The polylogarithm is a special case of the Lerch transcendent:

$$\mathrm{Li}_s(z) = z\Phi(z, s, 1).$$

Examples

For $z \in \{0, 1, -1\}$, the polylogarithm is automatically expressed using other functions:

```
>>> from sympy import polylog
>>> from sympy.abc import s
>>> polylog(s, 0)
0
>>> polylog(s, 1)
zeta(s)
>>> polylog(s, -1)
-dirichlet_eta(s)
```

If s is a negative integer, 0 or 1, the polylogarithm can be expressed using elementary functions. This can be done using `expand_func()`:

```
>>> from sympy import expand_func
>>> from sympy.abc import z
>>> expand_func(polylog(1, z))
-log(1 - z)
>>> expand_func(polylog(0, z))
z/(1 - z)
```

The derivative with respect to z can be computed in closed form:

```
>>> polylog(s, z).diff(z)
polylog(s - 1, z)/z
```

The polylogarithm can be expressed in terms of the lerch transcendent:

```
>>> from sympy import lerchphi
>>> polylog(s, z).rewrite(lerchphi)
z*lerchphi(z, s, 1)
```

See also:

zeta (page 514), *lerchphi* (page 517)

class sympy.functions.special.zeta_functions.**lerchphi**(**args*)

Lerch transcendent (Lerch phi function).

Explanation

For $\mathrm{Re}(a) > 0$, $|z| < 1$ and $s \in \mathbb{C}$, the Lerch transcendent is defined as

$$\Phi(z, s, a) = \sum_{n=0}^{\infty} \frac{z^n}{(n+a)^s},$$

where the standard branch of the argument is used for $n+a$, and by analytic continuation for other values of the parameters.

A commonly used related function is the Lerch zeta function, defined by

$$L(q, s, a) = \Phi(e^{2\pi i q}, s, a).$$

Analytic Continuation and Branching Behavior

It can be shown that

$$\Phi(z, s, a) = z\Phi(z, s, a+1) + a^{-s}.$$

This provides the analytic continuation to $\mathrm{Re}(a) \le 0$.

Assume now $\mathrm{Re}(a) > 0$. The integral representation

$$\Phi_0(z, s, a) = \int_0^\infty \frac{t^{s-1} e^{-at}}{1 - ze^{-t}} \frac{\mathrm{d}t}{\Gamma(s)}$$

provides an analytic continuation to $\mathbb{C} - [1, \infty)$. Finally, for $x \in (1, \infty)$ we find

$$\lim_{\epsilon \to 0^+} \Phi_0(x + i\epsilon, s, a) - \lim_{\epsilon \to 0^+} \Phi_0(x - i\epsilon, s, a) = \frac{2\pi i \log^{s-1} x}{x^a \Gamma(s)},$$

using the standard branch for both $\log x$ and $\log\log x$ (a branch of $\log\log x$ is needed to evaluate $\log x^{s-1}$). This concludes the analytic continuation. The Lerch transcendent is thus branched at $z \in \{0, 1, \infty\}$ and $a \in \mathbb{Z}_{\le 0}$. For fixed z, a outside these branch points, it is an entire function of s.

Examples

The Lerch transcendent is a fairly general function, for this reason it does not automatically evaluate to simpler functions. Use `expand_func()` to achieve this.

If $z = 1$, the Lerch transcendent reduces to the Hurwitz zeta function:

```
>>> from sympy import lerchphi, expand_func
>>> from sympy.abc import z, s, a
>>> expand_func(lerchphi(1, s, a))
zeta(s, a)
```

More generally, if z is a root of unity, the Lerch transcendent reduces to a sum of Hurwitz zeta functions:

```
>>> expand_func(lerchphi(-1, s, a))
zeta(s, a/2)/2**s - zeta(s, a/2 + 1/2)/2**s
```

If $a = 1$, the Lerch transcendent reduces to the polylogarithm:

```
>>> expand_func(lerchphi(z, s, 1))
polylog(s, z)/z
```

More generally, if a is rational, the Lerch transcendent reduces to a sum of polylogarithms:

```
>>> from sympy import S
>>> expand_func(lerchphi(z, s, S(1)/2))
2**(s - 1)*(polylog(s, sqrt(z))/sqrt(z) -
            polylog(s, sqrt(z)*exp_polar(I*pi))/sqrt(z))
>>> expand_func(lerchphi(z, s, S(3)/2))
-2**s/z + 2**(s - 1)*(polylog(s, sqrt(z))/sqrt(z) -
                      polylog(s, sqrt(z)*exp_polar(I*pi))/sqrt(z))/z
```

The derivatives with respect to z and a can be computed in closed form:

```
>>> lerchphi(z, s, a).diff(z)
(-a*lerchphi(z, s, a) + lerchphi(z, s - 1, a))/z
>>> lerchphi(z, s, a).diff(a)
-s*lerchphi(z, s + 1, a)
```

See also:

polylog (page 516), *zeta* (page 514)

References

[R418], [R419], [R420]

class sympy.functions.special.zeta_functions.**stieltjes**(*n*, *a=None*)

Represents Stieltjes constants, γ_k that occur in Laurent Series expansion of the Riemann zeta function.

Examples

```
>>> from sympy import stieltjes
>>> from sympy.abc import n, m
>>> stieltjes(n)
stieltjes(n)
```

The zero'th stieltjes constant:

```
>>> stieltjes(0)
EulerGamma
>>> stieltjes(0, 1)
EulerGamma
```

For generalized stieltjes constants:

```
>>> stieltjes(n, m)
stieltjes(n, m)
```

Constants are only defined for integers >= 0:

```
>>> stieltjes(-1)
zoo
```

References

[R421]

Hypergeometric Functions

class sympy.functions.special.hyper.**hyper**(*ap*, *bq*, *z*)

The generalized hypergeometric function is defined by a series where the ratios of successive terms are a rational function of the summation index. When convergent, it is continued analytically to the largest possible domain.

Explanation

The hypergeometric function depends on two vectors of parameters, called the numerator parameters a_p, and the denominator parameters b_q. It also has an argument z. The series definition is

$$ {}_pF_q\left(\begin{matrix} a_1, \cdots, a_p \\ b_1, \cdots, b_q \end{matrix} \middle| z \right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \cdots (a_p)_n}{(b_1)_n \cdots (b_q)_n} \frac{z^n}{n!}, $$

where $(a)_n = (a)(a+1)\cdots(a+n-1)$ denotes the rising factorial.

If one of the b_q is a non-positive integer then the series is undefined unless one of the a_p is a larger (i.e., smaller in magnitude) non-positive integer. If none of the b_q is a non-positive integer and one of the a_p is a non-positive integer, then the series reduces to a polynomial. To simplify the following discussion, we assume that none of the a_p or b_q is a non-positive integer. For more details, see the references.

The series converges for all z if $p \leq q$, and thus defines an entire single-valued function in this case. If $p = q+1$ the series converges for $|z| < 1$, and can be continued analytically into a half-plane. If $p > q+1$ the series is divergent for all z.

Please note the hypergeometric function constructor currently does *not* check if the parameters actually yield a well-defined function.

Examples

The parameters a_p and b_q can be passed as arbitrary iterables, for example:

```
>>> from sympy import hyper
>>> from sympy.abc import x, n, a
>>> hyper((1, 2, 3), [3, 4], x)
hyper((1, 2, 3), (3, 4), x)
```

There is also pretty printing (it looks better using Unicode):

```
>>> from sympy import pprint
>>> pprint(hyper((1, 2, 3), [3, 4], x), use_unicode=False)

 _
 |_  /1, 2, 3 |  \
 |   |        | x|
3  2 \  3, 4  |  /
```

The parameters must always be iterables, even if they are vectors of length one or zero:

```
>>> hyper((1, ), [], x)
hyper((1,), (), x)
```

But of course they may be variables (but if they depend on x then you should not expect much implemented functionality):

```
>>> hyper((n, a), (n**2,), x)
hyper((n, a), (n**2,), x)
```

The hypergeometric function generalizes many named special functions. The function `hyperexpand()` tries to express a hypergeometric function using named special functions. For example:

```
>>> from sympy import hyperexpand
>>> hyperexpand(hyper([], [], x))
exp(x)
```

You can also use `expand_func()`:

```
>>> from sympy import expand_func
>>> expand_func(x*hyper([1, 1], [2], -x))
log(x + 1)
```

More examples:

```
>>> from sympy import S
>>> hyperexpand(hyper([], [S(1)/2], -x**2/4))
cos(x)
>>> hyperexpand(x*hyper([S(1)/2, S(1)/2], [S(3)/2], x**2))
asin(x)
```

We can also sometimes `hyperexpand()` parametric functions:

```
>>> from sympy.abc import a
>>> hyperexpand(hyper([-a], [], x))
(1 - x)**a
```

See also:

sympy.simplify.hyperexpand (page 687), *gamma* (page 459), *meijerg* (page 522)

References

[R422], [R423]

property ap
Numerator parameters of the hypergeometric function.

property argument
Argument of the hypergeometric function.

property bq
Denominator parameters of the hypergeometric function.

property convergence_statement
Return a condition on z under which the series converges.

property eta
A quantity related to the convergence of the series.

property radius_of_convergence
Compute the radius of convergence of the defining series.

Explanation

Note that even if this is not `oo`, the function may still be evaluated outside of the radius of convergence by analytic continuation. But if this is zero, then the function is not actually defined anywhere else.

Examples

```
>>> from sympy import hyper
>>> from sympy.abc import z
>>> hyper((1, 2), [3], z).radius_of_convergence
1
>>> hyper((1, 2, 3), [4], z).radius_of_convergence
0
>>> hyper((1, 2), (3, 4), z).radius_of_convergence
oo
```

class sympy.functions.special.hyper.**meijerg**(**args*)
The Meijer G-function is defined by a Mellin-Barnes type integral that resembles an inverse Mellin transform. It generalizes the hypergeometric functions.

Explanation

The Meijer G-function depends on four sets of parameters. There are "*numerator parameters*" $a_1, \ldots, a_n$ and $a_{n+1}, \ldots, a_p$, and there are "*denominator parameters*" $b_1, \ldots, b_m$ and $b_{m+1}, \ldots, b_q$. Confusingly, it is traditionally denoted as follows (note the position of m, n, p, q, and how they relate to the lengths of the four parameter vectors):

$$G_{p,q}^{m,n} \left(\begin{matrix} a_1, \cdots, a_n & a_{n+1}, \cdots, a_p \\ b_1, \cdots, b_m & b_{m+1}, \cdots, b_q \end{matrix} \middle| z \right).$$

However, in SymPy the four parameter vectors are always available separately (see examples), so that there is no need to keep track of the decorating sub- and super-scripts on the G symbol.

The G function is defined as the following integral:

$$\frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^{m} \Gamma(b_j - s) \prod_{j=1}^{n} \Gamma(1 - a_j + s)}{\prod_{j=m+1}^{q} \Gamma(1 - b_j + s) \prod_{j=n+1}^{p} \Gamma(a_j - s)} z^s \mathrm{d}s,$$

where $\Gamma(z)$ is the gamma function. There are three possible contours which we will not describe in detail here (see the references). If the integral converges along more than one of them, the definitions agree. The contours all separate the poles of $\Gamma(1 - a_j + s)$ from the poles of $\Gamma(b_k - s)$, so in particular the G function is undefined if $a_j - b_k \in \mathbb{Z}_{>0}$ for some $j \leq n$ and $k \leq m$.

The conditions under which one of the contours yields a convergent integral are complicated and we do not state them here, see the references.

Please note currently the Meijer G-function constructor does *not* check any convergence conditions.

Examples

You can pass the parameters either as four separate vectors:

```
>>> from sympy import meijerg, Tuple, pprint
>>> from sympy.abc import x, a
>>> pprint(meijerg((1, 2), (a, 4), (5,), [], x), use_unicode=False)
 __1, 2 /1, 2  a, 4 |  \
/__     |           | x|
\_|4, 1 \ 5         |  /
```

Or as two nested vectors:

```
>>> pprint(meijerg([(1, 2), (3, 4)], ([5], Tuple()), x), use_
↪unicode=False)
 __1, 2 /1, 2  3, 4 |  \
/__     |           | x|
\_|4, 1 \ 5         |  /
```

As with the hypergeometric function, the parameters may be passed as arbitrary iterables. Vectors of length zero and one also have to be passed as iterables. The parameters need not be constants, but if they depend on the argument then not much implemented functionality should be expected.

All the subvectors of parameters are available:

```
>>> from sympy import pprint
>>> g = meijerg([1], [2], [3], [4], x)
>>> pprint(g, use_unicode=False)
 __1, 1 /1  2 |  \
/__     |     | x|
\_|2, 2 \3  4 |  /
>>> g.an
(1,)
>>> g.ap
(1, 2)
>>> g.aother
(2,)
>>> g.bm
(3,)
>>> g.bq
(3, 4)
>>> g.bother
(4,)
```

The Meijer G-function generalizes the hypergeometric functions. In some cases it can be expressed in terms of hypergeometric functions, using Slater's theorem. For example:

```
>>> from sympy import hyperexpand
>>> from sympy.abc import a, b, c
>>> hyperexpand(meijerg([a], [], [c], [b], x), allow_hyper=True)
x**c*gamma(-a + c + 1)*hyper((-a + c + 1,),
                             (-b + c + 1,), -x)/gamma(-b + c + 1)
```

Thus the Meijer G-function also subsumes many named functions as special cases. You can use `expand_func()` or `hyperexpand()` to (try to) rewrite a Meijer G-function in terms of named special functions. For example:

```
>>> from sympy import expand_func, S
>>> expand_func(meijerg([[],[]], [[0],[]], -x))
exp(x)
>>> hyperexpand(meijerg([[],[]], [[S(1)/2],[0]], (x/2)**2))
sin(x)/sqrt(pi)
```

See also:

hyper (page 520), *sympy.simplify.hyperexpand* (page 687)

References

[R424], [R425]

property an
First set of numerator parameters.

property aother
Second set of numerator parameters.

property ap
Combined numerator parameters.

property argument

Argument of the Meijer G-function.

property bm

First set of denominator parameters.

property bother

Second set of denominator parameters.

property bq

Combined denominator parameters.

property delta

A quantity related to the convergence region of the integral, c.f. references.

get_period()

Return a number P such that $G(x * exp(I * P)) == G(x)$.

Examples

```
>>> from sympy import meijerg, pi, S
>>> from sympy.abc import z
```

```
>>> meijerg([1], [], [], [], z).get_period()
2*pi
>>> meijerg([pi], [], [], [], z).get_period()
oo
>>> meijerg([1, 2], [], [], [], z).get_period()
oo
>>> meijerg([1,1], [2], [1, S(1)/2, S(1)/3], [1], z).get_period()
12*pi
```

integrand(s)

Get the defining integrand D(s).

property is_number

Returns true if expression has numeric data only.

property nu

A quantity related to the convergence region of the integral, c.f. references.

class sympy.functions.special.hyper.**appellf1**(*a*, *b1*, *b2*, *c*, *x*, *y*)

This is the Appell hypergeometric function of two variables as:

$$F_1(a, b_1, b_2, c, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a)_{m+n}(b_1)_m(b_2)_n}{(c)_{m+n}} \frac{x^m y^n}{m!n!}.$$

Examples

```
>>> from sympy import appellf1, symbols
>>> x, y, a, b1, b2, c = symbols('x y a b1 b2 c')
>>> appellf1(2., 1., 6., 4., 5., 6.)
0.0063339426292673
>>> appellf1(12., 12., 6., 4., 0.5, 0.12)
172870711.659936
>>> appellf1(40, 2, 6, 4, 15, 60)
appellf1(40, 2, 6, 4, 15, 60)
>>> appellf1(20., 12., 10., 3., 0.5, 0.12)
15605338197184.4
>>> appellf1(40, 2, 6, 4, x, y)
appellf1(40, 2, 6, 4, x, y)
>>> appellf1(a, b1, b2, c, x, y)
appellf1(a, b1, b2, c, x, y)
```

References

[R426], [R427]

Elliptic integrals

class sympy.functions.special.elliptic_integrals.**elliptic_k**(*m*)

The complete elliptic integral of the first kind, defined by

$$K(m) = F\left(\tfrac{\pi}{2}\middle|m\right)$$

where $F\left(z|m\right)$ is the Legendre incomplete elliptic integral of the first kind.

Explanation

The function $K(m)$ is a single-valued function on the complex plane with branch cut along the interval $(1, \infty)$.

Note that our notation defines the incomplete elliptic integral in terms of the parameter m instead of the elliptic modulus (eccentricity) k. In this case, the parameter m is defined as $m = k^2$.

Examples

```
>>> from sympy import elliptic_k, I
>>> from sympy.abc import m
>>> elliptic_k(0)
pi/2
>>> elliptic_k(1.0 + I)
1.50923695405127 + 0.625146415202697*I
>>> elliptic_k(m).series(n=3)
pi/2 + pi*m/8 + 9*pi*m**2/128 + O(m**3)
```

See also:

elliptic_f (page 527)

References

[R428], [R429]

class sympy.functions.special.elliptic_integrals.**elliptic_f**(*z*, *m*)

The Legendre incomplete elliptic integral of the first kind, defined by

$$F\left(z|m\right) = \int_0^z \frac{dt}{\sqrt{1 - m\sin^2 t}}$$

Explanation

This function reduces to a complete elliptic integral of the first kind, $K(m)$, when $z = \pi/2$.

Note that our notation defines the incomplete elliptic integral in terms of the parameter m instead of the elliptic modulus (eccentricity) k. In this case, the parameter m is defined as $m = k^2$.

Examples

```
>>> from sympy import elliptic_f, I
>>> from sympy.abc import z, m
>>> elliptic_f(z, m).series(z)
z + z**5*(3*m**2/40 - m/30) + m*z**3/6 + O(z**6)
>>> elliptic_f(3.0 + I/2, 1.0 + I)
2.909449841483 + 1.74720545502474*I
```

See also:

elliptic_k (page 526)

References

[R430], [R431]

class sympy.functions.special.elliptic_integrals.**elliptic_e**(*m*, *z=None*)

Called with two arguments z and m, evaluates the incomplete elliptic integral of the second kind, defined by

$$E\left(z|m\right) = \int_0^z \sqrt{1 - m\sin^2 t}dt$$

Called with a single argument m, evaluates the Legendre complete elliptic integral of the second kind

$$E(m) = E\left(\tfrac{\pi}{2}|m\right)$$

Explanation

The function $E(m)$ is a single-valued function on the complex plane with branch cut along the interval $(1, \infty)$.

Note that our notation defines the incomplete elliptic integral in terms of the parameter m instead of the elliptic modulus (eccentricity) k. In this case, the parameter m is defined as $m = k^2$.

Examples

```
>>> from sympy import elliptic_e, I
>>> from sympy.abc import z, m
>>> elliptic_e(z, m).series(z)
z + z**5*(-m**2/40 + m/30) - m*z**3/6 + O(z**6)
>>> elliptic_e(m).series(n=4)
pi/2 - pi*m/8 - 3*pi*m**2/128 - 5*pi*m**3/512 + O(m**4)
>>> elliptic_e(1 + I, 2 - I/2).n()
1.55203744279187 + 0.290764986058437*I
>>> elliptic_e(0)
pi/2
>>> elliptic_e(2.0 - I)
0.991052601328069 + 0.81879421395609*I
```

References

[R432], [R433], [R434]

class sympy.functions.special.elliptic_integrals.**elliptic_pi**(*n*, *m*, *z=None*)

Called with three arguments n, z and m, evaluates the Legendre incomplete elliptic integral of the third kind, defined by

$$\Pi\left(n; z\middle| m\right) = \int_0^z \frac{dt}{\left(1 - n\sin^2 t\right)\sqrt{1 - m\sin^2 t}}$$

Called with two arguments n and m, evaluates the complete elliptic integral of the third kind:

$$\Pi\left(n\middle| m\right) = \Pi\left(n; \tfrac{\pi}{2}\middle| m\right)$$

Explanation

Note that our notation defines the incomplete elliptic integral in terms of the parameter m instead of the elliptic modulus (eccentricity) k. In this case, the parameter m is defined as $m = k^2$.

Examples

```
>>> from sympy import elliptic_pi, I
>>> from sympy.abc import z, n, m
>>> elliptic_pi(n, z, m).series(z, n=4)
z + z**3*(m/6 + n/3) + O(z**4)
>>> elliptic_pi(0.5 + I, 1.0 - I, 1.2)
2.50232379629182 - 0.760939574180767*I
>>> elliptic_pi(0, 0)
pi/2
>>> elliptic_pi(1.0 - I/3, 2.0 + I)
3.29136443417283 + 0.32555634906645*I
```

References

[R435], [R436], [R437]

Mathieu Functions

class sympy.functions.special.mathieu_functions.**MathieuBase**(**args*)

Abstract base class for Mathieu functions.

This class is meant to reduce code duplication.

class sympy.functions.special.mathieu_functions.**mathieus**(*a*, *q*, *z*)

The Mathieu Sine function $S(a,q,z)$.

Explanation

This function is one solution of the Mathieu differential equation:

$$y(x)'' + (a - 2q\cos(2x))y(x) = 0$$

The other solution is the Mathieu Cosine function.

Examples

```
>>> from sympy import diff, mathieus
>>> from sympy.abc import a, q, z
```

```
>>> mathieus(a, q, z)
mathieus(a, q, z)
```

```
>>> mathieus(a, 0, z)
sin(sqrt(a)*z)
```

```
>>> diff(mathieus(a, q, z), z)
mathieusprime(a, q, z)
```

See also:

mathieuc **(page 530)**
Mathieu cosine function.

mathieusprime **(page 531)**
Derivative of Mathieu sine function.

mathieucprime **(page 531)**
Derivative of Mathieu cosine function.

References

[R438], [R439], [R440], [R441]

class sympy.functions.special.mathieu_functions.**mathieuc**(*a*, *q*, *z*)
The Mathieu Cosine function $C(a, q, z)$.

Explanation

This function is one solution of the Mathieu differential equation:

$$y(x)'' + (a - 2q\cos(2x))y(x) = 0$$

The other solution is the Mathieu Sine function.

Examples

```
>>> from sympy import diff, mathieuc
>>> from sympy.abc import a, q, z
```

```
>>> mathieuc(a, q, z)
mathieuc(a, q, z)
```

```
>>> mathieuc(a, 0, z)
cos(sqrt(a)*z)
```

```
>>> diff(mathieuc(a, q, z), z)
mathieucprime(a, q, z)
```

See also:

mathieus **(page 529)**
Mathieu sine function

mathieusprime **(page 531)**
Derivative of Mathieu sine function

mathieucprime **(page 531)**
Derivative of Mathieu cosine function

References

[R442], [R443], [R444], [R445]

class sympy.functions.special.mathieu_functions.**mathieusprime**(*a*, *q*, *z*)

The derivative $S'(a, q, z)$ of the Mathieu Sine function.

Explanation

This function is one solution of the Mathieu differential equation:

$$y(x)'' + (a - 2q\cos(2x))y(x) = 0$$

The other solution is the Mathieu Cosine function.

Examples

```
>>> from sympy import diff, mathieusprime
>>> from sympy.abc import a, q, z
```

```
>>> mathieusprime(a, q, z)
mathieusprime(a, q, z)
```

```
>>> mathieusprime(a, 0, z)
sqrt(a)*cos(sqrt(a)*z)
```

```
>>> diff(mathieusprime(a, q, z), z)
(-a + 2*q*cos(2*z))*mathieus(a, q, z)
```

See also:

mathieus **(page 529)**
Mathieu sine function

mathieuc **(page 530)**
Mathieu cosine function

mathieucprime **(page 531)**
Derivative of Mathieu cosine function

References

[R446], [R447], [R448], [R449]

class sympy.functions.special.mathieu_functions.**mathieucprime**(*a*, *q*, *z*)

The derivative $C'(a, q, z)$ of the Mathieu Cosine function.

Explanation

This function is one solution of the Mathieu differential equation:

$$y(x)'' + (a - 2q\cos(2x))y(x) = 0$$

The other solution is the Mathieu Sine function.

Examples

```
>>> from sympy import diff, mathieucprime
>>> from sympy.abc import a, q, z
```

```
>>> mathieucprime(a, q, z)
mathieucprime(a, q, z)
```

```
>>> mathieucprime(a, 0, z)
-sqrt(a)*sin(sqrt(a)*z)
```

```
>>> diff(mathieucprime(a, q, z), z)
(-a + 2*q*cos(2*z))*mathieuc(a, q, z)
```

See also:

mathieus **(page 529)**
Mathieu sine function

mathieuc **(page 530)**
Mathieu cosine function

mathieusprime **(page 531)**
Derivative of Mathieu sine function

References

[R450], [R451], [R452], [R453]

Orthogonal Polynomials

This module mainly implements special orthogonal polynomials.

See also functions.combinatorial.numbers which contains some combinatorial polynomials.

Jacobi Polynomials

class sympy.functions.special.polynomials.**jacobi**(*n*, *a*, *b*, *x*)

Jacobi polynomial $P_n^{(\alpha,\beta)}(x)$.

Explanation

`jacobi(n, alpha, beta, x)` gives the nth Jacobi polynomial in x, $P_n^{(\alpha,\beta)}(x)$.

The Jacobi polynomials are orthogonal on $[-1, 1]$ with respect to the weight $(1-x)^\alpha (1+x)^\beta$.

Examples

```
>>> from sympy import jacobi, S, conjugate, diff
>>> from sympy.abc import a, b, n, x
```

```
>>> jacobi(0, a, b, x)
1
>>> jacobi(1, a, b, x)
a/2 - b/2 + x*(a/2 + b/2 + 1)
>>> jacobi(2, a, b, x)
a**2/8 - a*b/4 - a/8 + b**2/8 - b/8 + x**2*(a**2/8 + a*b/4 + 7*a/8 +␣
↪b**2/8 + 7*b/8 + 3/2) + x*(a**2/4 + 3*a/4 - b**2/4 - 3*b/4) - 1/2
```

```
>>> jacobi(n, a, b, x)
jacobi(n, a, b, x)
```

```
>>> jacobi(n, a, a, x)
RisingFactorial(a + 1, n)*gegenbauer(n,
    a + 1/2, x)/RisingFactorial(2*a + 1, n)
```

```
>>> jacobi(n, 0, 0, x)
legendre(n, x)
```

```
>>> jacobi(n, S(1)/2, S(1)/2, x)
RisingFactorial(3/2, n)*chebyshevu(n, x)/factorial(n + 1)
```

```
>>> jacobi(n, -S(1)/2, -S(1)/2, x)
RisingFactorial(1/2, n)*chebyshevt(n, x)/factorial(n)
```

```
>>> jacobi(n, a, b, -x)
(-1)**n*jacobi(n, b, a, x)
```

```
>>> jacobi(n, a, b, 0)
gamma(a + n + 1)*hyper((-b - n, -n), (a + 1,), -1)/
↪(2**n*factorial(n)*gamma(a + 1))
>>> jacobi(n, a, b, 1)
RisingFactorial(a + 1, n)/factorial(n)
```

```
>>> conjugate(jacobi(n, a, b, x))
jacobi(n, conjugate(a), conjugate(b), conjugate(x))
```

```
>>> diff(jacobi(n,a,b,x), x)
(a/2 + b/2 + n/2 + 1/2)*jacobi(n - 1, a + 1, b + 1, x)
```

See also:

gegenbauer (page 535), *chebyshevt_root* (page 539), *chebyshevu* (page 538), *chebyshevu_root* (page 539), *legendre* (page 540), *assoc_legendre* (page 540), *hermite* (page 541), *laguerre* (page 542), *assoc_laguerre* (page 543), *sympy.polys.orthopolys.jacobi_poly* (page 2440), *sympy.polys.orthopolys.gegenbauer_poly* (page 2440), *sympy.polys.orthopolys.chebyshevt_poly* (page 2439), *sympy.polys.orthopolys.chebyshevu_poly* (page 2439), *sympy.polys.orthopolys.hermite_poly* (page 2440), *sympy.polys.orthopolys.legendre_poly* (page 2440), *sympy.polys.orthopolys.laguerre_poly* (page 2441)

References

[R454], [R455], [R456]

sympy.functions.special.polynomials.**jacobi_normalized**(*n*, *a*, *b*, *x*)

Jacobi polynomial $P_n^{(\alpha,\beta)}(x)$.

Parameters

n : integer degree of polynomial

a : alpha value

b : beta value

x : symbol

Explanation

`jacobi_normalized(n, alpha, beta, x)` gives the nth Jacobi polynomial in x, $P_n^{(\alpha,\beta)}(x)$.

The Jacobi polynomials are orthogonal on $[-1, 1]$ with respect to the weight $(1-x)^\alpha (1+x)^\beta$.

This functions returns the polynomials normilzed:

$$\int_{-1}^{1} P_m^{(\alpha,\beta)}(x) P_n^{(\alpha,\beta)}(x) (1-x)^\alpha (1+x)^\beta \mathrm{d}x = \delta_{m,n}$$

Examples

```
>>> from sympy import jacobi_normalized
>>> from sympy.abc import n,a,b,x
```

```
>>> jacobi_normalized(n, a, b, x)
jacobi(n, a, b, x)/sqrt(2**(a + b + 1)*gamma(a + n + 1)*gamma(b + n + 1)/
↪((a + b + 2*n + 1)*factorial(n)*gamma(a + b + n + 1)))
```

See also:

gegenbauer (page 535), *chebyshevt_root* (page 539), *chebyshevu* (page 538), *chebyshevu_root* (page 539), *legendre* (page 540), *assoc_legendre* (page 540), *hermite* (page 541), *laguerre* (page 542), *assoc_laguerre* (page 543), *sympy.polys.orthopolys.jacobi_poly* (page 2440), *sympy.polys.orthopolys.gegenbauer_poly* (page 2440), *sympy.polys.orthopolys.chebyshevt_poly* (page 2439), *sympy.polys.orthopolys.chebyshevu_poly* (page 2439), *sympy.polys.orthopolys.hermite_poly* (page 2440), *sympy.polys.orthopolys.legendre_poly* (page 2440), *sympy.polys.orthopolys.laguerre_poly* (page 2441)

References

[R457], [R458], [R459]

Gegenbauer Polynomials

class sympy.functions.special.polynomials.**gegenbauer**(*n*, *a*, *x*)

Gegenbauer polynomial $C_n^{(\alpha)}(x)$.

Explanation

`gegenbauer(n, alpha, x)` gives the nth Gegenbauer polynomial in x, $C_n^{(\alpha)}(x)$.

The Gegenbauer polynomials are orthogonal on $[-1, 1]$ with respect to the weight $\left(1-x^2\right)^{\alpha-\frac{1}{2}}$.

Examples

```
>>> from sympy import gegenbauer, conjugate, diff
>>> from sympy.abc import n,a,x
>>> gegenbauer(0, a, x)
1
>>> gegenbauer(1, a, x)
2*a*x
>>> gegenbauer(2, a, x)
-a + x**2*(2*a**2 + 2*a)
>>> gegenbauer(3, a, x)
x**3*(4*a**3/3 + 4*a**2 + 8*a/3) + x*(-2*a**2 - 2*a)
```

```
>>> gegenbauer(n, a, x)
gegenbauer(n, a, x)
>>> gegenbauer(n, a, -x)
(-1)**n*gegenbauer(n, a, x)
```

```
>>> gegenbauer(n, a, 0)
2**n*sqrt(pi)*gamma(a + n/2)/(gamma(a)*gamma(1/2 - n/2)*gamma(n + 1))
>>> gegenbauer(n, a, 1)
gamma(2*a + n)/(gamma(2*a)*gamma(n + 1))
```

```
>>> conjugate(gegenbauer(n, a, x))
gegenbauer(n, conjugate(a), conjugate(x))
```

```
>>> diff(gegenbauer(n, a, x), x)
2*a*gegenbauer(n - 1, a + 1, x)
```

See also:

jacobi (page 533), *chebyshevt_root* (page 539), *chebyshevu* (page 538), *chebyshevu_root* (page 539), *legendre* (page 540), *assoc_legendre* (page 540), *hermite* (page 541), *laguerre* (page 542), *assoc_laguerre* (page 543), *sympy.polys.orthopolys.jacobi_poly* (page 2440), *sympy.polys.orthopolys.gegenbauer_poly* (page 2440), *sympy.polys.orthopolys.chebyshevt_poly* (page 2439), *sympy.polys.orthopolys.chebyshevu_poly* (page 2439), *sympy.polys.orthopolys.hermite_poly* (page 2440), *sympy.polys.orthopolys.legendre_poly* (page 2440), *sympy.polys.orthopolys.laguerre_poly* (page 2441)

References

[R460], [R461], [R462]

Chebyshev Polynomials

class sympy.functions.special.polynomials.**chebyshevt**(*n*, *x*)

Chebyshev polynomial of the first kind, $T_n(x)$.

Explanation

`chebyshevt(n, x)` gives the nth Chebyshev polynomial (of the first kind) in x, $T_n(x)$.

The Chebyshev polynomials of the first kind are orthogonal on $[-1, 1]$ with respect to the weight $\frac{1}{\sqrt{1-x^2}}$.

Examples

```
>>> from sympy import chebyshevt, diff
>>> from sympy.abc import n,x
>>> chebyshevt(0, x)
1
>>> chebyshevt(1, x)
x
>>> chebyshevt(2, x)
2*x**2 - 1
```

```
>>> chebyshevt(n, x)
chebyshevt(n, x)
>>> chebyshevt(n, -x)
(-1)**n*chebyshevt(n, x)
>>> chebyshevt(-n, x)
chebyshevt(n, x)
```

```
>>> chebyshevt(n, 0)
cos(pi*n/2)
>>> chebyshevt(n, -1)
(-1)**n
```

```
>>> diff(chebyshevt(n, x), x)
n*chebyshevu(n - 1, x)
```


See also:

jacobi (page 533), *gegenbauer* (page 535), *chebyshevt_root* (page 539), *chebyshevu* (page 538), *chebyshevu_root* (page 539), *legendre* (page 540), *assoc_legendre* (page 540), *hermite* (page 541), *laguerre* (page 542), *assoc_laguerre* (page 543), *sympy.polys.orthopolys.jacobi_poly* (page 2440), *sympy.polys.orthopolys.gegenbauer_poly* (page 2440), *sympy.polys.orthopolys.chebyshevt_poly* (page 2439), *sympy.polys.orthopolys.chebyshevu_poly* (page 2439), *sympy.polys.orthopolys.hermite_poly* (page 2440), *sympy.polys.orthopolys.legendre_poly* (page 2440), *sympy.polys.orthopolys.laguerre_poly* (page 2441)

References

[R463], [R464], [R465], [R466], [R467]

class sympy.functions.special.polynomials.**chebyshevu**(*n*, *x*)

Chebyshev polynomial of the second kind, $U_n(x)$.

Explanation

`chebyshevu(n, x)` gives the nth Chebyshev polynomial of the second kind in x, $U_n(x)$.

The Chebyshev polynomials of the second kind are orthogonal on $[-1, 1]$ with respect to the weight $\sqrt{1-x^2}$.

Examples

```
>>> from sympy import chebyshevu, diff
>>> from sympy.abc import n,x
>>> chebyshevu(0, x)
1
>>> chebyshevu(1, x)
2*x
>>> chebyshevu(2, x)
4*x**2 - 1
```

```
>>> chebyshevu(n, x)
chebyshevu(n, x)
>>> chebyshevu(n, -x)
(-1)**n*chebyshevu(n, x)
>>> chebyshevu(-n, x)
-chebyshevu(n - 2, x)
```

```
>>> chebyshevu(n, 0)
cos(pi*n/2)
>>> chebyshevu(n, 1)
n + 1
```

```
>>> diff(chebyshevu(n, x), x)
(-x*chebyshevu(n, x) + (n + 1)*chebyshevt(n + 1, x))/(x**2 - 1)
```

See also:

jacobi (page 533), *gegenbauer* (page 535), *chebyshevt* (page 536), *chebyshevt_root* (page 539), *chebyshevu_root* (page 539), *legendre* (page 540), *assoc_legendre* (page 540), *hermite* (page 541), *laguerre* (page 542), *assoc_laguerre* (page 543), *sympy.polys.orthopolys.jacobi_poly* (page 2440), *sympy.polys.orthopolys.gegenbauer_poly* (page 2440), *sympy.polys.orthopolys.chebyshevt_poly* (page 2439), *sympy.polys.orthopolys.chebyshevu_poly* (page 2439), *sympy.polys.orthopolys.hermite_poly* (page 2440), *sympy.polys.orthopolys.legendre_poly* (page 2440), *sympy.polys.orthopolys.laguerre_poly* (page 2441)

References

[R468], [R469], [R470], [R471], [R472]

class sympy.functions.special.polynomials.**chebyshevt_root**(*n*, *k*)

chebyshev_root(n, k) returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the first kind; that is, if $0 \leq k < n$, chebyshevt(n, chebyshevt_root(n, k)) == 0.

Examples

```
>>> from sympy import chebyshevt, chebyshevt_root
>>> chebyshevt_root(3, 2)
-sqrt(3)/2
>>> chebyshevt(3, chebyshevt_root(3, 2))
0
```

See also:

jacobi (page 533), *gegenbauer* (page 535), *chebyshevt* (page 536), *chebyshevu* (page 538), *chebyshevu_root* (page 539), *legendre* (page 540), *assoc_legendre* (page 540), *hermite* (page 541), *laguerre* (page 542), *assoc_laguerre* (page 543), *sympy.polys.orthopolys.jacobi_poly* (page 2440), *sympy.polys.orthopolys.gegenbauer_poly* (page 2440), *sympy.polys.orthopolys.chebyshevt_poly* (page 2439), *sympy.polys.orthopolys.chebyshevu_poly* (page 2439), *sympy.polys.orthopolys.hermite_poly* (page 2440), *sympy.polys.orthopolys.legendre_poly* (page 2440), *sympy.polys.orthopolys.laguerre_poly* (page 2441)

class sympy.functions.special.polynomials.**chebyshevu_root**(*n*, *k*)

chebyshevu_root(n, k) returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the second kind; that is, if $0 \leq k < n$, chebyshevu(n, chebyshevu_root(n, k)) == 0.

Examples

```
>>> from sympy import chebyshevu, chebyshevu_root
>>> chebyshevu_root(3, 2)
-sqrt(2)/2
>>> chebyshevu(3, chebyshevu_root(3, 2))
0
```

See also:

chebyshevt (page 536), *chebyshevt_root* (page 539), *chebyshevu* (page 538), *legendre* (page 540), *assoc_legendre* (page 540), *hermite* (page 541), *laguerre* (page 542), *assoc_laguerre* (page 543), *sympy.polys.orthopolys.jacobi_poly* (page 2440), *sympy.polys.orthopolys.gegenbauer_poly* (page 2440), *sympy.polys.orthopolys.chebyshevt_poly* (page 2439), *sympy.polys.orthopolys.chebyshevu_poly* (page 2439), *sympy.polys.orthopolys.hermite_poly* (page 2440), *sympy.polys.orthopolys.legendre_poly* (page 2440), *sympy.polys.orthopolys.laguerre_poly* (page 2441)

Legendre Polynomials

class sympy.functions.special.polynomials.**legendre**(*n*, *x*)

legendre(n, x) gives the nth Legendre polynomial of x, $P_n(x)$

Explanation

The Legendre polynomials are orthogonal on $[-1, 1]$ with respect to the constant weight 1. They satisfy $P_n(1) = 1$ for all n; further, P_n is odd for odd n and even for even n.

Examples

```
>>> from sympy import legendre, diff
>>> from sympy.abc import x, n
>>> legendre(0, x)
1
>>> legendre(1, x)
x
>>> legendre(2, x)
3*x**2/2 - 1/2
>>> legendre(n, x)
legendre(n, x)
>>> diff(legendre(n,x), x)
n*(x*legendre(n, x) - legendre(n - 1, x))/(x**2 - 1)
```

See also:

jacobi (page 533), *gegenbauer* (page 535), *chebyshevt* (page 536), *chebyshevt_root* (page 539), *chebyshevu* (page 538), *chebyshevu_root* (page 539), *assoc_legendre* (page 540), *hermite* (page 541), *laguerre* (page 542), *assoc_laguerre* (page 543), *sympy.polys.orthopolys.jacobi_poly* (page 2440), *sympy.polys.orthopolys.gegenbauer_poly* (page 2440), *sympy.polys.orthopolys.chebyshevt_poly* (page 2439), *sympy.polys.orthopolys.chebyshevu_poly* (page 2439), *sympy.polys.orthopolys.hermite_poly* (page 2440), *sympy.polys.orthopolys.legendre_poly* (page 2440), *sympy.polys.orthopolys.laguerre_poly* (page 2441)

References

[R473], [R474], [R475], [R476]

class sympy.functions.special.polynomials.**assoc_legendre**(*n*, *m*, *x*)

assoc_legendre(n, m, x) gives $P_n^m(x)$, where n and m are the degree and order or an expression which is related to the nth order Legendre polynomial, $P_n(x)$ in the following manner:

$$P_n^m(x) = (-1)^m (1 - x^2)^{\frac{m}{2}} \frac{\mathrm{d}^m P_n(x)}{\mathrm{d}x^m}$$

Explanation

Associated Legendre polynomials are orthogonal on $[-1, 1]$ with:

- weight $= 1$ for the same m and different n.
- weight $= \frac{1}{1-x^2}$ for the same n and different m.

Examples

```
>>> from sympy import assoc_legendre
>>> from sympy.abc import x, m, n
>>> assoc_legendre(0,0, x)
1
>>> assoc_legendre(1,0, x)
x
>>> assoc_legendre(1,1, x)
-sqrt(1 - x**2)
>>> assoc_legendre(n,m,x)
assoc_legendre(n, m, x)
```

See also:

jacobi (page 533), *gegenbauer* (page 535), *chebyshevt* (page 536), *chebyshevt_root* (page 539), *chebyshevu* (page 538), *chebyshevu_root* (page 539), *legendre* (page 540), *hermite* (page 541), *laguerre* (page 542), *assoc_laguerre* (page 543), *sympy.polys.orthopolys.jacobi_poly* (page 2440), *sympy.polys.orthopolys.gegenbauer_poly* (page 2440), *sympy.polys.orthopolys.chebyshevt_poly* (page 2439), *sympy.polys.orthopolys.chebyshevu_poly* (page 2439), *sympy.polys.orthopolys.hermite_poly* (page 2440), *sympy.polys.orthopolys.legendre_poly* (page 2440), *sympy.polys.orthopolys.laguerre_poly* (page 2441)

References

[R477], [R478], [R479], [R480]

Hermite Polynomials

class sympy.functions.special.polynomials.**hermite**(*n*, *x*)

`hermite(n, x)` gives the nth Hermite polynomial in x, $H_n(x)$

Explanation

The Hermite polynomials are orthogonal on $(-\infty, \infty)$ with respect to the weight $\exp\left(-x^2\right)$.

Examples

```
>>> from sympy import hermite, diff
>>> from sympy.abc import x, n
>>> hermite(0, x)
1
>>> hermite(1, x)
2*x
>>> hermite(2, x)
4*x**2 - 2
>>> hermite(n, x)
hermite(n, x)
>>> diff(hermite(n,x), x)
2*n*hermite(n - 1, x)
>>> hermite(n, -x)
(-1)**n*hermite(n, x)
```

See also:

jacobi (page 533), *gegenbauer* (page 535), *chebyshevt* (page 536), *chebyshevt_root* (page 539), *chebyshevu* (page 538), *chebyshevu_root* (page 539), *legendre* (page 540), *assoc_legendre* (page 540), *laguerre* (page 542), *assoc_laguerre* (page 543), *sympy.polys.orthopolys.jacobi_poly* (page 2440), *sympy.polys.orthopolys.gegenbauer_poly* (page 2440), *sympy.polys.orthopolys.chebyshevt_poly* (page 2439), *sympy.polys.orthopolys.chebyshevu_poly* (page 2439), *sympy.polys.orthopolys.hermite_poly* (page 2440), *sympy.polys.orthopolys.legendre_poly* (page 2440), *sympy.polys.orthopolys.laguerre_poly* (page 2441)

References

[R481], [R482], [R483]

Laguerre Polynomials

class sympy.functions.special.polynomials.**laguerre**(*n*, *x*)

Returns the nth Laguerre polynomial in x, $L_n(x)$.

Parameters

n : int

Degree of Laguerre polynomial. Must be $n \geq 0$.

Examples

```
>>> from sympy import laguerre, diff
>>> from sympy.abc import x, n
>>> laguerre(0, x)
1
>>> laguerre(1, x)
1 - x
>>> laguerre(2, x)
x**2/2 - 2*x + 1
>>> laguerre(3, x)
-x**3/6 + 3*x**2/2 - 3*x + 1
```

```
>>> laguerre(n, x)
laguerre(n, x)
```

```
>>> diff(laguerre(n, x), x)
-assoc_laguerre(n - 1, 1, x)
```

See also:

jacobi (page 533), *gegenbauer* (page 535), *chebyshevt* (page 536), *chebyshevt_root* (page 539), *chebyshevu* (page 538), *chebyshevu_root* (page 539), *legendre* (page 540), *assoc_legendre* (page 540), *hermite* (page 541), *assoc_laguerre* (page 543), *sympy.polys.orthopolys.jacobi_poly* (page 2440), *sympy.polys.orthopolys.gegenbauer_poly* (page 2440), *sympy.polys.orthopolys.chebyshevt_poly* (page 2439), *sympy.polys.orthopolys.chebyshevu_poly* (page 2439), *sympy.polys.orthopolys.hermite_poly* (page 2440), *sympy.polys.orthopolys.legendre_poly* (page 2440), *sympy.polys.orthopolys.laguerre_poly* (page 2441)

References

[R484], [R485], [R486], [R487]

class sympy.functions.special.polynomials.**assoc_laguerre**(*n*, *alpha*, *x*)

Returns the nth generalized Laguerre polynomial in x, $L_n(x)$.

Parameters

n : int

Degree of Laguerre polynomial. Must be $n \geq 0$.

alpha : Expr

Arbitrary expression. For `alpha=0` regular Laguerre polynomials will be generated.

Examples

```
>>> from sympy import assoc_laguerre, diff
>>> from sympy.abc import x, n, a
>>> assoc_laguerre(0, a, x)
1
>>> assoc_laguerre(1, a, x)
a - x + 1
>>> assoc_laguerre(2, a, x)
a**2/2 + 3*a/2 + x**2/2 + x*(-a - 2) + 1
>>> assoc_laguerre(3, a, x)
a**3/6 + a**2 + 11*a/6 - x**3/6 + x**2*(a/2 + 3/2) +
    x*(-a**2/2 - 5*a/2 - 3) + 1
```

```
>>> assoc_laguerre(n, a, 0)
binomial(a + n, a)
```

```
>>> assoc_laguerre(n, a, x)
assoc_laguerre(n, a, x)
```

```
>>> assoc_laguerre(n, 0, x)
laguerre(n, x)
```

```
>>> diff(assoc_laguerre(n, a, x), x)
-assoc_laguerre(n - 1, a + 1, x)
```

```
>>> diff(assoc_laguerre(n, a, x), a)
Sum(assoc_laguerre(_k, a, x)/(-a + n), (_k, 0, n - 1))
```

See also:

jacobi (page 533), *gegenbauer* (page 535), *chebyshevt* (page 536), *chebyshevt_root* (page 539), *chebyshevu* (page 538), *chebyshevu_root* (page 539), *legendre* (page 540), *assoc_legendre* (page 540), *hermite* (page 541), *laguerre* (page 542), *sympy.polys.orthopolys.jacobi_poly* (page 2440), *sympy.polys.orthopolys.gegenbauer_poly* (page 2440), *sympy.polys.orthopolys.chebyshevt_poly* (page 2439), *sympy.polys.orthopolys.chebyshevu_poly* (page 2439), *sympy.polys.orthopolys.hermite_poly* (page 2440), *sympy.polys.orthopolys.legendre_poly* (page 2440), *sympy.polys.orthopolys.laguerre_poly* (page 2441)

References

[R488], [R489], [R490], [R491]

Spherical Harmonics

class sympy.functions.special.spherical_harmonics.**Ynm**(*n*, *m*, *theta*, *phi*)

Spherical harmonics defined as

$$Y_n^m(\theta, \varphi) := \sqrt{\frac{(2n+1)(n-m)!}{4\pi(n+m)!}} \exp(im\varphi) \mathrm{P}_n^m\left(\cos(\theta)\right)$$

Explanation

`Ynm()` gives the spherical harmonic function of order n and m in θ and φ, $Y_n^m(\theta, \varphi)$. The four parameters are as follows: $n \geq 0$ an integer and m an integer such that $-n \leq m \leq n$ holds. The two angles are real-valued with $\theta \in [0, \pi]$ and $\varphi \in [0, 2\pi]$.

Examples

```
>>> from sympy import Ynm, Symbol, simplify
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> Ynm(n, m, theta, phi)
Ynm(n, m, theta, phi)
```

Several symmetries are known, for the order:

```
>>> Ynm(n, -m, theta, phi)
(-1)**m*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

As well as for the angles:

```
>>> Ynm(n, m, -theta, phi)
Ynm(n, m, theta, phi)
```

```
>>> Ynm(n, m, theta, -phi)
exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

For specific integers n and m we can evaluate the harmonics to more useful expressions:

```
>>> simplify(Ynm(0, 0, theta, phi).expand(func=True))
1/(2*sqrt(pi))
```

```
>>> simplify(Ynm(1, -1, theta, phi).expand(func=True))
sqrt(6)*exp(-I*phi)*sin(theta)/(4*sqrt(pi))
```

```
>>> simplify(Ynm(1, 0, theta, phi).expand(func=True))
sqrt(3)*cos(theta)/(2*sqrt(pi))
```

```
>>> simplify(Ynm(1, 1, theta, phi).expand(func=True))
-sqrt(6)*exp(I*phi)*sin(theta)/(4*sqrt(pi))
```

```
>>> simplify(Ynm(2, -2, theta, phi).expand(func=True))
sqrt(30)*exp(-2*I*phi)*sin(theta)**2/(8*sqrt(pi))
```

```
>>> simplify(Ynm(2, -1, theta, phi).expand(func=True))
sqrt(30)*exp(-I*phi)*sin(2*theta)/(8*sqrt(pi))
```

```
>>> simplify(Ynm(2, 0, theta, phi).expand(func=True))
sqrt(5)*(3*cos(theta)**2 - 1)/(4*sqrt(pi))
```

```
>>> simplify(Ynm(2, 1, theta, phi).expand(func=True))
-sqrt(30)*exp(I*phi)*sin(2*theta)/(8*sqrt(pi))
```

```
>>> simplify(Ynm(2, 2, theta, phi).expand(func=True))
sqrt(30)*exp(2*I*phi)*sin(theta)**2/(8*sqrt(pi))
```

We can differentiate the functions with respect to both angles:

```
>>> from sympy import Ynm, Symbol, diff
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> diff(Ynm(n, m, theta, phi), theta)
m*cot(theta)*Ynm(n, m, theta, phi) + sqrt((-m + n)*(m + n + 1))*exp(-
↪I*phi)*Ynm(n, m + 1, theta, phi)
```

```
>>> diff(Ynm(n, m, theta, phi), phi)
I*m*Ynm(n, m, theta, phi)
```

Further we can compute the complex conjugation:

```
>>> from sympy import Ynm, Symbol, conjugate
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> conjugate(Ynm(n, m, theta, phi))
(-1)**(2*m)*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

To get back the well known expressions in spherical coordinates, we use full expansion:

```
>>> from sympy import Ynm, Symbol, expand_func
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> expand_func(Ynm(n, m, theta, phi))
sqrt((2*n + 1)*factorial(-m + n)/factorial(m + n))*exp(I*m*phi)*assoc_
↪legendre(n, m, cos(theta))/(2*sqrt(pi))
```

See also:

Ynm_c (page 547), *Znm* (page 547)

References

[R492], [R493], [R494], [R495]

sympy.functions.special.spherical_harmonics.**Ynm_c**(*n*, *m*, *theta*, *phi*)

Conjugate spherical harmonics defined as

$$\overline{Y_n^m(\theta, \varphi)} := (-1)^m Y_n^{-m}(\theta, \varphi).$$

Examples

```
>>> from sympy import Ynm_c, Symbol, simplify
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
>>> Ynm_c(n, m, theta, phi)
(-1)**(2*m)*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
>>> Ynm_c(n, m, -theta, phi)
(-1)**(2*m)*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

For specific integers n and m we can evaluate the harmonics to more useful expressions:

```
>>> simplify(Ynm_c(0, 0, theta, phi).expand(func=True))
1/(2*sqrt(pi))
>>> simplify(Ynm_c(1, -1, theta, phi).expand(func=True))
sqrt(6)*exp(I*(-phi + 2*conjugate(phi)))*sin(theta)/(4*sqrt(pi))
```

See also:

Ynm (page 545), *Znm* (page 547)

References

[R496], [R497], [R498]

class sympy.functions.special.spherical_harmonics.**Znm**(*n*, *m*, *theta*, *phi*)

Real spherical harmonics defined as

$$Z_n^m(\theta, \varphi) := \begin{cases} \frac{Y_n^m(\theta,\varphi) + \overline{Y_n^m(\theta,\varphi)}}{\sqrt{2}} & m > 0 \\ Y_n^m(\theta, \varphi) & m = 0 \\ \frac{Y_n^m(\theta,\varphi) - \overline{Y_n^m(\theta,\varphi)}}{i\sqrt{2}} & m < 0 \end{cases}$$

which gives in simplified form

$$Z_n^m(\theta,\varphi) = \begin{cases} \frac{Y_n^m(\theta,\varphi)+(-1)^m Y_n^{-m}(\theta,\varphi)}{\sqrt{2}} & m > 0 \\ Y_n^m(\theta,\varphi) & m = 0 \\ \frac{Y_n^m(\theta,\varphi)-(-1)^m Y_n^{-m}(\theta,\varphi)}{i\sqrt{2}} & m < 0 \end{cases}$$

Examples

```
>>> from sympy import Znm, Symbol, simplify
>>> from sympy.abc import n, m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
>>> Znm(n, m, theta, phi)
Znm(n, m, theta, phi)
```

For specific integers n and m we can evaluate the harmonics to more useful expressions:

```
>>> simplify(Znm(0, 0, theta, phi).expand(func=True))
1/(2*sqrt(pi))
>>> simplify(Znm(1, 1, theta, phi).expand(func=True))
-sqrt(3)*sin(theta)*cos(phi)/(2*sqrt(pi))
>>> simplify(Znm(2, 1, theta, phi).expand(func=True))
-sqrt(15)*sin(2*theta)*cos(phi)/(4*sqrt(pi))
```

See also:

Ynm (page 545), *Ynm_c* (page 547)

References

[R499], [R500], [R501]

Tensor Functions

sympy.functions.special.tensor_functions.**Eijk**(**args*, ***kwargs*)

Represent the Levi-Civita symbol.

This is a compatibility wrapper to `LeviCivita()`.

See also:

LeviCivita (page 548)

sympy.functions.special.tensor_functions.**eval_levicivita**(**args*)

Evaluate Levi-Civita symbol.

class sympy.functions.special.tensor_functions.**LeviCivita**(**args*)

Represent the Levi-Civita symbol.

Explanation

For even permutations of indices it returns 1, for odd permutations -1, and for everything else (a repeated index) it returns 0.

Thus it represents an alternating pseudotensor.

Examples

```
>>> from sympy import LeviCivita
>>> from sympy.abc import i, j, k
>>> LeviCivita(1, 2, 3)
1
>>> LeviCivita(1, 3, 2)
-1
>>> LeviCivita(1, 2, 2)
0
>>> LeviCivita(i, j, k)
LeviCivita(i, j, k)
>>> LeviCivita(i, j, i)
0
```

See also:

Eijk (page 548)

class sympy.functions.special.tensor_functions.**KroneckerDelta**(*i*, *j*, *delta_range=None*)

The discrete, or Kronecker, delta function.

Parameters

i : Number, Symbol

The first index of the delta function.

j : Number, Symbol

The second index of the delta function.

Explanation

A function that takes in two integers i and j. It returns 0 if i and j are not equal, or it returns 1 if i and j are equal.

Examples

An example with integer indices:

```
>>> from sympy import KroneckerDelta
>>> KroneckerDelta(1, 2)
0
>>> KroneckerDelta(3, 3)
1
```

Symbolic indices:

```
>>> from sympy.abc import i, j, k
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

See also:

eval (page 550), *DiracDelta* (page 450)

References

[R502]

classmethod eval(*i*, *j*, *delta_range=None*)

Evaluates the discrete delta function.

Examples

```
>>> from sympy import KroneckerDelta
>>> from sympy.abc import i, j, k
```

```
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

indirect doctest

property indices_contain_equal_information

Returns True if indices are either both above or below fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, q).indices_contain_equal_information
True
>>> KroneckerDelta(p, q+1).indices_contain_equal_information
True
>>> KroneckerDelta(i, p).indices_contain_equal_information
False
```

property is_above_fermi

True if Delta can be non-zero above fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_above_fermi
True
>>> KroneckerDelta(p, i).is_above_fermi
False
>>> KroneckerDelta(p, q).is_above_fermi
True
```

See also:

is_below_fermi (page 551), *is_only_below_fermi* (page 552), *is_only_above_fermi* (page 552)

property is_below_fermi

True if Delta can be non-zero below fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_below_fermi
False
>>> KroneckerDelta(p, i).is_below_fermi
True
```

(continues on next page)

(continued from previous page)

```
>>> KroneckerDelta(p, q).is_below_fermi
True
```

See also:

is_above_fermi (page 551), *is_only_above_fermi* (page 552), *is_only_below_fermi* (page 552)

property is_only_above_fermi

True if Delta is restricted to above fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_only_above_fermi
True
>>> KroneckerDelta(p, q).is_only_above_fermi
False
>>> KroneckerDelta(p, i).is_only_above_fermi
False
```

See also:

is_above_fermi (page 551), *is_below_fermi* (page 551), *is_only_below_fermi* (page 552)

property is_only_below_fermi

True if Delta is restricted to below fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, i).is_only_below_fermi
True
>>> KroneckerDelta(p, q).is_only_below_fermi
False
>>> KroneckerDelta(p, a).is_only_below_fermi
False
```

See also:

is_above_fermi (page 551), *is_below_fermi* (page 551), *is_only_above_fermi* (page 552)

property killable_index

Returns the index which is preferred to substitute in the final expression.

Explanation

The index to substitute is the index with less information regarding fermi level. If indices contain the same information, 'a' is preferred before 'b'.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).killable_index
p
>>> KroneckerDelta(p, a).killable_index
p
>>> KroneckerDelta(i, j).killable_index
j
```

See also:

preferred_index (page 553)

property preferred_index

Returns the index which is preferred to keep in the final expression.

Explanation

The preferred index is the index with more information regarding fermi level. If indices contain the same information, 'a' is preferred before 'b'.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).preferred_index
i
>>> KroneckerDelta(p, a).preferred_index
a
>>> KroneckerDelta(i, j).preferred_index
i
```

See also:

killable_index (page 552)

Integrals

This module documentation contains details about Meijer G-functions and SymPy integrals. functions.

Contents

Computing Integrals using Meijer G-Functions

This text aims do describe in some detail the steps (and subtleties) involved in using Meijer G-functions for computing definite and indefinite integrals. We shall ignore proofs completely.

Overview

The algorithm to compute $\int f(x)\mathrm{d}x$ or $\int_0^\infty f(x)\mathrm{d}x$ generally consists of three steps:

1. Rewrite the integrand using Meijer G-functions (one or sometimes two).
2. Apply an integration theorem, to get the answer (usually expressed as another G-function).
3. Expand the result in named special functions.

Step (3) is implemented in the function hyperexpand (q.v.). Steps (1) and (2) are described below. Moreover, G-functions are usually branched. Thus our treatment of branched functions is described first.

Some other integrals (e.g. $\int_{-\infty}^\infty$) can also be computed by first recasting them into one of the above forms. There is a lot of choice involved here, and the algorithm is heuristic at best.

Polar Numbers and Branched Functions

Both Meijer G-Functions and Hypergeometric functions are typically branched (possible branchpoints being 0, ± 1, ∞). This is not very important when e.g. expanding a single hypergeometric function into named special functions, since sorting out the branches can be left to the human user. However this algorithm manipulates and transforms G-functions, and to do this correctly it needs at least some crude understanding of the branchings involved.

To begin, we consider the set $\mathcal{S} = \{(r, \theta) : r > 0, \theta \in \mathbb{R}\}$. We have a map $p : \mathcal{S} :\to \mathbb{C} - \{0\}, (r, \theta) \mapsto re^{i\theta}$. Decreeing this to be a local biholomorphism gives $\mathcal{S}$ both a topology and a complex structure. This Riemann Surface is usually referred to as the Riemann Surface of the logarithm, for the following reason: We can define maps $\mathrm{Exp} : \mathbb{C} \to \mathcal{S}, (x + iy) \mapsto (\exp(x), y)$ and $\mathrm{Log} : \mathcal{S} \to \mathbb{C}, (e^x, y) \mapsto x + iy$. These can both be shown to be holomorphic, and are indeed mutual inverses.

We also sometimes formally attach a point "zero" (0) to $\mathcal{S}$ and denote the resulting object $\mathcal{S}_0$. Notably there is no complex structure defined near 0. A fundamental system of neighbourhoods is given by $\{\mathrm{Exp}(z) : \Re(z) < k\}$, which at least defines a topology. Elements of $\mathcal{S}_0$ shall be called polar numbers. We further define functions $\mathrm{Arg} : \mathcal{S} \to \mathbb{R}, (r, \theta) \mapsto \theta$ and $|.| : \mathcal{S}_0 \to \mathbb{R}_{>0}, (r, \theta) \mapsto r$. These have evident meaning and are both continuous everywhere.

Using these maps many operations can be extended from $\mathbb{C}$ to $\mathcal{S}$. We define $\mathrm{Exp}(a)\,\mathrm{Exp}(b) = \mathrm{Exp}(a + b)$ for $a, b \in \mathbb{C}$, also for $a \in \mathcal{S}$ and $b \in \mathbb{C}$ we define $a^b = \mathrm{Exp}(b\,\mathrm{Log}(a))$. It can be checked easily that using these definitions, many algebraic properties holding for positive reals (e.g.

$(ab)^c = a^c b^c$) which hold in $\mathbb{C}$ only for some numbers (because of branch cuts) hold indeed for all polar numbers.

As one peculiarity it should be mentioned that addition of polar numbers is not usually defined. However, formal sums of polar numbers can be used to express branching behaviour. For example, consider the functions $F(z) = \sqrt{1+z}$ and $G(a,b) = \sqrt{a+b}$, where a, b, z are polar numbers. The general rule is that functions of a single polar variable are defined in such a way that they are continuous on circles, and agree with the usual definition for positive reals. Thus if $S(z)$ denotes the standard branch of the square root function on $\mathbb{C}$, we are forced to define

$$F(z) = \begin{cases} S(p(z)) & : |z| < 1 \\ S(p(z)) & : -\pi < \operatorname{Arg}(z) + 4\pi n \leq \pi \text{ for some } n \in \mathbb{Z} \\ -S(p(z)) & : \text{else} \end{cases}.$$

(We are omitting $|z| = 1$ here, this does not matter for integration.) Finally we define $G(a,b) = \sqrt{a}F(b/a)$.

Representing Branched Functions on the Argand Plane

Suppose $f : \mathcal{S} \to \mathbb{C}$ is a holomorphic function. We wish to define a function F on (part of) the complex numbers $\mathbb{C}$ that represents f as closely as possible. This process is knows as "introducing branch cuts". In our situation, there is actually a canonical way of doing this (which is adhered to in all of SymPy), as follows: Introduce the "cut complex plane" $C = \mathbb{C}\backslash\mathbb{R}_{\leq 0}$. Define a function $l : C \to \mathcal{S}$ via $re^{i\theta} \mapsto r\operatorname{Exp}(i\theta)$. Here $r > 0$ and $-\pi < \theta \leq \pi$. Then l is holomorphic, and we define $G = f \circ l$. This called "lifting to the principal branch" throughout the SymPy documentation.

Table Lookups and Inverse Mellin Transforms

Suppose we are given an integrand $f(x)$ and are trying to rewrite it as a single G-function. To do this, we first split $f(x)$ into the form $x^s g(x)$ (where $g(x)$ is supposed to be simpler than $f(x)$). This is because multiplicative powers can be absorbed into the G-function later. This splitting is done by `_split_mul(f, x)`. Then we assemble a tuple of functions that occur in f (e.g. if $f(x) = e^x \cos x$, we would assemble the tuple $(\cos, \exp)$). This is done by the function `_mytype(f, x)`. Next we index a lookup table (created using `_create_lookup_table()`) with this tuple. This (hopefully) yields a list of Meijer G-function formulae involving these functions, we then pattern-match all of them. If one fits, we were successful, otherwise not and we have to try something else.

Suppose now we want to rewrite as a product of two G-functions. To do this, we (try to) find all inequivalent ways of splitting $f(x)$ into a product $f_1(x)f_2(x)$. We could try these splittings in any order, but it is often a good idea to minimize (a) the number of powers occurring in $f_i(x)$ and (b) the number of different functions occurring in $f_i(x)$. Thus given e.g. $f(x) = \sin x\, e^x \sin 2x$ we should try $f_1(x) = \sin x \sin 2x$, $f_2(x) = e^x$ first. All of this is done by the function `_mul_as_two_parts(f)`.

Finally, we can try a recursive Mellin transform technique. Since the Meijer G-function is defined essentially as a certain inverse mellin transform, if we want to write a function $f(x)$ as a G-function, we can compute its mellin transform $F(s)$. If $F(s)$ is in the right form, the G-function expression can be read off. This technique generalises many standard rewritings, e.g. $e^{ax}e^{bx} = e^{(a+b)x}$.

One twist is that some functions don't have mellin transforms, even though they can be written as G-functions. This is true for example for $f(x) = e^x \sin x$ (the function grows too rapidly to have a mellin transform). However if the function is recognised to be analytic, then we can try to compute the mellin-transform of $f(ax)$ for a parameter a, and deduce the G-function expression by analytic continuation. (Checking for analyticity is easy. Since we can only deal with a certain subset of functions anyway, we only have to filter out those which are not analyitc.)

The function `_rewrite_single` does the table lookup and recursive mellin transform. The functions `_rewrite1` and `_rewrite2` respectively use above-mentioned helpers and `_rewrite_single` to rewrite their argument as respectively one or two G-functions.

Applying the Integral Theorems

If the integrand has been recast into G-functions, evaluating the integral is relatively easy. We first do some substitutions to reduce e.g. the exponent of the argument of the G-function to unity (see `_rewrite_saxena_1` and `_rewrite_saxena`, respectively, for one or two G-functions). Next we go through a list of conditions under which the integral theorem applies. It can fail for basically two reasons: either the integral does not exist, or the manipulations in deriving the theorem may not be allowed (for more details, see this [BlogPost]).

Sometimes this can be remedied by reducing the argument of the G-functions involved. For example it is clear that the G-function representing e^z is satisfies $G(\mathrm{Exp}(2\pi i)z) = G(z)$ for all $z \in \mathcal{S}$. The function `meijerg.get_period()` can be used to discover this, and the function `principal_branch(z, period)` in `functions/elementary/complexes.py` can be used to exploit the information. This is done transparently by the integration code.

The G-Function Integration Theorems

This section intends to display in detail the definite integration theorems used in the code. The following two formulae go back to Meijer (In fact he proved more general formulae; indeed in the literature formulae are usually staded in more general form. However it is very easy to deduce the general formulae from the ones we give here. It seemed best to keep the theorems as simple as possible, since they are very complicated anyway.):

1.
$$\int_0^\infty G_{p,q}^{m,n}\left(\left.\begin{matrix}a_1,\cdots,a_p\\ b_1,\cdots,b_q\end{matrix}\right|\eta x\right)\mathrm{d}x = \frac{\prod_{j=1}^m\Gamma(b_j+1)\prod_{j=1}^n\Gamma(-a_j)}{\eta\prod_{j=m+1}^q\Gamma(-b_j)\prod_{j=n+1}^p\Gamma(a_j+1)}$$

2.
$$\int_0^\infty G_{u,v}^{s,t}\left(\left.\begin{matrix}c_1,\cdots,c_u\\ d_1,\cdots,d_v\end{matrix}\right|\sigma x\right)G_{p,q}^{m,n}\left(\left.\begin{matrix}a_1,\cdots,a_p\\ b_1,\cdots,b_q\end{matrix}\right|\omega x\right)\mathrm{d}x = G_{v+p,u+q}^{m+t,n+s}\left(\left.\begin{matrix}a_1,\cdots,a_n,-d_1,\cdots,-d_v,a_{n+1},\cdots,a_p\\ b_1,\cdots,b_m,-c_1,\cdots,-c_u,b_{m+1},\cdots,b_q\end{matrix}\right|\frac{\omega}{\sigma}\right)$$

The more interesting question is under what conditions these formulae are valid. Below we detail the conditions implemented in SymPy. They are an amalgamation of conditions found in [Prudnikov1990] and [Luke1969]; please let us know if you find any errors.