**6, (x, 2))
```

```
>>> series(f, x, 2, 3, ”-”)
tan(2) + (2 - x)*(-tan(2)**2 - 1) + (2 - x)**2*(tan(2)**3 + tan(2))
+ O((x - 2)**3, (x, 2))
```

```
>>> series(f, x, 2, oo, ”+”)
Traceback (most recent call last):
...
TypeError: 'Infinity' object cannot be interpreted as an integer
```

See also:

sympy.core.expr.Expr.series **(page 973)**
See the docstring of Expr.series() for complete details of this wrapper.

Order Terms

This module also implements automatic keeping track of the order of your expansion.

Examples

```
>>> from sympy import Symbol, Order
>>> x = Symbol('x')
>>> Order(x) + x**2
O(x)
>>> Order(x) + 1
1 + O(x)
```

Reference

class sympy.series.order.**Order**(*expr, *args, **kwargs*)
Represents the limiting behavior of some function.

Explanation

The order of a function characterizes the function based on the limiting behavior of the function as it goes to some limit. Only taking the limit point to be a number is currently supported. This is expressed in big O notation [R742].

The formal definition for the order of a function $g(x)$ about a point a is such that $g(x) = O(f(x))$ as $x \to a$ if and only if for any $\delta > 0$ there exists a $M > 0$ such that $|g(x)| \leq M|f(x)|$ for $|x - a| < \delta$. This is equivalent to $\lim_{x \to a} \sup |g(x)/f(x)| < \infty$.

Let's illustrate it on the following example by taking the expansion of $\sin(x)$ about 0:

$$\sin(x) = x - x^3/3! + O(x^5)$$

where in this case $O(x^5) = x^5/5! - x^7/7! + \cdots$. By the definition of O, for any $\delta > 0$ there is an M such that:

$$|x^5/5! - x^7/7! + ...| <= M|x^5| \text{ for } |x| < \delta$$

or by the alternate definition:

$$\lim_{x \to 0} |(x^5/5! - x^7/7! + ...)/x^5| < \infty$$

which surely is true, because

$$\lim_{x \to 0} |(x^5/5! - x^7/7! + ...)/x^5| = 1/5!$$

As it is usually used, the order of a function can be intuitively thought of representing all terms of powers greater than the one specified. For example, $O(x^3)$ corresponds to any terms proportional to $x^3, x^4, \ldots$ and any higher power. For a polynomial, this leaves terms proportional to x^2, x and constants.

Examples

```
>>> from sympy import O, oo, cos, pi
>>> from sympy.abc import x, y
```

```
>>> O(x + x**2)
O(x)
>>> O(x + x**2, (x, 0))
O(x)
>>> O(x + x**2, (x, oo))
O(x**2, (x, oo))
```

```
>>> O(1 + x*y)
O(1, x, y)
>>> O(1 + x*y, (x, 0), (y, 0))
O(1, x, y)
>>> O(1 + x*y, (x, oo), (y, oo))
O(x*y, (x, oo), (y, oo))
```

```
>>> O(1) in O(1, x)
True
>>> O(1, x) in O(1)
False
>>> O(x) in O(1, x)
True
>>> O(x**2) in O(x)
True
```

```
>>> O(x)*x
O(x**2)
>>> O(x) - O(x)
O(x)
>>> O(cos(x))
O(1)
>>> O(cos(x), (x, pi/2))
O(x - pi/2, (x, pi/2))
```

Notes

In `O(f(x), x)` the expression `f(x)` is assumed to have a leading term. `O(f(x), x)` is automatically transformed to `O(f(x).as_leading_term(x),x)`.

> `O(expr*f(x), x)` is `O(f(x), x)`
>
> `O(expr, x)` is `O(1)`
>
> `O(0, x)` is 0.

Multivariate O is also supported:

> `O(f(x, y), x, y)` is transformed to `O(f(x, y).as_leading_term(x,y).as_leading_term(y), x, y)`

In the multivariate case, it is assumed the limits w.r.t. the various symbols commute.

If no symbols are passed then all symbols in the expression are used and the limit point is assumed to be zero.

References

[R742]

contains(*expr*)

> Return True if expr belongs to Order(self.expr, *self.variables). Return False if self belongs to expr. Return None if the inclusion relation cannot be determined (e.g. when self and expr have different symbols).

Series Acceleration

TODO

Reference

sympy.series.acceleration.**richardson**(*A*, *k*, *n*, *N*)

Calculate an approximation for lim k->oo A(k) using Richardson extrapolation with the terms A(n), A(n+1), …, A(n+N+1). Choosing N ~= 2*n often gives good results.

Examples

A simple example is to calculate exp(1) using the limit definition. This limit converges slowly; n = 100 only produces two accurate digits:

```
>>> from sympy.abc import n
>>> e = (1 + 1/n)**n
>>> print(round(e.subs(n, 100).evalf(), 10))
2.7048138294
```

Richardson extrapolation with 11 appropriately chosen terms gives a value that is accurate to the indicated precision:

```
>>> from sympy import E
>>> from sympy.series.acceleration import richardson
>>> print(round(richardson(e, n, 10, 20).evalf(), 10))
2.7182818285
>>> print(round(E.evalf(), 10))
2.7182818285
```

Another useful application is to speed up convergence of series. Computing 100 terms of the zeta(2) series 1/k**2 yields only two accurate digits:

```
>>> from sympy.abc import k, n
>>> from sympy import Sum
>>> A = Sum(k**-2, (k, 1, n))
>>> print(round(A.subs(n, 100).evalf(), 10))
1.6349839002
```

Richardson extrapolation performs much better:

```
>>> from sympy import pi
>>> print(round(richardson(A, n, 10, 20).evalf(), 10))
1.6449340668
>>> print(round(((pi**2)/6).evalf(), 10))      # Exact value
1.6449340668
```

sympy.series.acceleration.**shanks**(*A*, *k*, *n*, *m=1*)

Calculate an approximation for lim k->oo A(k) using the n-term Shanks transformation S(A)(n). With m > 1, calculate the m-fold recursive Shanks transformation S(S(…S(A)…))(n).

The Shanks transformation is useful for summing Taylor series that converge slowly near a pole or singularity, e.g. for log(2):

```
>>> from sympy.abc import k, n
>>> from sympy import Sum, Integer
>>> from sympy.series.acceleration import shanks
>>> A = Sum(Integer(-1)**(k+1) / k, (k, 1, n))
>>> print(round(A.subs(n, 100).doit().evalf(), 10))
0.6881721793
>>> print(round(shanks(A, n, 25).evalf(), 10))
0.6931396564
>>> print(round(shanks(A, n, 25, 5).evalf(), 10))
0.6931471806
```

The correct value is 0.6931471805599453094172321215.

Residues

TODO

Reference

sympy.series.residues.**residue**(*expr, x, x0*)

Finds the residue of `expr` at the point x=x0.

The residue is defined as the coefficient of `1/(x-x0)` in the power series expansion about `x=x0`.

Examples

```
>>> from sympy import Symbol, residue, sin
>>> x = Symbol(”x”)
>>> residue(1/x, x, 0)
1
>>> residue(1/x**2, x, 0)
0
>>> residue(2/sin(x), x, 0)
2
```

This function is essential for the Residue Theorem [1].

References

[R743]

Sequences

A sequence is a finite or infinite lazily evaluated list.

`sympy.series.sequences.`**`sequence`**`(`*seq, limits=None*`)`

Returns appropriate sequence object.

Explanation

If `seq` is a SymPy sequence, returns *`SeqPer`* (page 635) object otherwise returns *`SeqFormula`* (page 634) object.

Examples

```
>>> from sympy import sequence
>>> from sympy.abc import n
>>> sequence(n**2, (n, 0, 5))
SeqFormula(n**2, (n, 0, 5))
>>> sequence((1, 2, 3), (n, 0, 5))
SeqPer((1, 2, 3), (n, 0, 5))
```

See also:

`sympy.series.sequences.SeqPer` (page 635), *`sympy.series.sequences.SeqFormula`* (page 634)

Sequences Base

`class` `sympy.series.sequences.`**`SeqBase`**`(`**args*`)`

Base class for sequences

`coeff``(`*pt*`)`

Returns the coefficient at point pt

`coeff_mul``(`*other*`)`

Should be used when `other` is not a sequence. Should be defined to define custom behaviour.

Examples

```
>>> from sympy import SeqFormula
>>> from sympy.abc import n
>>> SeqFormula(n**2).coeff_mul(2)
SeqFormula(2*n**2, (n, 0, oo))
```

Notes

'*' defines multiplication of sequences with sequences only.

find_linear_recurrence(*n*, *d=None*, *gfvar=None*)

Finds the shortest linear recurrence that satisfies the first n terms of sequence of order $\leq$ `n/2` if possible. If `d` is specified, find shortest linear recurrence of order $\leq$ min(d, n/2) if possible. Returns list of coefficients `[b(1), b(2), ...]` corresponding to the recurrence relation `x(n) = b(1)*x(n-1) + b(2)*x(n-2) + ...` Returns `[]` if no recurrence is found. If gfvar is specified, also returns ordinary generating function as a function of gfvar.

Examples

```
>>> from sympy import sequence, sqrt, oo, lucas
>>> from sympy.abc import n, x, y
>>> sequence(n**2).find_linear_recurrence(10, 2)
[]
>>> sequence(n**2).find_linear_recurrence(10)
[3, -3, 1]
>>> sequence(2**n).find_linear_recurrence(10)
[2]
>>> sequence(23*n**4+91*n**2).find_linear_recurrence(10)
[5, -10, 10, -5, 1]
>>> sequence(sqrt(5)*(((1 + sqrt(5))/2)**n - (-(1 + sqrt(5))/2)**(-
↪n))/5).find_linear_recurrence(10)
[1, 1]
>>> sequence(x+y*(-2)**(-n), (n, 0, oo)).find_linear_recurrence(30)
[1/2, 1/2]
>>> sequence(3*5**n + 12).find_linear_recurrence(20,gfvar=x)
([6, -5], 3*(5 - 21*x)/((x - 1)*(5*x - 1)))
>>> sequence(lucas(n)).find_linear_recurrence(15,gfvar=x)
([1, 1], (x - 2)/(x**2 + x - 1))
```

property free_symbols

This method returns the symbols in the object, excluding those that take on a specific value (i.e. the dummy symbols).

Examples

```
>>> from sympy import SeqFormula
>>> from sympy.abc import n, m
>>> SeqFormula(m*n**2, (n, 0, 5)).free_symbols
{m}
```

property gen

Returns the generator for the sequence

property interval

The interval on which the sequence is defined

property length

Length of the sequence

property start

The starting point of the sequence. This point is included

property stop

The ending point of the sequence. This point is included

property variables

Returns a tuple of variables that are bounded

Elementary Sequences

class sympy.series.sequences.**SeqFormula**(*formula, limits=None*)

Represents sequence based on a formula.

Elements are generated using a formula.

Examples

```
>>> from sympy import SeqFormula, oo, Symbol
>>> n = Symbol('n')
>>> s = SeqFormula(n**2, (n, 0, 5))
>>> s.formula
n**2
```

For value at a particular point

```
>>> s.coeff(3)
9
```

supports slicing

```
>>> s[:]
[0, 1, 4, 9, 16, 25]
```

iterable

```
>>> list(s)
[0, 1, 4, 9, 16, 25]
```

sequence starts from negative infinity

```
>>> SeqFormula(n**2, (-oo, 0))[0:6]
[0, 1, 4, 9, 16, 25]
```

See also:

sympy.series.sequences.SeqPer (page 635)

coeff_mul(*coeff*)

See docstring of SeqBase.coeff_mul

class sympy.series.sequences.**SeqPer**(*periodical*, *limits=None*)

Represents a periodic sequence.

The elements are repeated after a given period.

Examples

```
>>> from sympy import SeqPer, oo
>>> from sympy.abc import k
```

```
>>> s = SeqPer((1, 2, 3), (0, 5))
>>> s.periodical
(1, 2, 3)
>>> s.period
3
```

For value at a particular point

```
>>> s.coeff(3)
1
```

supports slicing

```
>>> s[:]
[1, 2, 3, 1, 2, 3]
```

iterable

```
>>> list(s)
[1, 2, 3, 1, 2, 3]
```

sequence starts from negative infinity

```
>>> SeqPer((1, 2, 3), (-oo, 0))[0:6]
[1, 2, 3, 1, 2, 3]
```

Periodic formulas

```
>>> SeqPer((k, k**2, k**3), (k, 0, oo))[0:6]
[0, 1, 8, 3, 16, 125]
```

See also:

sympy.series.sequences.SeqFormula (page 634)

coeff_mul(*coeff*)

See docstring of SeqBase.coeff_mul

Singleton Sequences

class sympy.series.sequences.**EmptySequence**

Represents an empty sequence.

The empty sequence is also available as a singleton as `S.EmptySequence`.

Examples

```
>>> from sympy import EmptySequence, SeqPer
>>> from sympy.abc import x
>>> EmptySequence
EmptySequence
>>> SeqPer((1, 2), (x, 0, 10)) + EmptySequence
SeqPer((1, 2), (x, 0, 10))
>>> SeqPer((1, 2)) * EmptySequence
EmptySequence
>>> EmptySequence.coeff_mul(-1)
EmptySequence
```

coeff_mul(*coeff*)

See docstring of SeqBase.coeff_mul

Compound Sequences

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

Represents term-wise addition of sequences.

Rules:

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