The error function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(erf2(x, y))
erf2(conjugate(x), conjugate(y))
```

Differentiation with respect to x, y is supported:

```
>>> from sympy import diff
>>> diff(erf2(x, y), x)
-2*exp(-x**2)/sqrt(pi)
>>> diff(erf2(x, y), y)
2*exp(-y**2)/sqrt(pi)
```

See also:

erf **(page 471)**
Gaussian error function.

erfc **(page 473)**
Complementary error function.

erfi **(page 474)**
Imaginary error function.

erfinv **(page 477)**
Inverse error function.

erfcinv **(page 478)**
Inverse Complementary error function.

erf2inv **(page 479)**
Inverse two-argument error function.

References

[R352]

class sympy.functions.special.error_functions.**erfinv**(*z*)

Inverse Error Function. The erfinv function is defined as:

$$\operatorname{erf}(x) = y \quad \Rightarrow \quad \operatorname{erfinv}(y) = x$$

Examples

```
>>> from sympy import erfinv
>>> from sympy.abc import x
```

Several special values are known:

```
>>> erfinv(0)
0
>>> erfinv(1)
oo
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(erfinv(x), x)
sqrt(pi)*exp(erfinv(x)**2)/2
```

We can numerically evaluate the inverse error function to arbitrary precision on [-1, 1]:

```
>>> erfinv(0.2).evalf(30)
0.179143454621291692285822705344
```

See also:

erf **(page 471)**
Gaussian error function.

erfc **(page 473)**
Complementary error function.

erfi **(page 474)**
Imaginary error function.

erf2 **(page 476)**
Two-argument error function.

erfcinv **(page 478)**
Inverse Complementary error function.

erf2inv **(page 479)**
Inverse two-argument error function.

References

[R353], [R354]

inverse(*argindex=1*)
Returns the inverse of this function.

class sympy.functions.special.error_functions.**erfcinv**(*z*)
Inverse Complementary Error Function. The erfcinv function is defined as:

$$\operatorname{erfc}(x) = y \quad \Rightarrow \quad \operatorname{erfcinv}(y) = x$$

Examples

```
>>> from sympy import erfcinv
>>> from sympy.abc import x
```

Several special values are known:

```
>>> erfcinv(1)
0
>>> erfcinv(0)
oo
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(erfcinv(x), x)
-sqrt(pi)*exp(erfcinv(x)**2)/2
```

See also:

erf **(page 471)**
Gaussian error function.

erfc **(page 473)**
Complementary error function.

erfi **(page 474)**
Imaginary error function.

erf2 **(page 476)**
Two-argument error function.

erfinv **(page 477)**
Inverse error function.

erf2inv **(page 479)**
Inverse two-argument error function.

References

[R355], [R356]

inverse(*argindex=1*)
Returns the inverse of this function.

class sympy.functions.special.error_functions.**erf2inv**(*x*, *y*)
Two-argument Inverse error function. The erf2inv function is defined as:

$$\operatorname{erf2}(x, w) = y \quad \Rightarrow \quad \operatorname{erf2inv}(x, y) = w$$

Examples

```
>>> from sympy import erf2inv, oo
>>> from sympy.abc import x, y
```

Several special values are known:

```
>>> erf2inv(0, 0)
0
>>> erf2inv(1, 0)
1
>>> erf2inv(0, 1)
oo
>>> erf2inv(0, y)
erfinv(y)
>>> erf2inv(oo, y)
erfcinv(-y)
```

Differentiation with respect to x and y is supported:

```
>>> from sympy import diff
>>> diff(erf2inv(x, y), x)
exp(-x**2 + erf2inv(x, y)**2)
>>> diff(erf2inv(x, y), y)
sqrt(pi)*exp(erf2inv(x, y)**2)/2
```

See also:

erf **(page 471)**
Gaussian error function.

erfc **(page 473)**
Complementary error function.

erfi **(page 474)**
Imaginary error function.

erf2 **(page 476)**
Two-argument error function.

erfinv **(page 477)**
Inverse error function.

erfcinv **(page 478)**
Inverse complementary error function.

References

[R357]

class sympy.functions.special.error_functions.**FresnelIntegral**(z)
Base class for the Fresnel integrals.

class sympy.functions.special.error_functions.**fresnels**(z)
Fresnel integral S.

Explanation

This function is defined by

$$\operatorname{S}(z) = \int_0^z \sin{\frac{\pi}{2} t^2} \mathrm{d}t.$$

It is an entire function.

Examples

```
>>> from sympy import I, oo, fresnels
>>> from sympy.abc import z
```

Several special values are known:

```
>>> fresnels(0)
0
>>> fresnels(oo)
1/2
>>> fresnels(-oo)
-1/2
>>> fresnels(I*oo)
-I/2
>>> fresnels(-I*oo)
I/2
```

In general one can pull out factors of -1 and i from the argument:

```
>>> fresnels(-z)
-fresnels(z)
>>> fresnels(I*z)
-I*fresnels(z)
```

The Fresnel S integral obeys the mirror symmetry $\overline{S(z)} = S(\bar{z})$:

```
>>> from sympy import conjugate
>>> conjugate(fresnels(z))
fresnels(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(fresnels(z), z)
sin(pi*z**2/2)
```

Defining the Fresnel functions via an integral:

```
>>> from sympy import integrate, pi, sin, expand_func
>>> integrate(sin(pi*z**2/2), z)
3*fresnels(z)*gamma(3/4)/(4*gamma(7/4))
>>> expand_func(integrate(sin(pi*z**2/2), z))
fresnels(z)
```

We can numerically evaluate the Fresnel integral to arbitrary precision on the whole complex plane:

```
>>> fresnels(2).evalf(30)
0.343415678363698242195300815958
```

```
>>> fresnels(-2*I).evalf(30)
0.343415678363698242195300815958*I
```

See also:

fresnelc **(page 482)**
Fresnel cosine integral.

References

[R358], [R359], [R360], [R361], [R362]

class sympy.functions.special.error_functions.**fresnelc**(*z*)
Fresnel integral C.

Explanation

This function is defined by

$$\mathrm{C}(z) = \int_0^z \cos \frac{\pi}{2} t^2 \mathrm{d}t.$$

It is an entire function.

Examples

```
>>> from sympy import I, oo, fresnelc
>>> from sympy.abc import z
```

Several special values are known:

```
>>> fresnelc(0)
0
>>> fresnelc(oo)
1/2
>>> fresnelc(-oo)
-1/2
>>> fresnelc(I*oo)
I/2
>>> fresnelc(-I*oo)
-I/2
```

In general one can pull out factors of -1 and i from the argument:

```
>>> fresnelc(-z)
-fresnelc(z)
>>> fresnelc(I*z)
I*fresnelc(z)
```

The Fresnel C integral obeys the mirror symmetry $\overline{C(z)} = C(\bar{z})$:

```
>>> from sympy import conjugate
>>> conjugate(fresnelc(z))
fresnelc(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(fresnelc(z), z)
cos(pi*z**2/2)
```

Defining the Fresnel functions via an integral:

```
>>> from sympy import integrate, pi, cos, expand_func
>>> integrate(cos(pi*z**2/2), z)
fresnelc(z)*gamma(1/4)/(4*gamma(5/4))
>>> expand_func(integrate(cos(pi*z**2/2), z))
fresnelc(z)
```

We can numerically evaluate the Fresnel integral to arbitrary precision on the whole complex plane:

```
>>> fresnelc(2).evalf(30)
0.488253406075340754500223503357
```

```
>>> fresnelc(-2*I).evalf(30)
-0.488253406075340754500223503357*I
```

See also:

fresnels **(page 480)**
Fresnel sine integral.

References

[R363], [R364], [R365], [R366], [R367]

Exponential, Logarithmic and Trigonometric Integrals

class sympy.functions.special.error_functions.**Ei**(z)
The classical exponential integral.

Explanation

For use in SymPy, this function is defined as

$$\operatorname{Ei}(x) = \sum_{n=1}^{\infty} \frac{x^n}{n\, n!} + \log(x) + \gamma,$$

where γ is the Euler-Mascheroni constant.

If x is a polar number, this defines an analytic function on the Riemann surface of the logarithm. Otherwise this defines an analytic function in the cut plane $\mathbb{C} \setminus (-\infty, 0]$.

Background

The name exponential integral comes from the following statement:

$$\operatorname{Ei}(x) = \int_{-\infty}^{x} \frac{e^t}{t} \mathrm{d}t$$

If the integral is interpreted as a Cauchy principal value, this statement holds for $x > 0$ and $\mathrm{Ei}(x)$ as defined above.

Examples

```
>>> from sympy import Ei, polar_lift, exp_polar, I, pi
>>> from sympy.abc import x
```

```
>>> Ei(-1)
Ei(-1)
```

This yields a real value:

```
>>> Ei(-1).n(chop=True)
-0.219383934395520
```

On the other hand the analytic continuation is not real:

```
>>> Ei(polar_lift(-1)).n(chop=True)
-0.21938393439552 + 3.14159265358979*I
```

The exponential integral has a logarithmic branch point at the origin:

```
>>> Ei(x*exp_polar(2*I*pi))
Ei(x) + 2*I*pi
```

Differentiation is supported:

```
>>> Ei(x).diff(x)
exp(x)/x
```

The exponential integral is related to many other special functions. For example:

```
>>> from sympy import expint, Shi
>>> Ei(x).rewrite(expint)
-expint(1, x*exp_polar(I*pi)) - I*pi
>>> Ei(x).rewrite(Shi)
Chi(x) + Shi(x)
```

See also:

expint **(page 485)**
    Generalised exponential integral.

E1 **(page 487)**
    Special case of the generalised exponential integral.

li **(page 487)**
    Logarithmic integral.

Li **(page 489)**
    Offset logarithmic integral.

Si **(page 491)**
    Sine integral.

Ci **(page 492)**
Cosine integral.

Shi **(page 494)**
Hyperbolic sine integral.

Chi **(page 495)**
Hyperbolic cosine integral.

uppergamma **(page 466)**
Upper incomplete gamma function.

References

[R368], [R369], [R370]

class sympy.functions.special.error_functions.**expint**(*nu*, *z*)

Generalized exponential integral.

Explanation

This function is defined as

$$\operatorname{E}_\nu(z) = z^{\nu - 1} \Gamma(1 - \nu, z),$$

where $\Gamma(1 - \nu, z)$ is the upper incomplete gamma function (uppergamma).

Hence for z with positive real part we have

$$\operatorname{E}_\nu(z) = \int_1^\infty \frac{e^{-zt}}{t^\nu} \mathrm{d}t,$$

which explains the name.

The representation as an incomplete gamma function provides an analytic continuation for $\operatorname{E}_\nu(z)$. If ν is a non-positive integer, the exponential integral is thus an unbranched function of z, otherwise there is a branch point at the origin. Refer to the incomplete gamma function documentation for details of the branching behavior.

Examples

```
>>> from sympy import expint, S
>>> from sympy.abc import nu, z
```

Differentiation is supported. Differentiation with respect to z further explains the name: for integral orders, the exponential integral is an iterated integral of the exponential function.

```
>>> expint(nu, z).diff(z)
-expint(nu - 1, z)
```

Differentiation with respect to ν has no classical expression:

```
>>> expint(nu, z).diff(nu)
-z**(nu - 1)*meijerg(((), (1, 1)), ((0, 0, 1 - nu), ()), z)
```

At non-postive integer orders, the exponential integral reduces to the exponential function:

```
>>> expint(0, z)
exp(-z)/z
>>> expint(-1, z)
exp(-z)/z + exp(-z)/z**2
```

At half-integers it reduces to error functions:

```
>>> expint(S(1)/2, z)
sqrt(pi)*erfc(sqrt(z))/sqrt(z)
```

At positive integer orders it can be rewritten in terms of exponentials and `expint(1, z)`. Use `expand_func()` to do this:

```
>>> from sympy import expand_func
>>> expand_func(expint(5, z))
z**4*expint(1, z)/24 + (-z**3 + z**2 - 2*z + 6)*exp(-z)/24
```

The generalised exponential integral is essentially equivalent to the incomplete gamma function:

```
>>> from sympy import uppergamma
>>> expint(nu, z).rewrite(uppergamma)
z**(nu - 1)*uppergamma(1 - nu, z)
```

As such it is branched at the origin:

```
>>> from sympy import exp_polar, pi, I
>>> expint(4, z*exp_polar(2*pi*I))
I*pi*z**3/3 + expint(4, z)
>>> expint(nu, z*exp_polar(2*pi*I))
z**(nu - 1)*(exp(2*I*pi*nu) - 1)*gamma(1 - nu) + expint(nu, z)
```

See also:

***Ei* (page 483)**
Another related function called exponential integral.

***E1* (page 487)**
The classical case, returns expint(1, z).

***li* (page 487)**
Logarithmic integral.

***Li* (page 489)**
Offset logarithmic integral.

***Si* (page 491)**
Sine integral.

***Ci* (page 492)**
Cosine integral.

Shi **(page 494)**
Hyperbolic sine integral.

Chi **(page 495)**
Hyperbolic cosine integral.

uppergamma (page 466)

References

[R371], [R372], [R373]

`sympy.functions.special.error_functions.E1(z)`

Classical case of the generalized exponential integral.

Explanation

This is equivalent to `expint(1, z)`.

Examples

```
>>> from sympy import E1
>>> E1(0)
expint(1, 0)
```

```
>>> E1(5)
expint(1, 5)
```

See also:

Ei **(page 483)**
Exponential integral.

expint **(page 485)**
Generalised exponential integral.

li **(page 487)**
Logarithmic integral.

Li **(page 489)**
Offset logarithmic integral.

Si **(page 491)**
Sine integral.

Ci **(page 492)**
Cosine integral.

Shi **(page 494)**
Hyperbolic sine integral.

Chi **(page 495)**
Hyperbolic cosine integral.

class sympy.functions.special.error_functions.**li**(z)

The classical logarithmic integral.

Explanation

For use in SymPy, this function is defined as

$$\operatorname{li}(x) = \int_0^x \frac{1}{\log(t)} \mathrm{d}t\,.$$

Examples

```
>>> from sympy import I, oo, li
>>> from sympy.abc import z
```

Several special values are known:

```
>>> li(0)
0
>>> li(1)
-oo
>>> li(oo)
oo
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(li(z), z)
1/log(z)
```

Defining the li function via an integral: >>> from sympy import integrate >>> integrate(li(z)) z*li(z) - Ei(2*log(z))

```
>>> integrate(li(z),z)
z*li(z) - Ei(2*log(z))
```

The logarithmic integral can also be defined in terms of Ei:

```
>>> from sympy import Ei
>>> li(z).rewrite(Ei)
Ei(log(z))
>>> diff(li(z).rewrite(Ei), z)
1/log(z)
```

We can numerically evaluate the logarithmic integral to arbitrary precision on the whole complex plane (except the singular points):

```
>>> li(2).evalf(30)
1.04516378011749278484458888919
```

```
>>> li(2*I).evalf(30)
1.0652795784357498247001125598 + 3.08346052231061726610939702133*I
```

We can even compute Soldner's constant by the help of mpmath:

```
>>> from mpmath import findroot
>>> findroot(li, 2)
1.45136923488338
```

Further transformations include rewriting `li` in terms of the trigonometric integrals `Si`, `Ci`, `Shi` and `Chi`:

```
>>> from sympy import Si, Ci, Shi, Chi
>>> li(z).rewrite(Si)
-log(I*log(z)) - log(1/log(z))/2 + log(log(z))/2 + Ci(I*log(z)) +␣
↪Shi(log(z))
>>> li(z).rewrite(Ci)
-log(I*log(z)) - log(1/log(z))/2 + log(log(z))/2 + Ci(I*log(z)) +␣
↪Shi(log(z))
>>> li(z).rewrite(Shi)
-log(1/log(z))/2 + log(log(z))/2 + Chi(log(z)) - Shi(log(z))
>>> li(z).rewrite(Chi)
-log(1/log(z))/2 + log(log(z))/2 + Chi(log(z)) - Shi(log(z))
```

See also:

Li **(page 489)**
Offset logarithmic integral.

Ei **(page 483)**
Exponential integral.

expint **(page 485)**
Generalised exponential integral.

E1 **(page 487)**
Special case of the generalised exponential integral.

Si **(page 491)**
Sine integral.

Ci **(page 492)**
Cosine integral.

Shi **(page 494)**
Hyperbolic sine integral.

Chi **(page 495)**
Hyperbolic cosine integral.

References

[R374], [R375], [R376], [R377]

class sympy.functions.special.error_functions.**Li**(*z*)
The offset logarithmic integral.

Explanation

For use in SymPy, this function is defined as

$$\mathrm{Li}(x) = \mathrm{li}(x) - \mathrm{li}(2)$$

Examples

```
>>> from sympy import Li
>>> from sympy.abc import z
```

The following special value is known:

```
>>> Li(2)
0
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(Li(z), z)
1/log(z)
```

The shifted logarithmic integral can be written in terms of $li(z)$:

```
>>> from sympy import li
>>> Li(z).rewrite(li)
li(z) - li(2)
```


We can numerically evaluate the logarithmic integral to arbitrary precision on the whole complex plane (except the singular points):

```
>>> Li(2).evalf(30)
0
```

```
>>> Li(4).evalf(30)
1.92242131492155809316615998938
```

See also:

li **(page 487)**
Logarithmic integral.

Ei **(page 483)**
Exponential integral.

expint **(page 485)**
Generalised exponential integral.

E1 **(page 487)**
Special case of the generalised exponential integral.

Si **(page 491)**
Sine integral.

Ci **(page 492)**
Cosine integral.

Shi **(page 494)**
Hyperbolic sine integral.

Chi **(page 495)**
Hyperbolic cosine integral.

References

[R378], [R379], [R380]

class sympy.functions.special.error_functions.**Si**(*z*)

Sine integral.

Explanation

This function is defined by

$$\operatorname{Si}(z) = \int_0^z \frac{\sin{t}}{t} \mathrm{d}t.$$

It is an entire function.

Examples

```
>>> from sympy import Si
>>> from sympy.abc import z
```


The sine integral is an antiderivative of $sin(z)/z$:

```
>>> Si(z).diff(z)
sin(z)/z
```

It is unbranched:

```
>>> from sympy import exp_polar, I, pi
>>> Si(z*exp_polar(2*I*pi))
Si(z)
```

Sine integral behaves much like ordinary sine under multiplication by `I`:

```
>>> Si(I*z)
I*Shi(z)
>>> Si(-z)
-Si(z)
```

It can also be expressed in terms of exponential integrals, but beware that the latter is branched:

```
>>> from sympy import expint
>>> Si(z).rewrite(expint)
-I*(-expint(1, z*exp_polar(-I*pi/2))/2 +
     expint(1, z*exp_polar(I*pi/2))/2) + pi/2
```

It can be rewritten in the form of sinc function (by definition):

```
>>> from sympy import sinc
>>> Si(z).rewrite(sinc)
Integral(sinc(t), (t, 0, z))
```

See also:

Ci **(page 492)**
Cosine integral.

Shi **(page 494)**
Hyperbolic sine integral.

Chi **(page 495)**
Hyperbolic cosine integral.

Ei **(page 483)**
Exponential integral.

expint **(page 485)**
Generalised exponential integral.

sinc **(page 394)**
unnormalized sinc function

E1 **(page 487)**
Special case of the generalised exponential integral.

li **(page 487)**
Logarithmic integral.

Li **(page 489)**
Offset logarithmic integral.

References

[R381]

class sympy.functions.special.error_functions.**Ci**(*z*)

Cosine integral.

Explanation

This function is defined for positive x by

$$\operatorname{Ci}(x) = \gamma + \log x + \int_0^x \frac{\cos t - 1}{t} \mathrm{d}t = -\int_x^\infty \frac{\cos t}{t} \mathrm{d}t,$$

where γ is the Euler-Mascheroni constant.

We have

$$\operatorname{Ci}(z) = -\frac{\operatorname{E}_1\left(e^{i\pi/2} z\right) + \operatorname{E}_1\left(e^{-i\pi/2} z\right)}{2}$$

which holds for all polar z and thus provides an analytic continuation to the Riemann surface of the logarithm.

The formula also holds as stated for $z \in \mathbb{C}$ with $\Re(z) > 0$. By lifting to the principal branch, we obtain an analytic function on the cut complex plane.

Examples

```
>>> from sympy import Ci
>>> from sympy.abc import z
```

The cosine integral is a primitive of $\cos(z)/z$:

```
>>> Ci(z).diff(z)
cos(z)/z
```

It has a logarithmic branch point at the origin:

```
>>> from sympy import exp_polar, I, pi
>>> Ci(z*exp_polar(2*I*pi))
Ci(z) + 2*I*pi
```

The cosine integral behaves somewhat like ordinary cos under multiplication by i:

```
>>> from sympy import polar_lift
>>> Ci(polar_lift(I)*z)
Chi(z) + I*pi/2
>>> Ci(polar_lift(-1)*z)
Ci(z) + I*pi
```

It can also be expressed in terms of exponential integrals:

```
>>> from sympy import expint
>>> Ci(z).rewrite(expint)
-expint(1, z*exp_polar(-I*pi/2))/2 - expint(1, z*exp_polar(I*pi/2))/2
```

See also:

Si **(page 491)**
Sine integral.

Shi **(page 494)**
Hyperbolic sine integral.

Chi **(page 495)**
Hyperbolic cosine integral.

Ei **(page 483)**
Exponential integral.

expint **(page 485)**
Generalised exponential integral.

E1 **(page 487)**
Special case of the generalised exponential integral.

li **(page 487)**
Logarithmic integral.

Li **(page 489)**
Offset logarithmic integral.

References

[R382]

`class sympy.functions.special.error_functions.Shi(z)`

Sinh integral.

Explanation

This function is defined by

$$\operatorname{Shi}(z) = \int_0^z \frac{\sinh t}{t} \mathrm{d}t.$$

It is an entire function.

Examples

```
>>> from sympy import Shi
>>> from sympy.abc import z
```

The Sinh integral is a primitive of $\sinh(z)/z$:

```
>>> Shi(z).diff(z)
sinh(z)/z
```

It is unbranched:

```
>>> from sympy import exp_polar, I, pi
>>> Shi(z*exp_polar(2*I*pi))
Shi(z)
```

The sinh integral behaves much like ordinary sinh under multiplication by i:

```
>>> Shi(I*z)
I*Si(z)
>>> Shi(-z)
-Shi(z)
```

It can also be expressed in terms of exponential integrals, but beware that the latter is branched:

```
>>> from sympy import expint
>>> Shi(z).rewrite(expint)
expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```

See also:

Si **(page 491)**
Sine integral.

Ci **(page 492)**
Cosine integral.

Chi **(page 495)**
Hyperbolic cosine integral.

Ei **(page 483)**
Exponential integral.

expint **(page 485)**
Generalised exponential integral.

E1 **(page 487)**
Special case of the generalised exponential integral.

li **(page 487)**
Logarithmic integral.

Li **(page 489)**
Offset logarithmic integral.

References

[R383]

class sympy.functions.special.error_functions.**Chi**(*z*)

Cosh integral.

Explanation

This function is defined for positive x by

$$\operatorname{Chi}(x) = \gamma + \log x + \int_0^x \frac{\cosh t - 1}{t} \mathrm{d}t,$$

where γ is the Euler-Mascheroni constant.

We have

$$\operatorname{Chi}(z) = \operatorname{Ci}\left(e^{i\pi/2} z\right) - i\frac{\pi}{2},$$

which holds for all polar z and thus provides an analytic continuation to the Riemann surface of the logarithm. By lifting to the principal branch we obtain an analytic function on the cut complex plane.

Examples

```
>>> from sympy import Chi
>>> from sympy.abc import z
```

The cosh integral is a primitive of $\cosh(z)/z$:

```
>>> Chi(z).diff(z)
cosh(z)/z
```

It has a logarithmic branch point at the origin:

```
>>> from sympy import exp_polar, I, pi
>>> Chi(z*exp_polar(2*I*pi))
Chi(z) + 2*I*pi
```

The cosh integral behaves somewhat like ordinary cosh under multiplication by i:

```
>>> from sympy import polar_lift
>>> Chi(polar_lift(I)*z)
Ci(z) + I*pi/2
>>> Chi(polar_lift(-1)*z)
Chi(z) + I*pi
```

It can also be expressed in terms of exponential integrals:

```
>>> from sympy import expint
>>> Chi(z).rewrite(expint)
-expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```

See also:

***Si* (page 491)**
Sine integral.

***Ci* (page 492)**
Cosine integral.

***Shi* (page 494)**
Hyperbolic sine integral.

***Ei* (page 483)**
Exponential integral.

***expint* (page 485)**
Generalised exponential integral.

***E1* (page 487)**
Special case of the generalised exponential integral.

***li* (page 487)**
Logarithmic integral.

***Li* (page 489)**
Offset logarithmic integral.

References

[R384]

Bessel Type Functions

class sympy.functions.special.bessel.**BesselBase**(*nu*, *z*)

Abstract base class for Bessel-type functions.

This class is meant to reduce code duplication. All Bessel-type functions can 1) be differentiated, with the derivatives expressed in terms of similar functions, and 2) be rewritten in terms of other Bessel-type functions.

Here, Bessel-type functions are assumed to have one complex parameter.

To use this base class, define class attributes _a and _b such that `2*F_n' = -_a*F_{n+1} + b*F_{n-1}`.

property argument

The argument of the Bessel-type function.

property order

The order of the Bessel-type function.

class sympy.functions.special.bessel.**besselj**(*nu*, *z*)

Bessel function of the first kind.

Explanation

The Bessel J function of order ν is defined to be the function satisfying Bessel's differential equation

$$z^2 \frac{\mathrm{d}^2 w}{\mathrm{d}z^2} + z \frac{\mathrm{d}w}{\mathrm{d}z} + (z^2 - \nu^2) w = 0,$$

with Laurent expansion

$$J_\nu(z) = z^\nu \left(\frac{1}{\Gamma(\nu + 1) 2^\nu} + O(z^2) \right),$$

if ν is not a negative integer. If $\nu = -n \in \mathbb{Z}_{<0}$ *is* a negative integer, then the definition is

$$J_{-n}(z) = (-1)^n J_n(z).$$

Examples

Create a Bessel function object:

```
>>> from sympy import besselj, jn
>>> from sympy.abc import z, n
>>> b = besselj(n, z)
```

Differentiate it:

```
>>> b.diff(z)
besselj(n - 1, z)/2 - besselj(n + 1, z)/2
```

Rewrite in terms of spherical Bessel functions:

```
>>> b.rewrite(jn)
sqrt(2)*sqrt(z)*jn(n - 1/2, z)/sqrt(pi)
```

Access the parameter and argument:

```
>>> b.order
n
>>> b.argument
z
```

See also:

bessely (page 498), *besseli* (page 499), *besselk* (page 499)

References

[R385], [R386], [R387], [R388]

class sympy.functions.special.bessel.**bessely**(*nu*, *z*)

Bessel function of the second kind.

Explanation

The Bessel Y function of order ν is defined as

$$Y_\nu(z) = \lim_{\mu\to\nu} \frac{J_\mu(z) \cos(\pi\mu) - J_{-\mu}(z)}{\sin(\pi\mu)},$$

where $J_\mu(z)$ is the Bessel function of the first kind.

It is a solution to Bessel's equation, and linearly independent from J_ν.

Examples

```
>>> from sympy import bessely, yn
>>> from sympy.abc import z, n
>>> b = bessely(n, z)
>>> b.diff(z)
bessely(n - 1, z)/2 - bessely(n + 1, z)/2
>>> b.rewrite(yn)
sqrt(2)*sqrt(z)*yn(n - 1/2, z)/sqrt(pi)
```

See also:

besselj (page 497), *besseli* (page 499), *besselk* (page 499)

References

[R389]

class sympy.functions.special.bessel.**besseli**(*nu*, *z*)

Modified Bessel function of the first kind.

Explanation

The Bessel I function is a solution to the modified Bessel equation

$$z^2 \frac{\mathrm{d}^2 w}{\mathrm{d}z^2} + z \frac{\mathrm{d}w}{\mathrm{d}z} + (z^2 + \nu^2)^2 w = 0.$$

It can be defined as

$$I_\nu(z) = i^{-\nu} J_\nu(iz),$$

where $J_\nu(z)$ is the Bessel function of the first kind.

Examples

```
>>> from sympy import besseli
>>> from sympy.abc import z, n
>>> besseli(n, z).diff(z)
besseli(n - 1, z)/2 + besseli(n + 1, z)/2
```


See also:

besselj (page 497), *bessely* (page 498), *besselk* (page 499)

References

[R390]

class sympy.functions.special.bessel.**besselk**(*nu*, *z*)

Modified Bessel function of the second kind.

Explanation

The Bessel K function of order ν is defined as

$$K_\nu(z) = \lim_{\mu \to \nu} \frac{\pi}{2} \frac{I_{-\mu}(z) - I_\mu(z)}{\sin(\pi\mu)},$$

where $I_\mu(z)$ is the modified Bessel function of the first kind.

It is a solution of the modified Bessel equation, and linearly independent from Y_ν.

Examples

```
>>> from sympy import besselk
>>> from sympy.abc import z, n
>>> besselk(n, z).diff(z)
-besselk(n - 1, z)/2 - besselk(n + 1, z)/2
```

See also:

besselj (page 497), *besseli* (page 499), *bessely* (page 498)

References

[R391]

class sympy.functions.special.bessel.**hankel1**(*nu*, *z*)

Hankel function of the first kind.

Explanation

This function is defined as

$$H_\nu^{(1)} = J_\nu(z) + iY_\nu(z),$$

where $J_\nu(z)$ is the Bessel function of the first kind, and $Y_\nu(z)$ is the Bessel function of the second kind.

It is a solution to Bessel's equation.

Examples

```
>>> from sympy import hankel1
>>> from sympy.abc import z, n
>>> hankel1(n, z).diff(z)
hankel1(n - 1, z)/2 - hankel1(n + 1, z)/2
```

See also:

hankel2 (page 500), *besselj* (page 497), *bessely* (page 498)

References

[R392]

class sympy.functions.special.bessel.**hankel2**(*nu*, *z*)

Hankel function of the second kind.

Explanation

This function is defined as

$$H_{\nu}^{(2)} = J_{\nu}(z) - iY_{\nu}(z),$$

where $J_{\nu}(z)$ is the Bessel function of the first kind, and $Y_{\nu}(z)$ is the Bessel function of the second kind.

It is a solution to Bessel's equation, and linearly independent from $H_{\nu}^{(1)}$.

Examples

```
>>> from sympy import hankel2
>>> from sympy.abc import z, n
>>> hankel2(n, z).diff(z)
hankel2(n - 1, z)/2 - hankel2(n + 1, z)/2
```

See also:

hankel1 (page 500), *besselj* (page 497), *bessely* (page 498)

References

[R393]

class sympy.functions.special.bessel.**jn**(*nu*, *z*)

Spherical Bessel function of the first kind.

Explanation

This function is a solution to the spherical Bessel equation

$$z^2 \frac{\mathrm{d}^2 w}{\mathrm{d}z^2} + 2z \frac{\mathrm{d}w}{\mathrm{d}z} + (z^2 - \nu(\nu + 1))w = 0.$$

It can be defined as

$$j_{\nu}(z) = \sqrt{\frac{\pi}{2z}} J_{\nu+\frac{1}{2}}(z),$$

where $J_{\nu}(z)$ is the Bessel function of the first kind.

The spherical Bessel functions of integral order are calculated using the formula:

$$j_n(z) = f_n(z) \sin z + (-1)^{n+1} f_{-n-1}(z) \cos z,$$

where the coefficients $f_n(z)$ are available as *sympy.polys.orthopolys.spherical_bessel_fn()* (page 2441).

Examples

```
>>> from sympy import Symbol, jn, sin, cos, expand_func, besselj, bessely
>>> z = Symbol("z")
>>> nu = Symbol("nu", integer=True)
>>> print(expand_func(jn(0, z)))
sin(z)/z
>>> expand_func(jn(1, z)) == sin(z)/z**2 - cos(z)/z
True
>>> expand_func(jn(3, z))
(-6/z**2 + 15/z**4)*sin(z) + (1/z - 15/z**3)*cos(z)
>>> jn(nu, z).rewrite(besselj)
sqrt(2)*sqrt(pi)*sqrt(1/z)*besselj(nu + 1/2, z)/2
>>> jn(nu, z).rewrite(bessely)
(-1)**nu*sqrt(2)*sqrt(pi)*sqrt(1/z)*bessely(-nu - 1/2, z)/2
>>> jn(2, 5.2+0.3j).evalf(20)
0.099419756723640344491 - 0.054525080242173562897*I
```

See also:

besselj (page 497), *bessely* (page 498), *besselk* (page 499), *yn* (page 502)

References

[R394]

class sympy.functions.special.bessel.**yn**(*nu*, *z*)

Spherical Bessel function of the second kind.

Explanation

This function is another solution to the spherical Bessel equation, and linearly independent from j_n. It can be defined as

$$y_\nu(z) = \sqrt{\frac{\pi}{2z}} Y_{\nu+\frac{1}{2}}(z),$$

where $Y_\nu(z)$ is the Bessel function of the second kind.

For integral orders n, y_n is calculated using the formula:

$$y_n(z) = (-1)^{n+1} j_{-n-1}(z)$$

Examples

```
>>> from sympy import Symbol, yn, sin, cos, expand_func, besselj, bessely
>>> z = Symbol("z")
>>> nu = Symbol("nu", integer=True)
>>> print(expand_func(yn(0, z)))
-cos(z)/z
>>> expand_func(yn(1, z)) == -cos(z)/z**2-sin(z)/z
```

(continues on next page)

(continued from previous page)

```
True
>>> yn(nu, z).rewrite(besselj)
(-1)**(nu + 1)*sqrt(2)*sqrt(pi)*sqrt(1/z)*besselj(-nu - 1/2, z)/2
>>> yn(nu, z).rewrite(bessely)
sqrt(2)*sqrt(pi)*sqrt(1/z)*bessely(nu + 1/2, z)/2
>>> yn(2, 5.2+0.3j).evalf(20)
0.18525034196069722536 + 0.014895573969924817587*I
```

See also:

besselj (page 497), *bessely* (page 498), *besselk* (page 499), *jn* (page 501)

References

[R395]

sympy.functions.special.bessel.**jn_zeros**(*n*, *k*, *method='sympy'*, *dps=15*)

Zeros of the spherical Bessel function of the first kind.

Parameters

n : integer

order of Bessel function

k : integer

number of zeros to return

Explanation

This returns an array of zeros of jn up to the k-th zero.

- method = "sympy": uses mpmath.besseljzero
- method = "scipy": uses the SciPy's sph_jn and newton to find all roots, which is faster than computing the zeros using a general numerical solver, but it requires SciPy and only works with low precision floating point numbers. (The function used with method="sympy" is a recent addition to mpmath; before that a general solver was used.)

Examples

```
>>> from sympy import jn_zeros
>>> jn_zeros(2, 4, dps=5)
[5.7635, 9.095, 12.323, 15.515]
```

See also:

jn (page 501), *yn* (page 502), *besselj* (page 497), *besselk* (page 499), *bessely* (page 498)

class sympy.functions.special.bessel.**marcumq**(*m*, *a*, *b*)

The Marcum Q-function.

Explanation

The Marcum Q-function is defined by the meromorphic continuation of

$$Q_m(a, b) = a^{-m+1} \int_b^{\infty} x^m e^{-\frac{a^2}{2} - \frac{x^2}{2}} I_{m-1}(ax)\, dx$$

Examples

```
>>> from sympy import marcumq
>>> from sympy.abc import m, a, b
>>> marcumq(m, a, b)
marcumq(m, a, b)
```

Special values:

```
>>> marcumq(m, 0, b)
uppergamma(m, b**2/2)/gamma(m)
>>> marcumq(0, 0, 0)
0
>>> marcumq(0, a, 0)
1 - exp(-a**2/2)
>>> marcumq(1, a, a)
1/2 + exp(-a**2)*besseli(0, a**2)/2
>>> marcumq(2, a, a)
1/2 + exp(-a**2)*besseli(0, a**2)/2 + exp(-a**2)*besseli(1, a**2)
```

Differentiation with respect to a and b is supported:

```
>>> from sympy import diff
>>> diff(marcumq(m, a, b), a)
a*(-marcumq(m, a, b) + marcumq(m + 1, a, b))
>>> diff(marcumq(m, a, b), b)
-a**(1 - m)*b**m*exp(-a**2/2 - b**2/2)*besseli(m - 1, a*b)
```

References

[R396], [R397]

Airy Functions

class sympy.functions.special.bessel.**AiryBase**(**args*)

Abstract base class for Airy functions.

This class is meant to reduce code duplication.

class sympy.functions.special.bessel.**airyai**(*arg*)

The Airy function Ai of the first kind.

Explanation

The Airy function $\operatorname{Ai}(z)$ is defined to be the function satisfying Airy's differential equation

$$\frac{\mathrm{d}^2 w(z)}{\mathrm{d}z^2} - zw(z) = 0.$$

Equivalently, for real z

$$\operatorname{Ai}(z) := \frac{1}{\pi} \int_0^\infty \cos\left(\frac{t^3}{3} + zt\right) \mathrm{d}t.$$

Examples

Create an Airy function object:

```
>>> from sympy import airyai
>>> from sympy.abc import z
```

```
>>> airyai(z)
airyai(z)
```

Several special values are known:

```
>>> airyai(0)
3**(1/3)/(3*gamma(2/3))
>>> from sympy import oo
>>> airyai(oo)
0
>>> airyai(-oo)
0
```


The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airyai(z))
airyai(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(airyai(z), z)
airyaiprime(z)
>>> diff(airyai(z), z, 2)
z*airyai(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airyai(z), z, 0, 3)
3**(5/6)*gamma(1/3)/(6*pi) - 3**(1/6)*z*gamma(2/3)/(2*pi) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airyai(-2).evalf(50)
0.22740742820168557599192443603787379946077222541710
```

Rewrite $\mathrm{Ai}(z)$ in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airyai(z).rewrite(hyper)
-3**(2/3)*z*hyper((), (4/3,), z**3/9)/(3*gamma(1/3)) + 3**(1/3)*hyper((),
↪ (2/3,), z**3/9)/(3*gamma(2/3))
```

See also:

***airybi* (page 506)**
Airy function of the second kind.

***airyaiprime* (page 508)**
Derivative of the Airy function of the first kind.

***airybiprime* (page 509)**
Derivative of the Airy function of the second kind.

References

[R398], [R399], [R400], [R401]

class sympy.functions.special.bessel.**airybi**(*arg*)

The Airy function Bi of the second kind.

Explanation

The Airy function $\mathrm{Bi}(z)$ is defined to be the function satisfying Airy's differential equation

$$\frac{\mathrm{d}^2 w(z)}{\mathrm{d}z^2} - z w(z) = 0.$$

Equivalently, for real z

$$\mathrm{Bi}(z) := \frac{1}{\pi} \int_0^\infty \exp\left(-\frac{t^3}{3} + zt\right) + \sin\left(\frac{t^3}{3} + zt\right) \mathrm{d}t.$$

Examples

Create an Airy function object:

```
>>> from sympy import airybi
>>> from sympy.abc import z
```

```
>>> airybi(z)
airybi(z)
```

Several special values are known:

```
>>> airybi(0)
3**(5/6)/(3*gamma(2/3))
>>> from sympy import oo
>>> airybi(oo)
oo
>>> airybi(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airybi(z))
airybi(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(airybi(z), z)
airybiprime(z)
>>> diff(airybi(z), z, 2)
z*airybi(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airybi(z), z, 0, 3)
3**(1/3)*gamma(1/3)/(2*pi) + 3**(2/3)*z*gamma(2/3)/(2*pi) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airybi(-2).evalf(50)
-0.41230258795639848808323405461146104203453483447240
```

Rewrite $\mathrm{Bi}(z)$ in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airybi(z).rewrite(hyper)
3**(1/6)*z*hyper((), (4/3,), z**3/9)/gamma(1/3) + 3**(5/6)*hyper((), (2/
↪3,), z**3/9)/(3*gamma(2/3))
```

See also:

airyai **(page 504)**
Airy function of the first kind.

airyaiprime **(page 508)**
Derivative of the Airy function of the first kind.

airybiprime **(page 509)**
Derivative of the Airy function of the second kind.

References

[R402], [R403], [R404], [R405]

class sympy.functions.special.bessel.**airyaiprime**(*arg*)

The derivative Ai' of the Airy function of the first kind.

Explanation

The Airy function $\operatorname{Ai}'(z)$ is defined to be the function

$$\operatorname{Ai}'(z) := \frac{\mathrm{d} \operatorname{Ai}(z)}{\mathrm{d} z}.$$

Examples

Create an Airy function object:

```
>>> from sympy import airyaiprime
>>> from sympy.abc import z
```

```
>>> airyaiprime(z)
airyaiprime(z)
```

Several special values are known:

```
>>> airyaiprime(0)
-3**(2/3)/(3*gamma(1/3))
>>> from sympy import oo
>>> airyaiprime(oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airyaiprime(z))
airyaiprime(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(airyaiprime(z), z)
z*airyai(z)
>>> diff(airyaiprime(z), z, 2)
z*airyaiprime(z) + airyai(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airyaiprime(z), z, 0, 3)
-3**(2/3)/(3*gamma(1/3)) + 3**(1/3)*z**2/(6*gamma(2/3)) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airyaiprime(-2).evalf(50)
0.61825902074169104140626429133247528291577794512415
```

Rewrite $\mathrm{Ai}'(z)$ in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airyaiprime(z).rewrite(hyper)
3**(1/3)*z**2*hyper((), (5/3,), z**3/9)/(6*gamma(2/3)) - 3**(2/
↪3)*hyper((), (1/3,), z**3/9)/(3*gamma(1/3))
```

See also:

airyai **(page 504)**
Airy function of the first kind.

airybi **(page 506)**
Airy function of the second kind.

airybiprime **(page 509)**
Derivative of the Airy function of the second kind.

References

[R406], [R407], [R408], [R409]

class sympy.functions.special.bessel.**airybiprime**(*arg*)

The derivative Bi' of the Airy function of the first kind.

Explanation

The Airy function $\mathrm{Bi}'(z)$ is defined to be the function

$$\mathrm{Bi}'(z) := \frac{\mathrm{d}\,\mathrm{Bi}(z)}{\mathrm{d}z}.$$

Examples

Create an Airy function object:

```
>>> from sympy import airybiprime
>>> from sympy.abc import z
```

```
>>> airybiprime(z)
airybiprime(z)
```

Several special values are known:

```
>>> airybiprime(0)
3**(1/6)/gamma(1/3)
>>> from sympy import oo
>>> airybiprime(oo)
oo
>>> airybiprime(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airybiprime(z))
airybiprime(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(airybiprime(z), z)
z*airybi(z)
>>> diff(airybiprime(z), z, 2)
z*airybiprime(z) + airybi(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airybiprime(z), z, 0, 3)
3**(1/6)/gamma(1/3) + 3**(5/6)*z**2/(6*gamma(2/3)) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airybiprime(-2).evalf(50)
0.27879516692116952268509756941098324140300059345163
```

Rewrite $\mathrm{Bi}'(z)$ in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airybiprime(z).rewrite(hyper)
3**(5/6)*z**2*hyper((), (5/3,), z**3/9)/(6*gamma(2/3)) + 3**(1/
↪6)*hyper((), (1/3,), z**3/9)/gamma(1/3)
```

See also:

airyai **(page 504)**
  Airy function of the first kind.

airybi **(page 506)**
  Airy function of the second kind.

airyaiprime **(page 508)**
  Derivative of the Airy function of the first kind.

References

[R410], [R411], [R412], [R413]

B-Splines

sympy.functions.special.bsplines.**bspline_basis**(*d*, *knots*, *n*, *x*)

The n-th B-spline at x of degree d with knots.

Parameters

d : integer

degree of bspline

knots : list of integer values

list of knots points of bspline

n : integer

n-th B-spline

x : symbol

Explanation

B-Splines are piecewise polynomials of degree d. They are defined on a set of knots, which is a sequence of integers or floats.

Examples

The 0th degree splines have a value of 1 on a single interval:

```
>>> from sympy import bspline_basis
>>> from sympy.abc import x
>>> d = 0
>>> knots = tuple(range(5))
>>> bspline_basis(d, knots, 0, x)
Piecewise((1, (x >= 0) & (x <= 1)), (0, True))
```

For a given (d, knots) there are len(knots)-d-1 B-splines defined, that are indexed by n (starting at 0).

Here is an example of a cubic B-spline:

```
>>> bspline_basis(3, tuple(range(5)), 0, x)
Piecewise((x**3/6, (x >= 0) & (x <= 1)),
          (-x**3/2 + 2*x**2 - 2*x + 2/3,
          (x >= 1) & (x <= 2)),
          (x**3/2 - 4*x**2 + 10*x - 22/3,
          (x >= 2) & (x <= 3)),
          (-x**3/6 + 2*x**2 - 8*x + 32/3,
          (x >= 3) & (x <= 4)),
          (0, True))
```

By repeating knot points, you can introduce discontinuities in the B-splines and their derivatives:

```
>>> d = 1
>>> knots = (0, 0, 2, 3, 4)
>>> bspline_basis(d, knots, 0, x)
Piecewise((1 - x/2, (x >= 0) & (x <= 2)), (0, True))
```

It is quite time consuming to construct and evaluate B-splines. If you need to evaluate a B-spline many times, it is best to lambdify them first:

```
>>> from sympy import lambdify
>>> d = 3
>>> knots = tuple(range(10))
>>> b0 = bspline_basis(d, knots, 0, x)
>>> f = lambdify(x, b0)
>>> y = f(0.5)
```

See also:

bspline_basis_set (page 512)

References

[R414]

sympy.functions.special.bsplines.**bspline_basis_set**(*d*, *knots*, *x*)

Return the `len(knots)-d-1` B-splines at *x* of degree *d* with *knots*.

Parameters

d : integer

degree of bspline

knots : list of integers

list of knots points of bspline

x : symbol

Explanation

This function returns a list of piecewise polynomials that are the `len(knots)-d-1` B-splines of degree *d* for the given knots. This function calls `bspline_basis(d, knots, n, x)` for different values of *n*.

Examples

```
>>> from sympy import bspline_basis_set
>>> from sympy.abc import x
>>> d = 2
>>> knots = range(5)
>>> splines = bspline_basis_set(d, knots, x)
>>> splines
[Piecewise((x**2/2, (x >= 0) & (x <= 1)),
           (-x**2 + 3*x - 3/2, (x >= 1) & (x <= 2)),
           (x**2/2 - 3*x + 9/2, (x >= 2) & (x <= 3)),
           (0, True)),
Piecewise((x**2/2 - x + 1/2, (x >= 1) & (x <= 2)),
          (-x**2 + 5*x - 11/2, (x >= 2) & (x <= 3)),
          (x**2/2 - 4*x + 8, (x >= 3) & (x <= 4)),
          (0, True))]
```

See also:

bspline_basis (page 511)

sympy.functions.special.bsplines.**interpolating_spline**(*d, x, X, Y*)

Return spline of degree d, passing through the given X and Y values.

Parameters

d : integer

Degree of Bspline strictly greater than equal to one

x : symbol

X : list of strictly increasing integer values

list of X coordinates through which the spline passes

Y : list of strictly increasing integer values

list of Y coordinates through which the spline passes

Explanation

This function returns a piecewise function such that each part is a polynomial of degree not greater than d. The value of d must be 1 or greater and the values of X must be strictly increasing.

Examples

```
>>> from sympy import interpolating_spline
>>> from sympy.abc import x
>>> interpolating_spline(1, x, [1, 2, 4, 7], [3, 6, 5, 7])
Piecewise((3*x, (x >= 1) & (x <= 2)),
        (7 - x/2, (x >= 2) & (x <= 4)),
        (2*x/3 + 7/3, (x >= 4) & (x <= 7)))
>>> interpolating_spline(3, x, [-2, 0, 1, 3, 4], [4, 2, 1, 1, 3])
```

(continues on next page)

(continued from previous page)

```
Piecewise((7*x**3/117 + 7*x**2/117 - 131*x/117 + 2, (x >= -2) & (x <=␣
↪1)),
        (10*x**3/117 - 2*x**2/117 - 122*x/117 + 77/39, (x >= 1) & (x <=␣
↪4)))
```

See also:

bspline_basis_set (page 512), *interpolating_poly* (page 2438)

Riemann Zeta and Related Functions

class sympy.functions.special.zeta_functions.**zeta**(*z*, *a_=None*)

Hurwitz zeta function (or Riemann zeta function).

Explanation

For $\operatorname{Re}(a) > 0$ and $\operatorname{Re}(s) > 1$, this function is defined as

$$\zeta(s,a) = \sum_{n=0}^{\infty} \frac{1}{(n+a)^s},$$

where the standard choice of argument for $n + a$ is used. For fixed a with $\operatorname{Re}(a) > 0$ the Hurwitz zeta function admits a meromorphic continuation to all of $\mathbb{C}$, it is an unbranched function with a simple pole at $s = 1$.

Analytic continuation to other a is possible under some circumstances, but this is not typically done.

The Hurwitz zeta function is a special case of the Lerch transcendent:

$$\zeta(s,a) = \Phi(1,s,a).$$

This formula defines an analytic continuation for all possible values of s and a (also $\operatorname{Re}(a) < 0$), see the documentation of *lerchphi* (page 517) for a description of the branching behavior.

If no value is passed for a, by this function assumes a default value of $a = 1$, yielding the Riemann zeta function.

Examples

For $a = 1$ the Hurwitz zeta function reduces to the famous Riemann zeta function:

$$\zeta(s,1) = \zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}.$$

```
>>> from sympy import zeta
>>> from sympy.abc import s
>>> zeta(s, 1)
zeta(s)
>>> zeta(s)
zeta(s)
```

The Riemann zeta function can also be expressed using the Dirichlet eta function:

```
>>> from sympy import dirichlet_eta
>>> zeta(s).rewrite(dirichlet_eta)
dirichlet_eta(s)/(1 - 2**(1 - s))
```

The Riemann zeta function at positive even integer and negative odd integer values is related to the Bernoulli numbers:

```
>>> zeta(2)
pi**2/6
>>> zeta(4)
pi**4/90
>>> zeta(-1)
-1/12
```

The specific formulae are:

$$\zeta(2n) = (-1)^{n+1} \frac{B_{2n} (2\pi)^{2n}}{2(2n)!}$$

$$\zeta(-n) = -\frac{B_{n+1}}{n+1}$$

At negative even integers the Riemann zeta function is zero:

```
>>> zeta(-4)
0
```


No closed-form expressions are known at positive odd integers, but numerical evaluation is possible:

```
>>> zeta(3).n()
1.20205690315959
```

The derivative of $\zeta(s, a)$ with respect to a can be computed:

```
>>> from sympy.abc import a
>>> zeta(s, a).diff(a)
-s*zeta(s + 1, a)
```

However the derivative with respect to s has no useful closed form expression:

```
>>> zeta(s, a).diff(s)
Derivative(zeta(s, a), s)
```

The Hurwitz zeta function can be expressed in terms of the Lerch transcendent, *lerchphi* (page 517):

```
>>> from sympy import lerchphi
>>> zeta(s, a).rewrite(lerchphi)
lerchphi(1, s, a)
```

See also:

dirichlet_eta (page 516), *lerchphi* (page 517), *polylog* (page 516)

References

[R415], [R416]

class sympy.functions.special.zeta_functions.**dirichlet_eta**(*s*)

Dirichlet eta function.

Explanation

For $\operatorname{Re}(s) > 0$, this function is defined as

$$\eta(s) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{n^s}.$$

It admits a unique analytic continuation to all of $\mathbb{C}$. It is an entire, unbranched function.

Examples

The Dirichlet eta function is closely related to the Riemann zeta function:

```
>>> from sympy import dirichlet_eta, zeta
>>> from sympy.abc import s
>>> dirichlet_eta(s).rewrite(zeta)
(1 - 2**(1 - s))*zeta(s)
```

See also:

zeta (page 514)

References

[R417]

class sympy.functions.special.zeta_functions.**polylog**(*s*, *z*)

Polylogarithm function.

Explanation

For $|z| < 1$ and $s \in \mathbb{C}$, the polylogarithm is defined by

$$\operatorname{Li}_s(z) = \sum_{n=1}^{\infty} \frac{z^n}{n^s},$$

where the standard branch of the argument is used for n. It admits an analytic continuation which is branched at $z = 1$ (notably not on the sheet of initial definition), $z = 0$ and $z = \infty$.

The name polylogarithm comes from the fact that for $s = 1$, the polylogarithm is related to the ordinary logarithm (see examples), and that

$$\operatorname{Li}_{s+1}(z) = \int_0^z \frac{\operatorname{Li}_s(t)}{t} \mathrm{d}t.$$