Sage Reference Manual: Functions

Release 6.9

The Sage Development Team

October 13, 2015

CONTENTS

CHAPTER

ONE

LOGARITHMIC FUNCTIONS

AUTHORS:

- Yoora Yi Tenen (2012-11-16): Add documentation for `log()` (trac ticket #12113)

class sage.functions.log.**Function_dilog**

Bases: sage.symbolic.function.GinacFunction

The dilogarithm function $\mathrm{Li}_2(z) = \sum_{k=1}^{\infty} z^k/k^2$.

This is simply an alias for polylog(2, z).

EXAMPLES:

```
sage: dilog(1)
1/6*pi^2
sage: dilog(1/2)
1/12*pi^2 - 1/2*log(2)^2
sage: dilog(x^2+1)
dilog(x^2 + 1)
sage: dilog(-1)
-1/12*pi^2
sage: dilog(-1.1)
-0.890838090262283
sage: float(dilog(1))
1.6449340668482262
sage: var('z')
z
sage: dilog(z).diff(z, 2)
log(-z + 1)/z^2 - 1/((z - 1)*z)
sage: dilog(z).series(z==1/2, 3)
(1/12*pi^2 - 1/2*log(2)^2) + (-2*log(1/2))*(z - 1/2) + (2*log(1/2) + 2)*(z - 1/2)^2 + Order(1/8*

sage: latex(dilog(z))
{\rm Li}_2\left(z\right)
```

TESTS:

`conjugate(dilog(x))==dilog(conjugate(x))` unless on the branch cuts which run along the positive real axis beginning at 1.:

```
sage: conjugate(dilog(x))
conjugate(dilog(x))
sage: var('y',domain='positive')
y
sage: conjugate(dilog(y))
conjugate(dilog(y))
sage: conjugate(dilog(1/19))
dilog(1/19)
```

```
sage: conjugate(dilog(1/2*I))
dilog(-1/2*I)
sage: dilog(conjugate(1/2*I))
dilog(-1/2*I)
sage: conjugate(dilog(2))
conjugate(dilog(2))
```

class sage.functions.log.**Function_exp**

Bases: sage.symbolic.function.GinacFunction

The exponential function, $\exp(x) = e^x$.

EXAMPLES:

```
sage: exp(-1)
e^(-1)
sage: exp(2)
e^2
sage: exp(2).n(100)
7.3890560989306502272304274606
sage: exp(x^2 + log(x))
e^(x^2 + log(x))
sage: exp(x^2 + log(x)).simplify()
x*e^(x^2)
sage: exp(2.5)
12.1824939607035
sage: exp(float(2.5))
12.182493960703473
sage: exp(RDF('2.5'))
12.182493960703473
```

To prevent automatic evaluation, use the `hold` parameter:

```
sage: exp(I*pi,hold=True)
e^(I*pi)
sage: exp(0,hold=True)
e^0
```

To then evaluate again, we currently must use Maxima via sage.symbolic.expression.Expression.simplify():

```
sage: exp(0,hold=True).simplify()
1

sage: exp(pi*I/2)
I
sage: exp(pi*I)
-1
sage: exp(8*pi*I)
1
sage: exp(7*pi*I/2)
-I
```

The precision for the result is deduced from the precision of the input. Convert the input to a higher precision explicitly if a result with higher precision is desired:

```
sage: t = exp(RealField(100)(2)); t
7.3890560989306502272304274606
sage: t.prec()
100
sage: exp(2).n(100)
7.3890560989306502272304274606
```

TEST:

```
sage: latex(exp(x))
e^{x}
sage: latex(exp(sqrt(x)))
e^{\sqrt{x}}
sage: latex(exp)
\exp
sage: latex(exp(sqrt(x))^x)
\left(e^{\sqrt{x}}\right)^{x}
sage: latex(exp(sqrt(x)^x))
e^{\left(\sqrt{x}^{x}\right)}
```

Test conjugates:

```
sage: conjugate(exp(x))
e^conjugate(x)
```

Test simplifications when taking powers of exp, #7264:

```
sage: var('a,b,c,II')
(a, b, c, II)
sage: model_exp = exp(II)**a*(b)
sage: sol1_l={b: 5.0, a: 1.1}
sage: model_exp.subs(sol1_l)
5.00000000000000*(e^II)^1.10000000000000

sage: exp(3)^II*exp(x)
(e^3)^II*e^x
sage: exp(x)*exp(x)
e^(2*x)
sage: exp(x)*exp(a)
e^(a + x)
sage: exp(x)*exp(a)^2
e^(2*a + x)
```

Another instance of the same problem, #7394:

```
sage: 2*sqrt(e)
2*sqrt(e)
```

class sage.functions.log.**Function_lambert_w**

Bases: sage.symbolic.function.BuiltinFunction

The integral branches of the Lambert W function $W_n(z)$.

This function satisfies the equation

$$z = W_n(z)e^{W_n(z)}$$

INPUT:

- n - an integer. $n = 0$ corresponds to the principal branch.
- z - a complex number

If called with a single argument, that argument is z and the branch n is assumed to be 0 (the principal branch).

ALGORITHM:

Numerical evaluation is handled using the mpmath and SciPy libraries.

REFERENCES:

- Wikipedia article Lambert_W_function

EXAMPLES:

Evaluation of the principal branch:

```
sage: lambert_w(1.0)
0.567143290409784
sage: lambert_w(-1).n()
-0.318131505204764 + 1.33723570143069*I
sage: lambert_w(-1.5 + 5*I)
1.17418016254171 + 1.10651494102011*I
```

Evaluation of other branches:

```
sage: lambert_w(2, 1.0)
-2.40158510486800 + 10.7762995161151*I
```

Solutions to certain exponential equations are returned in terms of lambert_w:

```
sage: S = solve(e^(5*x)+x==0, x, to_poly_solve=True)
sage: z = S[0].rhs(); z
-1/5*lambert_w(5)
sage: N(z)
-0.265344933048440
```

Check the defining equation numerically at $z = 5$:

```
sage: N(lambert_w(5)*exp(lambert_w(5)) - 5)
0.000000000000000
```

There are several special values of the principal branch which are automatically simplified:

```
sage: lambert_w(0)
0
sage: lambert_w(e)
1
sage: lambert_w(-1/e)
-1
```

Integration (of the principal branch) is evaluated using Maxima:

```
sage: integrate(lambert_w(x), x)
(lambert_w(x)^2 - lambert_w(x) + 1)*x/lambert_w(x)
sage: integrate(lambert_w(x), x, 0, 1)
(lambert_w(1)^2 - lambert_w(1) + 1)/lambert_w(1) - 1
sage: integrate(lambert_w(x), x, 0, 1.0)
0.3303661247616807
```

Warning: The integral of a non-principal branch is not implemented, neither is numerical integration using GSL. The `numerical_integral()` function does work if you pass a lambda function:

```
sage: numerical_integral(lambda x: lambert_w(x), 0, 1)
(0.33036612476168054, 3.667800782666048e-15)
```

class sage.functions.log.**Function_log**

Bases: sage.symbolic.function.GinacFunction

The natural logarithm of x. See *log*? for more information about its behavior.

EXAMPLES:

```
sage: ln(e^2)
2
sage: ln(2)
log(2)
sage: ln(10)
log(10)

sage: ln(RDF(10))
2.302585092994046
sage: ln(2.718)
0.999896315728952
sage: ln(2.0)
0.693147180559945
sage: ln(float(-1))
3.141592653589793j
sage: ln(complex(-1))
3.141592653589793j
```

The `hold` parameter can be used to prevent automatic evaluation:

```
sage: log(-1,hold=True)
log(-1)
sage: log(-1)
I*pi
sage: I.log(hold=True)
log(I)
sage: I.log(hold=True).simplify()
1/2*I*pi
```

TESTS:

```
sage: latex(x.log())
\log\left(x\right)
sage: latex(log(1/4))
\log\left(\frac{1}{4}\right)
sage: loads(dumps(ln(x)+1))
log(x) + 1
```

`conjugate(log(x))==log(conjugate(x))` unless on the branch cut which runs along the negative real axis.:

```
sage: conjugate(log(x))
conjugate(log(x))
sage: var('y', domain='positive')
y
sage: conjugate(log(y))
log(y)
sage: conjugate(log(y+I))
conjugate(log(y + I))
sage: conjugate(log(-1))
-I*pi
sage: log(conjugate(-1))
I*pi
```

Check if float arguments are handled properly.:

```
sage: from sage.functions.log import function_log as log
sage: log(float(5))
1.6094379124341003
```

```
sage: log(float(0))
-inf
sage: log(float(-1))
3.141592653589793j
sage: log(x).subs(x=float(-1))
3.141592653589793j
```

class sage.functions.log.**Function_polylog**

Bases: sage.symbolic.function.GinacFunction

The polylog function $\mathrm{Li}_n(z) = \sum_{k=1}^{\infty} z^k/k^n$.

INPUT:

- n - object
- z - object

EXAMPLES:

```
sage: polylog(1, x)
-log(-x + 1)
sage: polylog(2,1)
1/6*pi^2
sage: polylog(2,x^2+1)
polylog(2, x^2 + 1)
sage: polylog(4,0.5)
polylog(4, 0.500000000000000)

sage: f = polylog(4, 1); f
1/90*pi^4
sage: f.n()
1.08232323371114

sage: polylog(4, 2).n()
2.42786280675470 - 0.174371300025453*I
sage: complex(polylog(4,2))
(2.4278628067547032-0.17437130002545306j)
sage: float(polylog(4,0.5))
0.5174790616738993

sage: z = var('z')
sage: polylog(2,z).series(z==0, 5)
1*z + 1/4*z^2 + 1/9*z^3 + 1/16*z^4 + Order(z^5)

sage: loads(dumps(polylog))
polylog

sage: latex(polylog(5, x))
{\rm Li}_{5}(x)
```

TESTS:

Check if trac ticket #8459 is fixed:

```
sage: t = maxima(polylog(5,x)).sage(); t
polylog(5, x)
sage: t.operator() == polylog
True
sage: t.subs(x=.5).n()
0.508400579242269
```

CHAPTER

TWO

TRIGONOMETRIC FUNCTIONS

class sage.functions.trig.**Function_arccos**

Bases: sage.symbolic.function.GinacFunction

The arccosine function.

EXAMPLES:

```
sage: arccos(0.5)
1.04719755119660
sage: arccos(1/2)
1/3*pi
sage: arccos(1 + 1.0*I)
0.904556894302381 - 1.06127506190504*I
sage: arccos(3/4).n(100)
0.72273424781341561117837735264
```

We can delay evaluation using the `hold` parameter:

```
sage: arccos(0,hold=True)
arccos(0)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = arccos(0,hold=True); a.simplify()
1/2*pi
```

`conjugate(arccos(x))==arccos(conjugate(x))`, unless on the branch cuts, which run along the real axis outside the interval [-1, +1].:

```
sage: conjugate(arccos(x))
conjugate(arccos(x))
sage: var('y', domain='positive')
y
sage: conjugate(arccos(y))
conjugate(arccos(y))
sage: conjugate(arccos(y+I))
conjugate(arccos(y + I))
sage: conjugate(arccos(1/16))
arccos(1/16)
sage: conjugate(arccos(2))
conjugate(arccos(2))
sage: conjugate(arccos(-2))
pi - conjugate(arccos(2))
```

TESTS:

```
sage: arccos(x).operator()
arccos
```

class sage.functions.trig.**Function_arccot**
Bases: sage.symbolic.function.BuiltinFunction

The arccotangent function.

EXAMPLES:

```
sage: arccot(1/2)
arccot(1/2)
sage: RDF(arccot(1/2))
1.1071487177940904
sage: arccot(1 + I)
arccot(I + 1)
```

We can delay evaluation using the `hold` parameter:

```
sage: arccot(1,hold=True)
arccot(1)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = arccot(1,hold=True); a.simplify()
1/4*pi
```

class sage.functions.trig.**Function_arccsc**
Bases: sage.symbolic.function.BuiltinFunction

The arccosecant function.

EXAMPLES:

```
sage: arccsc(2)
arccsc(2)
sage: RDF(arccsc(2))  # rel tol 1e-15
0.5235987755982988
sage: arccsc(1 + I)
arccsc(I + 1)
```

We can delay evaluation using the `hold` parameter:

```
sage: arccsc(1,hold=True)
arccsc(1)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = arccsc(1,hold=True); a.simplify()
1/2*pi
```

class sage.functions.trig.**Function_arcsec**
Bases: sage.symbolic.function.BuiltinFunction

The arcsecant function.

EXAMPLES:

```
sage: arcsec(2)
arcsec(2)
sage: arcsec(2.0)
1.04719755119660
```

```
sage: RDF(arcsec(2))  # abs tol 1e-15
1.0471975511965976
sage: arcsec(1 + I)
arcsec(I + 1)
```

We can delay evaluation using the `hold` parameter:

```
sage: arcsec(1,hold=True)
arcsec(1)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = arcsec(1,hold=True); a.simplify()
0
```

class `sage.functions.trig.`**`Function_arcsin`**

Bases: `sage.symbolic.function.GinacFunction`

The arcsine function.

EXAMPLES:

```
sage: arcsin(0.5)
0.523598775598299
sage: arcsin(1/2)
1/6*pi
sage: arcsin(1 + 1.0*I)
0.666239432492515 + 1.06127506190504*I
```

We can delay evaluation using the `hold` parameter:

```
sage: arcsin(0,hold=True)
arcsin(0)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = arcsin(0,hold=True); a.simplify()
0
```

`conjugate(arcsin(x))==arcsin(conjugate(x))`, unless on the branch cuts which run along the real axis outside the interval [-1, +1].:

```
sage: conjugate(arcsin(x))
conjugate(arcsin(x))
sage: var('y', domain='positive')
y
sage: conjugate(arcsin(y))
conjugate(arcsin(y))
sage: conjugate(arcsin(y+I))
conjugate(arcsin(y + I))
sage: conjugate(arcsin(1/16))
arcsin(1/16)
sage: conjugate(arcsin(2))
conjugate(arcsin(2))
sage: conjugate(arcsin(-2))
-conjugate(arcsin(2))
```

TESTS:

```
sage: arcsin(x).operator()
arcsin
```

class sage.functions.trig.**Function_arctan**
Bases: sage.symbolic.function.GinacFunction

The arctangent function.

EXAMPLES:

```
sage: arctan(1/2)
arctan(1/2)
sage: RDF(arctan(1/2))  # rel tol 1e-15
0.46364760900080615
sage: arctan(1 + I)
arctan(I + 1)
sage: arctan(1/2).n(100)
0.46364760900080611621425623146
```

We can delay evaluation using the `hold` parameter:

```
sage: arctan(0,hold=True)
arctan(0)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = arctan(0,hold=True); a.simplify()
0
```

`conjugate(arctan(x))==arctan(conjugate(x))`, unless on the branch cuts which run along the imaginary axis outside the interval [-I, +I].:

```
sage: conjugate(arctan(x))
conjugate(arctan(x))
sage: var('y', domain='positive')
y
sage: conjugate(arctan(y))
arctan(y)
sage: conjugate(arctan(y+I))
conjugate(arctan(y + I))
sage: conjugate(arctan(1/16))
arctan(1/16)
sage: conjugate(arctan(-2*I))
conjugate(arctan(-2*I))
sage: conjugate(arctan(2*I))
conjugate(arctan(2*I))
sage: conjugate(arctan(I/2))
arctan(-1/2*I)
```

TESTS:

```
sage: arctan(x).operator()
arctan
```

class sage.functions.trig.**Function_arctan2**
Bases: sage.symbolic.function.GinacFunction

The modified arctangent function.

Returns the arc tangent (measured in radians) of y/x, where unlike `arctan(y/x)`, the signs of both `x` and `y` are considered. In particular, this function measures the angle of a ray through the origin and (x, y), with the positive x-axis the zero mark, and with output angle θ being between $-\pi < \theta <= \pi$.

Hence, `arctan2(y,x) = arctan(y/x)` only for $x > 0$. One may consider the usual arctan to measure angles of lines through the origin, while the modified function measures rays through the origin.

Note that the y-coordinate is by convention the first input.

EXAMPLES:

Note the difference between the two functions:

```
sage: arctan2(1,-1)
3/4*pi
sage: arctan(1/-1)
-1/4*pi
```

This is consistent with Python and Maxima:

```
sage: maxima.atan2(1,-1)
3*%pi/4
sage: math.atan2(1,-1)
2.356194490192345
```

More examples:

```
sage: arctan2(1,0)
1/2*pi
sage: arctan2(2,3)
arctan(2/3)
sage: arctan2(-1,-1)
-3/4*pi
```

Of course we can approximate as well:

```
sage: arctan2(-1/2,1).n(100)
-0.46364760900080611621425623146
sage: arctan2(2,3).n(100)
0.58800260354756755124561108063
```

We can delay evaluation using the `hold` parameter:

```
sage: arctan2(-1/2,1,hold=True)
arctan2(-1/2, 1)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: arctan2(-1/2,1,hold=True).simplify()
-arctan(1/2)
```

The function also works with numpy arrays as input:

```
sage: import numpy
sage: a = numpy.linspace(1, 3, 3)
sage: b = numpy.linspace(3, 6, 3)
sage: atan2(a, b)
array([ 0.32175055,  0.41822433,  0.46364761])

sage: atan2(1,a)
array([ 0.78539816,  0.46364761,  0.32175055])

sage: atan2(a, 1)
array([ 0.78539816,  1.10714872,  1.24904577])
```

TESTS:

```
sage: x,y = var('x,y')
sage: arctan2(y,x).operator()
arctan2
```

Check if trac ticket #8565 is fixed:

```
sage: atan2(-pi,0)
-1/2*pi
```

Check if trac ticket #8564 is fixed:

```
sage: arctan2(x,x)._sympy_()
atan2(x, x)
```

Check if numerical evaluation works trac ticket #9913:

```
sage: arctan2(0, -log(2)).n()
3.14159265358979
```

Check if atan2(0,0) throws error of trac ticket #11423:

```
sage: atan2(0,0)
Traceback (most recent call last):
...
RuntimeError: arctan2_eval(): arctan2(0,0) encountered

sage: atan2(0,0,hold=True)
arctan2(0, 0)

sage: atan2(0,0,hold=True).n()
Traceback (most recent call last):
...
ValueError: arctan2(0,0) undefined
```

Check if trac ticket #10062 is fixed, this was caused by `(I*I).is_positive()` returning `True`:

```
sage: arctan2(0, I*I)
pi
```

class sage.functions.trig.**Function_cos**

Bases: sage.symbolic.function.GinacFunction

The cosine function.

EXAMPLES:

```
sage: cos(pi)
-1
sage: cos(x).subs(x==pi)
-1
sage: cos(2).n(100)
-0.41614683654714238699756822950
sage: loads(dumps(cos))
cos
```

We can prevent evaluation using the `hold` parameter:

```
sage: cos(0,hold=True)
cos(0)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = cos(0,hold=True); a.simplify()
1
```

TESTS:

```
sage: conjugate(cos(x))
cos(conjugate(x))
sage: cos(complex(1,1))      # rel tol 1e-15
(0.8337300251311491-0.9888977057628651j)
```

class sage.functions.trig.**Function_cot**

Bases: sage.symbolic.function.BuiltinFunction

The cotangent function.

EXAMPLES:

```
sage: cot(pi/4)
1
sage: RR(cot(pi/4))
1.00000000000000
sage: cot(1/2)
cot(1/2)
sage: cot(0.5)
1.83048772171245

sage: latex(cot(x))
\cot\left(x\right)
```

We can prevent evaluation using the `hold` parameter:

```
sage: cot(pi/4,hold=True)
cot(1/4*pi)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = cot(pi/4,hold=True); a.simplify()
1
```

class sage.functions.trig.**Function_csc**

Bases: sage.symbolic.function.BuiltinFunction

The cosecant function.

EXAMPLES:

```
sage: csc(pi/4)
sqrt(2)
sage: RR(csc(pi/4))
1.41421356237310
sage: n(csc(pi/4),100)
1.4142135623730950488016887242
sage: csc(1/2)
csc(1/2)
sage: csc(0.5)
2.08582964293349

sage: latex(csc(x))
\csc\left(x\right)
```

We can prevent evaluation using the `hold` parameter:

```
sage: csc(pi/4,hold=True)
csc(1/4*pi)
```

To then evaluate again, we currently must use Maxima via sage.symbolic.expression.Expression.simplify():

```
sage: a = csc(pi/4,hold=True); a.simplify()
sqrt(2)
```

class sage.functions.trig.**Function_sec**

Bases: sage.symbolic.function.BuiltinFunction

The secant function.

EXAMPLES:

```
sage: sec(pi/4)
sqrt(2)
sage: RR(sec(pi/4))
1.41421356237310
sage: n(sec(pi/4),100)
1.4142135623730950488016887242
sage: sec(1/2)
sec(1/2)
sage: sec(0.5)
1.13949392732455

sage: latex(sec(x))
\sec\left(x\right)
```

We can prevent evaluation using the hold parameter:

```
sage: sec(pi/4,hold=True)
sec(1/4*pi)
```

To then evaluate again, we currently must use Maxima via sage.symbolic.expression.Expression.simplify():

```
sage: a = sec(pi/4,hold=True); a.simplify()
sqrt(2)
```

class sage.functions.trig.**Function_sin**

Bases: sage.symbolic.function.GinacFunction

The sine function.

EXAMPLES:

```
sage: sin(0)
0
sage: sin(x).subs(x==0)
0
sage: sin(2).n(100)
0.90929742682568169539601986591
sage: loads(dumps(sin))
sin
```

We can prevent evaluation using the hold parameter:

```
sage: sin(0,hold=True)
sin(0)
```

To then evaluate again, we currently must use Maxima via sage.symbolic.expression.Expression.simplify():

```
sage: a = sin(0,hold=True); a.simplify()
0
```

TESTS:

```
sage: conjugate(sin(x))
sin(conjugate(x))
sage: sin(complex(1,1))      # rel tol 1e-15
(1.2984575814159773+0.6349639147847361j)

sage: sin(pi/5)
1/4*sqrt(-2*sqrt(5) + 10)
sage: sin(pi/8)
1/2*sqrt(-sqrt(2) + 2)
sage: sin(pi/24)
1/4*sqrt(-2*sqrt(6) - 2*sqrt(2) + 8)
sage: sin(pi/30)
-1/8*sqrt(5) + 1/4*sqrt(-3/2*sqrt(5) + 15/2) - 1/8
sage: cos(pi/8)
1/2*sqrt(sqrt(2) + 2)
sage: cos(pi/10)
1/2*sqrt(1/2*sqrt(5) + 5/2)
sage: cos(pi/12)
1/12*sqrt(6)*(sqrt(3) + 3)
sage: cos(pi/15)
1/8*sqrt(5) + 1/4*sqrt(3/2*sqrt(5) + 15/2) - 1/8
sage: cos(pi/24)
1/4*sqrt(2*sqrt(6) + 2*sqrt(2) + 8)
sage: tan(pi/5)
sqrt(-2*sqrt(5) + 5)
sage: tan(pi/8)
sqrt(2) - 1
sage: tan(pi/10)
sqrt(-2/5*sqrt(5) + 1)
sage: tan(pi/16)
-sqrt(2) + sqrt(2*sqrt(2) + 4) - 1
sage: tan(pi/20)
sqrt(5) - 1/2*sqrt(8*sqrt(5) + 20) + 1
sage: tan(pi/24)
sqrt(6) - sqrt(3) + sqrt(2) - 2

sage: all(sin(rat*pi).n(200)-sin(rat*pi,hold=True).n(200) < 1e-30 for rat in [1/5,2/5,1/30,7/30,
True
sage: all(cos(rat*pi).n(200)-cos(rat*pi,hold=True).n(200) < 1e-30 for rat in [1/10,3/10,1/12,5/1
True
sage: all(tan(rat*pi).n(200)-tan(rat*pi,hold=True).n(200) < 1e-30 for rat in [1/5,2/5,1/10,3/10,
True
```

class sage.functions.trig.**Function_tan**

Bases: sage.symbolic.function.GinacFunction

The tangent function.

EXAMPLES:

```
sage: tan(pi)
0
sage: tan(3.1415)
-0.0000926535900581913
sage: tan(3.1415/4)
0.999953674278156
sage: tan(pi/4)
1
sage: tan(1/2)
```

```
tan(1/2)
sage: RR(tan(1/2))
0.546302489843790
```

We can prevent evaluation using the `hold` parameter:

```
sage: tan(pi/4,hold=True)
tan(1/4*pi)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = tan(pi/4,hold=True); a.simplify()
1
```

TESTS:

```
sage: conjugate(tan(x))
tan(conjugate(x))
sage: tan(complex(1,1))      # rel tol 1e-15
(0.2717525853195118+1.0839233273386946j)
```

CHAPTER

THREE

HYPERBOLIC FUNCTIONS

class sage.functions.hyperbolic.**Function_arccosh**

Bases: sage.symbolic.function.GinacFunction

The inverse of the hyperbolic cosine function.

EXAMPLES:

```
sage: arccosh(1/2)
arccosh(1/2)
sage: arccosh(1 + I*1.0)
1.06127506190504 + 0.904556894302381*I
sage: float(arccosh(2))
1.3169578969248168
sage: cosh(float(arccosh(2)))
2.0
```

Warning: If the input is in the complex field or symbolic (which includes rational and integer input), the output will be complex. However, if the input is a real decimal, the output will be real or NaN. See the examples for details.

```
sage: arccosh(0.5)
NaN
sage: arccosh(1/2)
arccosh(1/2)
sage: arccosh(1/2).n()
NaN
sage: arccosh(CC(0.5))
1.04719755119660*I
sage: arccosh(0)
1/2*I*pi
sage: arccosh(-1)
I*pi
```

To prevent automatic evaluation use the `hold` argument:

```
sage: arccosh(-1,hold=True)
arccosh(-1)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: arccosh(-1,hold=True).simplify()
I*pi
```

`conjugate(arccosh(x))==arccosh(conjugate(x))` unless on the branch cut which runs along the real axis from +1 to -inf.:

```
sage: conjugate(arccosh(x))
conjugate(arccosh(x))
sage: var('y', domain='positive')
y
sage: conjugate(arccosh(y))
conjugate(arccosh(y))
sage: conjugate(arccosh(y+I))
conjugate(arccosh(y + I))
sage: conjugate(arccosh(1/16))
conjugate(arccosh(1/16))
sage: conjugate(arccosh(2))
arccosh(2)
sage: conjugate(arccosh(I/2))
arccosh(-1/2*I)
```

TESTS:

```
sage: arccosh(x).operator()
arccosh
sage: latex(arccosh(x))
{\rm arccosh}\left(x\right)
```

class sage.functions.hyperbolic.**Function_arccoth**

Bases: sage.functions.hyperbolic.HyperbolicFunction

The inverse of the hyperbolic cotangent function.

EXAMPLES:

```
sage: arccoth(2.0)
0.549306144334055
sage: arccoth(2)
arccoth(2)
sage: arccoth(1 + I*1.0)
0.402359478108525 - 0.553574358897045*I
sage: arccoth(2).n(200)
0.54930614433405484569762261846126285232374527891137472586735
```

Using first the $.n(53)$ method is slightly more precise than converting directly to a `float`:

```
sage: float(arccoth(2))  # abs tol 1e-16
0.5493061443340548
sage: float(arccoth(2).n(53))    # Correct result to 53 bits
0.5493061443340549
sage: float(arccoth(2).n(100))  # Compute 100 bits and then round to 53
0.5493061443340549
```

TESTS:

```
sage: latex(arccoth(x))
{\rm arccoth}\left(x\right)
```

class sage.functions.hyperbolic.**Function_arccsch**

Bases: sage.functions.hyperbolic.HyperbolicFunction

The inverse of the hyperbolic cosecant function.

EXAMPLES:

```
sage: arccsch(2.0)
0.481211825059603
```

```
sage: arccsch(2)
arccsch(2)
sage: arccsch(1 + I*1.0)
0.530637530952518 - 0.452278447151191*I
sage: arccsch(1).n(200)
0.88137358701954302523260932497979230902816032826163541075330
sage: float(arccsch(1))
0.881373587019543

sage: latex(arccsch(x))
{\rm arccsch}\left(x\right)
```

class sage.functions.hyperbolic.**Function_arcsech**
Bases: sage.functions.hyperbolic.HyperbolicFunction

The inverse of the hyperbolic secant function.

EXAMPLES:

```
sage: arcsech(0.5)
1.31695789692482
sage: arcsech(1/2)
arcsech(1/2)
sage: arcsech(1 + I*1.0)
0.530637530952518 - 1.11851787964371*I
sage: arcsech(1/2).n(200)
1.3169578969248167086250463473079684440269819714675164797685
sage: float(arcsech(1/2))
1.3169578969248168

sage: latex(arcsech(x))
{\rm arcsech}\left(x\right)
```

class sage.functions.hyperbolic.**Function_arcsinh**
Bases: sage.symbolic.function.GinacFunction

The inverse of the hyperbolic sine function.

EXAMPLES:

```
sage: arcsinh
arcsinh
sage: arcsinh(0.5)
0.481211825059603
sage: arcsinh(1/2)
arcsinh(1/2)
sage: arcsinh(1 + I*1.0)
1.06127506190504 + 0.666239432492515*I
```

To prevent automatic evaluation use the `hold` argument:

```
sage: arcsinh(-2,hold=True)
arcsinh(-2)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: arcsinh(-2,hold=True).simplify()
-arcsinh(2)
```

`conjugate(arcsinh(x))==arcsinh(conjugate(x))` unless on the branch cuts which run along the imaginary axis outside the interval [-I, +I].:

```
sage: conjugate(arcsinh(x))
conjugate(arcsinh(x))
sage: var('y', domain='positive')
y
sage: conjugate(arcsinh(y))
arcsinh(y)
sage: conjugate(arcsinh(y+I))
conjugate(arcsinh(y + I))
sage: conjugate(arcsinh(1/16))
arcsinh(1/16)
sage: conjugate(arcsinh(I/2))
arcsinh(-1/2*I)
sage: conjugate(arcsinh(2*I))
conjugate(arcsinh(2*I))
```

TESTS:

```
sage: arcsinh(x).operator()
arcsinh
sage: latex(arcsinh(x))
{\rm arcsinh}\left(x\right)
```

class sage.functions.hyperbolic.**Function_arctanh**

Bases: sage.symbolic.function.GinacFunction

The inverse of the hyperbolic tangent function.

EXAMPLES:

```
sage: arctanh(0.5)
0.549306144334055
sage: arctanh(1/2)
arctanh(1/2)
sage: arctanh(1 + I*1.0)
0.402359478108525 + 1.01722196789785*I
```

To prevent automatic evaluation use the `hold` argument:

```
sage: arctanh(-1/2,hold=True)
arctanh(-1/2)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: arctanh(-1/2,hold=True).simplify()
-arctanh(1/2)
```

`conjugate(arctanh(x))==arctanh(conjugate(x))` unless on the branch cuts which run along the real axis outside the interval [-1, +1].:

```
sage: conjugate(arctanh(x))
conjugate(arctanh(x))
sage: var('y', domain='positive')
y
sage: conjugate(arctanh(y))
conjugate(arctanh(y))
sage: conjugate(arctanh(y+I))
conjugate(arctanh(y + I))
sage: conjugate(arctanh(1/16))
arctanh(1/16)
sage: conjugate(arctanh(I/2))
arctanh(-1/2*I)
```

```
sage: conjugate(arctanh(-2*I))
arctanh(2*I)
```

TESTS:

```
sage: arctanh(x).operator()
arctanh
sage: latex(arctanh(x))
{\rm arctanh}\left(x\right)
```

class sage.functions.hyperbolic.**Function_cosh**

Bases: sage.symbolic.function.GinacFunction

The hyperbolic cosine function.

EXAMPLES:

```
sage: cosh(pi)
cosh(pi)
sage: cosh(3.1415)
11.5908832931176
sage: float(cosh(pi))
11.591953275521519
sage: RR(cosh(1/2))
1.12762596520638

sage: latex(cosh(x))
\cosh\left(x\right)
```

To prevent automatic evaluation, use the `hold` parameter:

```
sage: cosh(arcsinh(x),hold=True)
cosh(arcsinh(x))
```

To then evaluate again, we currently must use Maxima via sage.symbolic.expression.Expression.simplify():

```
sage: cosh(arcsinh(x),hold=True).simplify()
sqrt(x^2 + 1)
```

class sage.functions.hyperbolic.**Function_coth**

Bases: sage.functions.hyperbolic.HyperbolicFunction

The hyperbolic cotangent function.

EXAMPLES:

```
sage: coth(pi)
coth(pi)
sage: coth(3.1415)
1.00374256795520
sage: float(coth(pi))
1.0037418731973213
sage: RR(coth(pi))
1.00374187319732

sage: latex(coth(x))
\coth\left(x\right)
```

class sage.functions.hyperbolic.**Function_csch**

Bases: sage.functions.hyperbolic.HyperbolicFunction

The hyperbolic cosecant function.

EXAMPLES:

```
sage: csch(pi)
csch(pi)
sage: csch(3.1415)
0.0865975907592133
sage: float(csch(pi))
0.0865895375300469...
sage: RR(csch(pi))
0.0865895375300470

sage: latex(csch(x))
{\rm csch}\left(x\right)
```

class sage.functions.hyperbolic.**Function_sech**
Bases: sage.functions.hyperbolic.HyperbolicFunction

The hyperbolic secant function.

EXAMPLES:

```
sage: sech(pi)
sech(pi)
sage: sech(3.1415)
0.0862747018248192
sage: float(sech(pi))
0.0862667383340544...
sage: RR(sech(pi))
0.0862667383340544

sage: latex(sech(x))
{\rm sech}\left(x\right)
```

class sage.functions.hyperbolic.**Function_sinh**
Bases: sage.symbolic.function.GinacFunction

The hyperbolic sine function.

EXAMPLES:

```
sage: sinh(pi)
sinh(pi)
sage: sinh(3.1415)
11.5476653707437
sage: float(sinh(pi))
11.54873935725774...
sage: RR(sinh(pi))
11.5487393572577

sage: latex(sinh(x))
\sinh\left(x\right)
```

To prevent automatic evaluation, use the `hold` parameter:

```
sage: sinh(arccosh(x),hold=True)
sinh(arccosh(x))
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: sinh(arccosh(x),hold=True).simplify()
sqrt(x + 1)*sqrt(x - 1)
```

class sage.functions.hyperbolic.**Function_tanh**

Bases: sage.symbolic.function.GinacFunction

The hyperbolic tangent function.

EXAMPLES:

```
sage: tanh(pi)
tanh(pi)
sage: tanh(3.1415)
0.996271386633702
sage: float(tanh(pi))
0.99627207622075
sage: tan(3.1415/4)
0.999953674278156
sage: tanh(pi/4)
tanh(1/4*pi)
sage: RR(tanh(1/2))
0.462117157260010

sage: CC(tanh(pi + I*e))
0.997524731976164 - 0.00279068768100315*I
sage: ComplexField(100)(tanh(pi + I*e))
0.99752473197616361034204366446 - 0.0027906876810031453884245163923*I
sage: CDF(tanh(pi + I*e))  # rel tol 2e-15
0.9975247319761636 - 0.002790687681003147*I
```

To prevent automatic evaluation, use the `hold` parameter:

```
sage: tanh(arcsinh(x),hold=True)
tanh(arcsinh(x))
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: tanh(arcsinh(x),hold=True).simplify()
x/sqrt(x^2 + 1)
```

TESTS:

```
sage: latex(tanh(x))
\tanh\left(x\right)
```

class sage.functions.hyperbolic.**HyperbolicFunction**(*name*, *latex_name=None*, *conversions=None*, *evalf_float=None*)

Bases: sage.symbolic.function.BuiltinFunction

Abstract base class for the functions defined in this file.

EXAMPLES:

```
sage: from sage.functions.hyperbolic import HyperbolicFunction
sage: f = HyperbolicFunction('foo', latex_name='\\foo', conversions={'mathematica':'Foo'},evalf_
sage: f(x)
foo(x)
sage: f(0.5r)
1.0
sage: latex(f(x))
\foo\left(x\right)
```

```
sage: f(x)._mathematica_init_()
'Foo[x]'
```

CHAPTER

FOUR

NUMBER-THEORETIC FUNCTIONS

class sage.functions.transcendental.**DickmanRho**

Bases: sage.symbolic.function.BuiltinFunction

Dickman's function is the continuous function satisfying the differential equation

$$x\rho'(x) + \rho(x-1) = 0$$

with initial conditions $\rho(x) = 1$ for $0 \leq x \leq 1$. It is useful in estimating the frequency of smooth numbers as asymptotically

$$\Psi(a, a^{1/s}) \sim a\rho(s)$$

where $\Psi(a, b)$ is the number of b-smooth numbers less than a.

ALGORITHM:

Dickmans's function is analytic on the interval $[n, n+1]$ for each integer n. To evaluate at $n + t, 0 \leq t < 1$, a power series is recursively computed about $n + 1/2$ using the differential equation stated above. As high precision arithmetic may be needed for intermediate results the computed series are cached for later use.

Simple explicit formulas are used for the intervals [0,1] and [1,2].

EXAMPLES:

```
sage: dickman_rho(2)
0.306852819440055
sage: dickman_rho(10)
2.77017183772596e-11
sage: dickman_rho(10.000000000000000000000000000000000000000)
2.7701718377259589887581212006343423263 4e-11
sage: plot(log(dickman_rho(x)), (x, 0, 15))
Graphics object consisting of 1 graphics primitive
```

AUTHORS:

- Robert Bradshaw (2008-09)

REFERENCES:

- G. Marsaglia, A. Zaman, J. Marsaglia. "Numerical Solutions to some Classical Differential-Difference Equations." Mathematics of Computation, Vol. 53, No. 187 (1989).

approximate(*x*, *parent=None*)

Approximate using de Bruijn's formula

$$\rho(x) \sim \frac{exp(-x\xi + Ei(\xi))}{\sqrt{2\pi x}\xi}$$

which is asymptotically equal to Dickman's function, and is much faster to compute.

REFERENCES:

- N. De Bruijn, "The Asymptotic behavior of a function occurring in the theory of primes." J. Indian Math Soc. v 15. (1951)

EXAMPLES:

```
sage: dickman_rho.approximate(10)
2.41739196365564e-11
sage: dickman_rho(10)
2.77017183772596e-11
sage: dickman_rho.approximate(1000)
4.32938809066403e-3464
```

power_series (*n*, *abs_prec*)

This function returns the power series about $n+1/2$ used to evaluate Dickman's function. It is scaled such that the interval $[n, n+1]$ corresponds to x in $[-1, 1]$.

INPUT:

- `n` - the lower endpoint of the interval for which this power series holds
- `abs_prec` - the absolute precision of the resulting power series

EXAMPLES:

```
sage: f = dickman_rho.power_series(2, 20); f
-9.9376e-8*x^11 + 3.7722e-7*x^10 - 1.4684e-6*x^9 + 5.8783e-6*x^8 - 0.000024259*x^7 + 0.00010
sage: f(-1), f(0), f(1)
(0.30685, 0.13032, 0.048608)
sage: dickman_rho(2), dickman_rho(2.5), dickman_rho(3)
(0.306852819440055, 0.130319561832251, 0.0486083882911316)
```

class `sage.functions.transcendental.`**Function_HurwitzZeta**

Bases: `sage.symbolic.function.BuiltinFunction`

TESTS:

```
sage: latex(hurwitz_zeta(x, 2))
\zeta\left(x, 2\right)
sage: hurwitz_zeta(x, 2)._sympy_()
zeta(x, 2)
```

class `sage.functions.transcendental.`**Function_zeta**

Bases: `sage.symbolic.function.GinacFunction`

Riemann zeta function at s with s a real or complex number.

INPUT:

- `s` - real or complex number

If s is a real number the computation is done using the MPFR library. When the input is not real, the computation is done using the PARI C library.

EXAMPLES:

```
sage: zeta(x)
zeta(x)
sage: zeta(2)
1/6*pi^2
sage: zeta(2.)
1.64493406684823
```

```
sage: RR = RealField(200)
sage: zeta(RR(2))
1.6449340668482264364724151666460251892189499012067984377356
sage: zeta(I)
zeta(I)
sage: zeta(I).n()
0.00330022368532410 - 0.418155449141322*I
```

It is possible to use the `hold` argument to prevent automatic evaluation:

```
sage: zeta(2,hold=True)
zeta(2)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = zeta(2,hold=True); a.simplify()
1/6*pi^2
```

Check that trac ticket #15846 is resolved:

```
sage: zeta(x).series(x==1, 1)
1*(x - 1)^(-1) + (euler_gamma + log(2) + log(pi) + 2*zetaderiv(1, 0)) + Order(x - 1)
sage: zeta(x).residue(x==1)
1
```

TESTS:

```
sage: latex(zeta(x))
\zeta(x)
sage: a = loads(dumps(zeta(x)))
sage: a.operator() == zeta
True

sage: zeta(1)
Infinity
sage: zeta(x).subs(x=1)
Infinity
```

class `sage.functions.transcendental.`**Function_zetaderiv**

Bases: `sage.symbolic.function.GinacFunction`

Derivatives of the Riemann zeta function.

EXAMPLES:

```
sage: zetaderiv(1, x)
zetaderiv(1, x)
sage: zetaderiv(1, x).diff(x)
zetaderiv(2, x)
sage: var('n')
n
sage: zetaderiv(n,x)
zetaderiv(n, x)
sage: zetaderiv(1, 4).n()
-0.0689112658961254
sage: import mpmath; mpmath.diff(lambda x: mpmath.zeta(x), 4)
mpf('-0.068911265896125382')
```

TESTS:

```
sage: latex(zetaderiv(2,x))
\zeta^\prime\left(2, x\right)
sage: a = loads(dumps(zetaderiv(2,x)))
sage: a.operator() == zetaderiv
True
```

sage.functions.transcendental.**hurwitz_zeta**(*s*, *x*, *prec=None*, ***kwargs*)

The Hurwitz zeta function $\zeta(s,x)$, where s and x are complex.

The Hurwitz zeta function is one of the many zeta functions. It defined as

$$\zeta(s,x) = \sum_{k=0}^{\infty}(k+x)^{-s}.$$

When $x = 1$, this coincides with Riemann's zeta function. The Dirichlet L-functions may be expressed as a linear combination of Hurwitz zeta functions.

EXAMPLES:

Symbolic evaluations:

```
sage: hurwitz_zeta(x, 1)
zeta(x)
sage: hurwitz_zeta(4, 3)
1/90*pi^4 - 17/16
sage: hurwitz_zeta(-4, x)
-1/5*x^5 + 1/2*x^4 - 1/3*x^3 + 1/30*x
sage: hurwitz_zeta(7, -1/2)
127*zeta(7) - 128
sage: hurwitz_zeta(-3, 1)
1/120
```

Numerical evaluations:

```
sage: hurwitz_zeta(3, 1/2).n()
8.41439832211716
sage: hurwitz_zeta(11/10, 1/2).n()
12.1038134956837
sage: hurwitz_zeta(3, x).series(x, 60).subs(x=0.5).n()
8.41439832211716
sage: hurwitz_zeta(3, 0.5)
8.41439832211716
```

REFERENCES:

- Wikipedia article Hurwitz_zeta_function

sage.functions.transcendental.**zeta_symmetric**(*s*)

Completed function $\xi(s)$ that satisfies $\xi(s) = \xi(1-s)$ and has zeros at the same points as the Riemann zeta function.

INPUT:

- s - real or complex number

If s is a real number the computation is done using the MPFR library. When the input is not real, the computation is done using the PARI C library.

More precisely,

$$xi(s) = \gamma(s/2+1) * (s-1) * \pi^{-s/2} * \zeta(s).$$

EXAMPLES:

```
sage: zeta_symmetric(0.7)
0.497580414651127
sage: zeta_symmetric(1-0.7)
0.497580414651127
sage: RR = RealField(200)
sage: zeta_symmetric(RR(0.7))
0.49758041465112690357779107525638385212657443284080589766062
sage: C.<i> = ComplexField()
sage: zeta_symmetric(0.5 + i*14.0)
0.000201294444235258 + 1.49077798716757e-19*I
sage: zeta_symmetric(0.5 + i*14.1)
0.0000489893483255687 + 4.40457132572236e-20*I
sage: zeta_symmetric(0.5 + i*14.2)
-0.0000868931282620101 + 7.11507675693612e-20*I
```

REFERENCE:

- I copied the definition of xi from http://web.viu.ca/pughg/RiemannZeta/RiemannZetaLong.html

CHAPTER

FIVE

PIECEWISE-DEFINED FUNCTIONS

Sage implements a very simple class of piecewise-defined functions. Functions may be any type of symbolic expression. Infinite intervals are not supported. The endpoints of each interval must line up.

TODO:

- Implement max/min location and values,
- Need: parent object - ring of piecewise functions
- This class should derive from an element-type class, and should define `_add_`, `_mul_`, etc. That will automatically take care of left multiplication and proper coercion. The coercion mentioned below for scalar mult on right is bad, since it only allows ints and rationals. The right way is to use an element class and only define `_mul_`, and have a parent, so anything gets coerced properly.

AUTHORS:

- David Joyner (2006-04): initial version
- David Joyner (2006-09): added __eq__, extend_by_zero_to, unextend, convolution, trapezoid, trapezoid_integral_approximation, riemann_sum, riemann_sum_integral_approximation, tangent_line fixed bugs in __mul__, __add__
- David Joyner (2007-03): adding Hann filter for FS, added general FS filter methods for computing and plotting, added options to plotting of FS (eg, specifying rgb values are now allowed). Fixed bug in documentation reported by Pablo De Napoli.
- David Joyner (2007-09): bug fixes due to behaviour of SymbolicArithmetic
- David Joyner (2008-04): fixed docstring bugs reported by J Morrow; added support for Laplace transform of functions with infinite support.
- David Joyner (2008-07): fixed a left multiplication bug reported by C. Boncelet (by defining __rmul__ = __mul__).
- Paul Butler (2009-01): added indefinite integration and default_variable

TESTS:

```
sage: R.<x> = QQ[]
sage: f = Piecewise([[(0,1),1*x^0]])
sage: 2*f
Piecewise defined function with 1 parts, [[(0, 1), 2]]
```

`sage.functions.piecewise.`**`Piecewise`**(*list_of_pairs*, *var=None*)

Returns a piecewise function from a list of (interval, function) pairs.

`list_of_pairs` is a list of pairs (I, fcn), where fcn is a Sage function (such as a polynomial over RR, or functions using the lambda notation), and I is an interval such as I = (1,3). Two consecutive intervals must share a common endpoint.

If the optional `var` is specified, then any symbolic expressions in the list will be converted to symbolic functions using `fcn.function(var)`. (This says which variable is considered to be "piecewise".)

We assume that these definitions are consistent (ie, no checking is done).

EXAMPLES:

```
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: f(1)
-1
sage: f(3)
2
sage: f = Piecewise([[(0,1),x], [(1,2),x^2]], x); f
Piecewise defined function with 2 parts, [[(0, 1), x |--> x], [(1, 2), x |--> x^2]]
sage: f(0.9)
0.900000000000000
sage: f(1.1)
1.21000000000000
```

class sage.functions.piecewise.**PiecewisePolynomial**(*list_of_pairs*, *var=None*)

Returns a piecewise function from a list of (interval, function) pairs.

EXAMPLES:

```
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: f(1)
-1
sage: f(3)
2
```

base_ring()

Returns the base ring of the function pieces. This is useful when this class is extended.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f3(x) = x^2-5
sage: f = Piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3]])
sage: base_ring(f)
Symbolic Ring

sage: R.<x> = QQ[]
sage: f1 = x^0
sage: f2 = 10*x - x^2
sage: f3 = 3*x^4 - 156*x^3 + 3036*x^2 - 26208*x
sage: f = Piecewise([[(0,3),f1],[(3,10),f2],[(10,20),f3]])
sage: f.base_ring()
Rational Field
```

convolution(*other*)

Returns the convolution function, $f * g(t) = \int_{-\infty}^{\infty} f(u)g(t-u)du$, for compactly supported f, g.

EXAMPLES:

```
sage: x = PolynomialRing(QQ,'x').gen()
sage: f = Piecewise([[(0,1),1*x^0]])  ## example 0
sage: g = f.convolution(f)
```

```
sage: h = f.convolution(g)
sage: P = f.plot(); Q = g.plot(rgbcolor=(1,1,0)); R = h.plot(rgbcolor=(0,1,1));
sage: # Type show(P+Q+R) to view
sage: f = Piecewise([[(0,1),1*x^0],[(1,2),2*x^0],[(2,3),1*x^0]])  ## example 1
sage: g = f.convolution(f)
sage: h = f.convolution(g)
sage: P = f.plot(); Q = g.plot(rgbcolor=(1,1,0)); R = h.plot(rgbcolor=(0,1,1));
sage: # Type show(P+Q+R) to view
sage: f = Piecewise([[(-1,1),1]])                              ## example 2
sage: g = Piecewise([[(0,3),x]])
sage: f.convolution(g)
Piecewise defined function with 3 parts, [[(-1, 1), 0], [(1, 2), -3/2*x], [(2, 4), -3/2*x]]
sage: g = Piecewise([[(0,3),1*x^0],[(3,4),2*x^0]])
sage: f.convolution(g)
Piecewise defined function with 5 parts, [[(-1, 1), x + 1], [(1, 2), 3], [(2, 3), x], [(3, 4
```

cosine_series_coefficient(*n*, *L*)

Returns the n-th cosine series coefficient of $\cos(n\pi x/L)$, a_n.

INPUT:

- `self` - the function f(x), defined over 0 x L (no checking is done to insure this)
- `n` - an integer n=0
- `L` - (the period)/2

OUTPUT: $a_n = \frac{2}{L}\int_{-L}^{L} f(x)\cos(n\pi x/L)dx$ such that

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(\frac{n\pi x}{L}),\ \ 0 < x < L.$$

EXAMPLES:

```
sage: f(x) = x
sage: f = Piecewise([[(0,1),f]])
sage: f.cosine_series_coefficient(2,1)
0
sage: f.cosine_series_coefficient(3,1)
-4/9/pi^2
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: f.cosine_series_coefficient(2,pi)
0
sage: f.cosine_series_coefficient(3,pi)
2/pi
sage: f.cosine_series_coefficient(111,pi)
2/37/pi
sage: f1 = lambda x: x*(pi-x)
sage: f = Piecewise([[(0,pi),f1]])
sage: f.cosine_series_coefficient(0,pi)
1/3*pi^2
```

critical_points()

Return the critical points of this piecewise function.

> **Warning:** Uses maxima, which prints the warning to use results with caution. Only works for piecewise functions whose parts are polynomials with real critical not occurring on the interval endpoints.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f1 = x^0
sage: f2 = 10*x - x^2
sage: f3 = 3*x^4 - 156*x^3 + 3036*x^2 - 26208*x
sage: f = Piecewise([[(0,3),f1],[(3,10),f2],[(10,20),f3]])
sage: expected = [5, 12, 13, 14]
sage: all(abs(e-a) < 0.001 for e,a in zip(expected, f.critical_points()))
True
```

TESTS:

Use variables other than x (trac ticket #13836):

```
sage: R.<y> = QQ[]
sage: f1 = y^0
sage: f2 = 10*y - y^2
sage: f3 = 3*y^4 - 156*y^3 + 3036*y^2 - 26208*y
sage: f = Piecewise([[(0,3),f1],[(3,10),f2],[(10,20),f3]])
sage: expected = [5, 12, 13, 14]
sage: all(abs(e-a) < 0.001 for e,a in zip(expected, f.critical_points()))
True
```

default_variable()

Return the default variable. The default variable is defined as the first variable in the first piece that has a variable. If no pieces have a variable (each piece is a constant value), x is returned.

The result is cached.

AUTHOR: Paul Butler

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 5*x
sage: p = Piecewise([[(0,1),f1],[(1,4),f2]])
sage: p.default_variable()
x

sage: f1 = 3*var('y')
sage: p = Piecewise([[(0,1),4],[(1,4),f1]])
sage: p.default_variable()
y
```

derivative()

Returns the derivative (as computed by maxima) Piecewise(I,‘(d/dx)(selfl_I)‘), as I runs over the intervals belonging to self. self must be piecewise polynomial.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.derivative()
Piecewise defined function with 2 parts, [[(0, 1), x |--> 0], [(1, 2), x |--> -1]]
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: f.derivative()
Piecewise defined function with 2 parts, [[(0, 1/2*pi), x |--> 0], [(1/2*pi, pi), x |--> 0]]
```

```
sage: f = Piecewise([[(0,1), (x * 2)]], x)
sage: f.derivative()
Piecewise defined function with 1 parts, [[(0, 1), x |--> 2]]
```

domain()

Returns the domain of the function.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f3(x) = exp(x)
sage: f4(x) = sin(2*x)
sage: f = Piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3],[(3,10),f4]])
sage: f.domain()
(0, 10)
```

end_points()

Returns a list of all interval endpoints for this function.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f3(x) = x^2-5
sage: f = Piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3]])
sage: f.end_points()
[0, 1, 2, 3]
```

extend_by_zero_to(*xmin=-1000*, *xmax=1000*)

This function simply returns the piecewise defined function which is extended by 0 so it is defined on all of (xmin,xmax). This is needed to add two piecewise functions in a reasonable way.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1 - x
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.extend_by_zero_to(-1, 3)
Piecewise defined function with 4 parts, [[(-1, 0), 0], [(0, 1), x |--> 1], [(1, 2), x |-->
```

fourier_series_cosine_coefficient(*n*, *L*)

Returns the n-th Fourier series coefficient of $\cos(n\pi x/L)$, a_n.

INPUT:

- `self` - the function f(x), defined over -L x L
- `n` - an integer n=0
- `L` - (the period)/2

OUTPUT: $a_n = \frac{1}{L}\int_{-L}^{L} f(x)\cos(n\pi x/L)dx$

EXAMPLES:

```
sage: f(x) = x^2
sage: f = Piecewise([[(-1,1),f]])
sage: f.fourier_series_cosine_coefficient(2,1)
pi^(-2)
sage: f(x) = x^2
sage: f = Piecewise([[(-pi,pi),f]])
```

```
sage: f.fourier_series_cosine_coefficient(2,pi)
1
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: f.fourier_series_cosine_coefficient(5,pi)
-3/5/pi
```

fourier_series_partial_sum(*N*, *L*)

Returns the partial sum

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N} [a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

as a string.

EXAMPLE:

```
sage: f(x) = x^2
sage: f = Piecewise([[(-1,1),f]])
sage: f.fourier_series_partial_sum(3,1)
cos(2*pi*x)/pi^2 - 4*cos(pi*x)/pi^2 + 1/3
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: f.fourier_series_partial_sum(3,pi)
-3*cos(x)/pi - 3*sin(2*x)/pi + 3*sin(x)/pi - 1/4
```

fourier_series_partial_sum_cesaro(*N*, *L*)

Returns the Cesaro partial sum

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N} (1 - n/N) * [a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

as a string. This is a "smoother" partial sum - the Gibbs phenomenon is mollified.

EXAMPLE:

```
sage: f(x) = x^2
sage: f = Piecewise([[(-1,1),f]])
sage: f.fourier_series_partial_sum_cesaro(3,1)
1/3*cos(2*pi*x)/pi^2 - 8/3*cos(pi*x)/pi^2 + 1/3
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: f.fourier_series_partial_sum_cesaro(3,pi)
-2*cos(x)/pi - sin(2*x)/pi + 2*sin(x)/pi - 1/4
```

fourier_series_partial_sum_filtered(*N*, *L*, *F*)

Returns the "filtered" partial sum

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N} F_n * [a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

as a string, where $F = [F_1, F_2, ..., F_N]$ is a list of length N consisting of real numbers. This can be used to plot FS solutions to the heat and wave PDEs.

EXAMPLE:

```
sage: f(x) = x^2
sage: f = Piecewise([[(-1,1),f]])
sage: f.fourier_series_partial_sum_filtered(3,1,[1,1,1])
cos(2*pi*x)/pi^2 - 4*cos(pi*x)/pi^2 + 1/3
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: f.fourier_series_partial_sum_filtered(3,pi,[1,1,1])
-3*cos(x)/pi - 3*sin(2*x)/pi + 3*sin(x)/pi - 1/4
```

fourier_series_partial_sum_hann (N, L)

Returns the Hann-filtered partial sum (named after von Hann, not Hamming)

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N} H_N(n) * [a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

as a string, where $H_N(x) = (1+\cos(\pi x/N))/2$. This is a "smoother" partial sum - the Gibbs phenomenon is mollified.

EXAMPLE:

```
sage: f(x) = x^2
sage: f = Piecewise([[(-1,1),f]])
sage: f.fourier_series_partial_sum_hann(3,1)
1/4*cos(2*pi*x)/pi^2 - 3*cos(pi*x)/pi^2 + 1/3
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: f.fourier_series_partial_sum_hann(3,pi)
-9/4*cos(x)/pi - 3/4*sin(2*x)/pi + 9/4*sin(x)/pi - 1/4
```

fourier_series_sine_coefficient (n, L)

Returns the n-th Fourier series coefficient of $\sin(n\pi x/L)$, b_n.

INPUT:

- `self` - the function f(x), defined over -L x L
- `n` - an integer n0
- `L` - (the period)/2

OUTPUT: $b_n = \frac{1}{L}\int_{-L}^{L} f(x)\sin(n\pi x/L)dx$

EXAMPLES:

```
sage: f(x) = x^2
sage: f = Piecewise([[(-1,1),f]])
sage: f.fourier_series_sine_coefficient(2,1)  # L=1, n=2
0
```

fourier_series_value (x, L)

Returns the value of the Fourier series coefficient of self at x,

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{\infty}[a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})], \quad -L < x < L.$$

This method applies to piecewise non-polynomial functions as well.

INPUT:

- `self` - the function f(x), defined over -L x L
- `x` - a real number
- `L` - (the period)/2

OUTPUT: $(f^*(x+) + f^*(x-)/2$, where f^* denotes the function f extended to $\mathbf{R}$ with period $2L$ (Dirichlet's Theorem for Fourier series).

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f3(x) = exp(x)
sage: f4(x) = sin(2*x)
sage: f = Piecewise([[(-10,1),f1],[(1,2),f2],[(2,3),f3],[(3,10),f4]])
sage: f.fourier_series_value(101,10)
1/2
sage: f.fourier_series_value(100,10)
1
sage: f.fourier_series_value(10,10)
1/2*sin(20) + 1/2
sage: f.fourier_series_value(20,10)
1
sage: f.fourier_series_value(30,10)
1/2*sin(20) + 1/2
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,0),lambda x:0],[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: f.fourier_series_value(-1,pi)
0
sage: f.fourier_series_value(20,pi)
-1
sage: f.fourier_series_value(pi/2,pi)
1/2
```

functions()

Returns the list of functions (the "pieces").

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f3(x) = exp(x)
sage: f4(x) = sin(2*x)
sage: f = Piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3],[(3,10),f4]])
sage: f.functions()
[x |--> 1, x |--> -x + 1, x |--> e^x, x |--> sin(2*x)]
```

integral(*x=None*, *a=None*, *b=None*, *definite=False*)

By default, returns the indefinite integral of the function. If definite=True is given, returns the definite integral.

AUTHOR:

- Paul Butler

EXAMPLES:

```
sage: f1(x) = 1-x
sage: f = Piecewise([[(0,1),1],[(1,2),f1]])
sage: f.integral(definite=True)
1/2
```

```
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: f.integral(definite=True)
1/2*pi

sage: f1(x) = 2
sage: f2(x) = 3 - x
sage: f = Piecewise([[(-2, 0), f1], [(0, 3), f2]])
sage: f.integral()
Piecewise defined function with 2 parts, [[(-2, 0), x |--> 2*x + 4], [(0, 3), x |--> -1/2*x^

sage: f1(y) = -1
sage: f2(y) = y + 3
sage: f3(y) = -y - 1
sage: f4(y) = y^2 - 1
sage: f5(y) = 3
sage: f = Piecewise([[(-4,-3),f1],[(-3,-2),f2],[(-2,0),f3],[(0,2),f4],[(2,3),f5]])
sage: F = f.integral(y)
sage: F
Piecewise defined function with 5 parts, [[(-4, -3), y |--> -y - 4], [(-3, -2), y |--> 1/2*y
```

Ensure results are consistent with FTC:

```
sage: F(-3) - F(-4)
-1
sage: F(-1) - F(-3)
1
sage: F(2) - F(0)
2/3
sage: f.integral(y, 0, 2)
2/3
sage: F(3) - F(-4)
19/6
sage: f.integral(y, -4, 3)
19/6
sage: f.integral(definite=True)
19/6

sage: f1(y) = (y+3)^2
sage: f2(y) = y+3
sage: f3(y) = 3
sage: f = Piecewise([[(-infinity, -3), f1], [(-3, 0), f2], [(0, infinity), f3]])
sage: f.integral()
Piecewise defined function with 3 parts, [[(-Infinity, -3), y |--> 1/3*y^3 + 3*y^2 + 9*y + 9

sage: f1(x) = e^(-abs(x))
sage: f = Piecewise([[(-infinity, infinity), f1]])
sage: f.integral(definite=True)
2
sage: f.integral()
Piecewise defined function with 1 parts, [[(-Infinity, +Infinity), x |--> -1/2*((sgn(x) - 1)

sage: f = Piecewise([((0, 5), cos(x))])
sage: f.integral()
Piecewise defined function with 1 parts, [[(0, 5), x |--> sin(x)]]
```

TESTS:

Verify that piecewise integrals of zero work (trac #10841):

```
sage: f0(x) = 0
sage: f = Piecewise([[(0,1),f0]])
sage: f.integral(x,0,1)
0
sage: f = Piecewise([[(0,1), 0]])
sage: f.integral(x,0,1)
0
sage: f = Piecewise([[(0,1), SR(0)]])
sage: f.integral(x,0,1)
0
```

intervals()

A piecewise non-polynomial example.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f3(x) = exp(x)
sage: f4(x) = sin(2*x)
sage: f = Piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3],[(3,10),f4]])
sage: f.intervals()
[(0, 1), (1, 2), (2, 3), (3, 10)]
```

laplace(*x='x'*, *s='t'*)

Returns the Laplace transform of self with respect to the variable var.

INPUT:

- x - variable of self
- s - variable of Laplace transform.

We assume that a piecewise function is 0 outside of its domain and that the left-most endpoint of the domain is 0.

EXAMPLES:

```
sage: x, s, w = var('x, s, w')
sage: f = Piecewise([[(0,1),1],[(1,2), 1-x]])
sage: f.laplace(x, s)
-e^(-s)/s + (s + 1)*e^(-2*s)/s^2 + 1/s - e^(-s)/s^2
sage: f.laplace(x, w)
-e^(-w)/w + (w + 1)*e^(-2*w)/w^2 + 1/w - e^(-w)/w^2

sage: y, t = var('y, t')
sage: f = Piecewise([[(1,2), 1-y]])
sage: f.laplace(y, t)
(t + 1)*e^(-2*t)/t^2 - e^(-t)/t^2

sage: s = var('s')
sage: t = var('t')
sage: f1(t) = -t
sage: f2(t) = 2
sage: f = Piecewise([[(0,1),f1],[(1,infinity),f2]])
sage: f.laplace(t,s)
(s + 1)*e^(-s)/s^2 + 2*e^(-s)/s - 1/s^2
```

length()
Returns the number of pieces of this function.

EXAMPLES:
```
sage: f1(x) = 1
sage: f2(x) = 1 - x
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.length()
2
```

list()
Returns the pieces of this function as a list of functions.

EXAMPLES:
```
sage: f1(x) = 1
sage: f2(x) = 1 - x
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.list()
[[(0, 1), x |--> 1], [(1, 2), x |--> -x + 1]]
```

plot(**args*, ***kwds*)
Returns the plot of self.

Keyword arguments are passed onto the plot command for each piece of the function. E.g., the plot_points keyword affects each segment of the plot.

EXAMPLES:
```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f3(x) = exp(x)
sage: f4(x) = sin(2*x)
sage: f = Piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3],[(3,10),f4]])
sage: P = f.plot(rgbcolor=(0.7,0.1,0), plot_points=40)
sage: P
Graphics object consisting of 4 graphics primitives
```

Remember: to view this, type show(P) or P.save("path/myplot.png") and then open it in a graphics viewer such as GIMP.

TESTS:

We should not add each piece to the legend individually, since this creates duplicates (trac ticket #12651). This tests that only one of the graphics objects in the plot has a non-`None` `legend_label`:
```
sage: f1 = sin(x)
sage: f2 = cos(x)
sage: f = piecewise([[(-1,0), f1],[(0,1), f2]])
sage: p = f.plot(legend_label='$f(x)$')
sage: lines = [
...       line
...       for line in p._objects
...       if line.options()['legend_label'] is not None ]
sage: len(lines)
1
```

plot_fourier_series_partial_sum(*N*, *L*, *xmin*, *xmax*, ***kwds*)
Plots the partial sum

$$f(x) \sim \frac{a_0}{2} + sum_{n=1}^{N}[a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

over xmin x xmin.

EXAMPLE:

```
sage: f1(x) = -2
sage: f2(x) = 1
sage: f3(x) = -1
sage: f4(x) = 2
sage: f = Piecewise([[(-pi,-pi/2),f1],[(-pi/2,0),f2],[(0,pi/2),f3],[(pi/2,pi),f4]])
sage: P = f.plot_fourier_series_partial_sum(3,pi,-5,5)     # long time
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: P = f.plot_fourier_series_partial_sum(15,pi,-5,5)    # long time
```

Remember, to view this type show(P) or P.save("path/myplot.png") and then open it in a graphics viewer such as GIMP.

plot_fourier_series_partial_sum_cesaro(*N*, *L*, *xmin*, *xmax*, ***kwds*)

Plots the partial sum

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N} (1 - n/N) * [a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

over xmin x xmin. This is a "smoother" partial sum - the Gibbs phenomenon is mollified.

EXAMPLE:

```
sage: f1(x) = -2
sage: f2(x) = 1
sage: f3(x) = -1
sage: f4(x) = 2
sage: f = Piecewise([[(-pi,-pi/2),f1],[(-pi/2,0),f2],[(0,pi/2),f3],[(pi/2,pi),f4]])
sage: P = f.plot_fourier_series_partial_sum_cesaro(3,pi,-5,5)     # long time
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: P = f.plot_fourier_series_partial_sum_cesaro(15,pi,-5,5)    # long time
```

Remember, to view this type show(P) or P.save("path/myplot.png") and then open it in a graphics viewer such as GIMP.

plot_fourier_series_partial_sum_filtered(*N*, *L*, *F*, *xmin*, *xmax*, ***kwds*)

Plots the partial sum

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N} F_n * [a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

over xmin x xmin, where $F = [F_1, F_2, ..., F_N]$ is a list of length N consisting of real numbers. This can be used to plot FS solutions to the heat and wave PDEs.

EXAMPLE:

```
sage: f1(x) = -2
sage: f2(x) = 1
sage: f3(x) = -1
sage: f4(x) = 2
sage: f = Piecewise([[(-pi,-pi/2),f1],[(-pi/2,0),f2],[(0,pi/2),f3],[(pi/2,pi),f4]])
sage: P = f.plot_fourier_series_partial_sum_filtered(3,pi,[1]*3,-5,5)     # long time
sage: f1(x) = -1
```

```
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,-pi/2),f1],[(-pi/2,0),f2],[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: P = f.plot_fourier_series_partial_sum_filtered(15,pi,[1]*15,-5,5)    # long time
```

Remember, to view this type show(P) or P.save("path/myplot.png") and then open it in a graphics viewer such as GIMP.

plot_fourier_series_partial_sum_hann (*N*, *L*, *xmin*, *xmax*, ***kwds*)

Plots the partial sum

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N} H_N(n) * [a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

over xmin x xmin, where H_N(x) = (0.5)+(0.5)*cos(x*pi/N) is the N-th Hann filter.

EXAMPLE:

```
sage: f1(x) = -2
sage: f2(x) = 1
sage: f3(x) = -1
sage: f4(x) = 2
sage: f = Piecewise([[(-pi,-pi/2),f1],[(-pi/2,0),f2],[(0,pi/2),f3],[(pi/2,pi),f4]])
sage: P = f.plot_fourier_series_partial_sum_hann(3,pi,-5,5)     # long time
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: P = f.plot_fourier_series_partial_sum_hann(15,pi,-5,5)    # long time
```

Remember, to view this type show(P) or P.save("path/myplot.png") and then open it in a graphics viewer such as GIMP.

riemann_sum (*N*, *mode=None*)

Returns the piecewise line function defined by the Riemann sums in numerical integration based on a subdivision into N subintervals. Set mode="midpoint" for the height of the rectangles to be determined by the midpoint of the subinterval; set mode="right" for the height of the rectangles to be determined by the right-hand endpoint of the subinterval; the default is mode="left" (the height of the rectangles to be determined by the left-hand endpoint of the subinterval).

EXAMPLES:

```
sage: f1(x) = x^2
sage: f2(x) = 5-x^2
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.riemann_sum(6,mode="midpoint")
Piecewise defined function with 6 parts, [[(0, 1/3), 1/36], [(1/3, 2/3), 1/4], [(2/3, 1), 25

sage: f = Piecewise([[(-1,1),(1-x^2).function(x)]])
sage: rsf = f.riemann_sum(7)
sage: P = f.plot(rgbcolor=(0.7,0.1,0.5), plot_points=40)
sage: Q = rsf.plot(rgbcolor=(0.7,0.6,0.6), plot_points=40)
sage: L = add([line([[a,0],[a,f(x=a)]],rgbcolor=(0.7,0.6,0.6)) for (a,b),f in rsf.list()])
sage: P + Q + L
Graphics object consisting of 15 graphics primitives

sage: f = Piecewise([[(-1,1),(1/2+x-x^3)]], x) ## example 3
sage: rsf = f.riemann_sum(8)
sage: P = f.plot(rgbcolor=(0.7,0.1,0.5), plot_points=40)
sage: Q = rsf.plot(rgbcolor=(0.7,0.6,0.6), plot_points=40)
sage: L = add([line([[a,0],[a,f(x=a)]],rgbcolor=(0.7,0.6,0.6)) for (a,b),f in rsf.list()])
```

```
sage: P + Q + L
Graphics object consisting of 17 graphics primitives
```

riemann_sum_integral_approximation(*N*, *mode=None*)

Returns the piecewise line function defined by the Riemann sums in numerical integration based on a subdivision into N subintervals.

Set mode="midpoint" for the height of the rectangles to be determined by the midpoint of the subinterval; set mode="right" for the height of the rectangles to be determined by the right-hand endpoint of the subinterval; the default is mode="left" (the height of the rectangles to be determined by the left-hand endpoint of the subinterval).

EXAMPLES:

```
sage: f1(x) = x^2                    ## example 1
sage: f2(x) = 5-x^2
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.riemann_sum_integral_approximation(6)
17/6
sage: f.riemann_sum_integral_approximation(6,mode="right")
19/6
sage: f.riemann_sum_integral_approximation(6,mode="midpoint")
3
sage: f.integral(definite=True)
3
```

sine_series_coefficient(*n*, *L*)

Returns the n-th sine series coefficient of $\sin(n\pi x/L)$, b_n.

INPUT:

- `self` - the function f(x), defined over 0 x L (no checking is done to insure this)
- `n` - an integer n0
- `L` - (the period)/2

OUTPUT:

$b_n = \frac{2}{L}\int_{-L}^{L} f(x)\sin(n\pi x/L)dx$ such that

$$f(x) \sim \sum_{n=1}^{\infty} b_n \sin(\frac{n\pi x}{L}),\ \ 0 < x < L.$$

EXAMPLES:

```
sage: f(x) = 1
sage: f = Piecewise([[(0,1),f]])
sage: f.sine_series_coefficient(2,1)
0
sage: f.sine_series_coefficient(3,1)
4/3/pi
```

tangent_line(*pt*)

Computes the linear function defining the tangent line of the piecewise function self.

EXAMPLES:

```
sage: f1(x) = x^2
sage: f2(x) = 5-x^3+x
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
```

```
sage: tf = f.tangent_line(0.9) ## tangent line at x=0.9
sage: P = f.plot(rgbcolor=(0.7,0.1,0.5), plot_points=40)
sage: Q = tf.plot(rgbcolor=(0.7,0.2,0.2), plot_points=40)
sage: P + Q
Graphics object consisting of 4 graphics primitives
```

trapezoid(*N*)

Returns the piecewise line function defined by the trapezoid rule for numerical integration based on a subdivision into N subintervals.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f1 = x^2
sage: f2 = 5-x^2
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.trapezoid(4)
Piecewise defined function with 4 parts, [[(0, 1/2), 1/2*x], [(1/2, 1), 9/2*x - 2], [(1, 3/2
```

```
sage: R.<x> = QQ[]
sage: f = Piecewise([[(-1,1),1-x^2]])
sage: tf = f.trapezoid(4)
sage: P = f.plot(rgbcolor=(0.7,0.1,0.5), plot_points=40)
sage: Q = tf.plot(rgbcolor=(0.7,0.6,0.6), plot_points=40)
sage: L = add([line([[a,0],[a,f(a)]],rgbcolor=(0.7,0.6,0.6)) for (a,b),f in tf.list()])
sage: P+Q+L
Graphics object consisting of 9 graphics primitives
```

```
sage: R.<x> = QQ[]
sage: f = Piecewise([[(-1,1),1/2+x-x^3]]) ## example 3
sage: tf = f.trapezoid(6)
sage: P = f.plot(rgbcolor=(0.7,0.1,0.5), plot_points=40)
sage: Q = tf.plot(rgbcolor=(0.7,0.6,0.6), plot_points=40)
sage: L = add([line([[a,0],[a,f(a)]],rgbcolor=(0.7,0.6,0.6)) for (a,b),f in tf.list()])
sage: P+Q+L
Graphics object consisting of 13 graphics primitives
```

TESTS:

Use variables other than x (trac ticket #13836):

```
sage: R.<y> = QQ[]
sage: f1 = y^2
sage: f2 = 5-y^2
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.trapezoid(4)
Piecewise defined function with 4 parts, [[(0, 1/2), 1/2*y], [(1/2, 1), 9/2*y - 2], [(1, 3/2
```

trapezoid_integral_approximation(*N*)

Returns the approximation given by the trapezoid rule for numerical integration based on a subdivision into N subintervals.

EXAMPLES:

```
sage: f1(x) = x^2                       ## example 1
sage: f2(x) = 1-(1-x)^2
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: P = f.plot(rgbcolor=(0.7,0.1,0.5), plot_points=40)
sage: tf = f.trapezoid(6)
sage: Q = tf.plot(rgbcolor=(0.7,0.6,0.6), plot_points=40)
```

```
sage: ta = f.trapezoid_integral_approximation(6)
sage: t = text('trapezoid approximation = %s'%ta, (1.5, 0.25))
sage: a = f.integral(definite=True)
sage: tt = text('area under curve = %s'%a, (1.5, -0.5))
sage: P + Q + t + tt
Graphics object consisting of 10 graphics primitives

sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])  ## example 2
sage: tf = f.trapezoid(4)
sage: ta = f.trapezoid_integral_approximation(4)
sage: Q = tf.plot(rgbcolor=(0.7,0.6,0.6), plot_points=40)
sage: t = text('trapezoid approximation = %s'%ta, (1.5, 0.25))
sage: a = f.integral(definite=True)
sage: tt = text('area under curve = %s'%a, (1.5, -0.5))
sage: P+Q+t+tt
Graphics object consisting of 8 graphics primitives
```

unextend()

This removes any parts in the front or back of the function which is zero (the inverse to extend_by_zero_to).

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = Piecewise([[(-3,-1),1+2+x],[(-1,1),1-x^2]])
sage: e = f.extend_by_zero_to(-10,10); e
Piecewise defined function with 4 parts, [[(-10, -3), 0], [(-3, -1), x + 3], [(-1, 1), -x^2
sage: d = e.unextend(); d
Piecewise defined function with 2 parts, [[(-3, -1), x + 3], [(-1, 1), -x^2 + 1]]
sage: d==f
True
```

which_function(*x0*)

Returns the function piece used to evaluate self at x0.

EXAMPLES:

```
sage: f1(z) = z
sage: f2(x) = 1-x
sage: f3(y) = exp(y)
sage: f4(t) = sin(2*t)
sage: f = Piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3],[(3,10),f4]])
sage: f.which_function(3/2)
x |--> -x + 1
```

sage.functions.piecewise.**piecewise**(*list_of_pairs*, *var=None*)

Returns a piecewise function from a list of (interval, function) pairs.

`list_of_pairs` is a list of pairs (I, fcn), where fcn is a Sage function (such as a polynomial over RR, or functions using the lambda notation), and I is an interval such as I = (1,3). Two consecutive intervals must share a common endpoint.

If the optional `var` is specified, then any symbolic expressions in the list will be converted to symbolic functions using `fcn.function(var)`. (This says which variable is considered to be "piecewise".)

We assume that these definitions are consistent (ie, no checking is done).

EXAMPLES:

```
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(0,pi/2),f1],[(pi/2,pi),f2]])
```

```
sage: f(1)
-1
sage: f(3)
2
sage: f = Piecewise([[(0,1),x], [(1,2),x^2]], x); f
Piecewise defined function with 2 parts, [[(0, 1), x |--> x], [(1, 2), x |--> x^2]]
sage: f(0.9)
0.900000000000000
sage: f(1.1)
1.21000000000000
```

CHAPTER

SIX

SPIKE FUNCTIONS

AUTHORS:

- William Stein (2007-07): initial version
- Karl-Dieter Crisman (2009-09): adding documentation and doctests

class sage.functions.spike_function.**SpikeFunction**(*v*, *eps=1e-07*)

Base class for spike functions.

INPUT:

- v - list of pairs (x, height)
- eps - parameter that determines approximation to a true spike

OUTPUT:

a function with spikes at each point x in v with the given height.

EXAMPLES:

```
sage: spike_function([(-3,4),(-1,1),(2,3)],0.001)
A spike function with spikes at [-3.0, -1.0, 2.0]
```

Putting the spikes too close together may delete some:

```
sage: spike_function([(1,1),(1.01,4)],0.1)
Some overlapping spikes have been deleted.
You might want to use a smaller value for eps.
A spike function with spikes at [1.0]
```

Note this should normally be used indirectly via spike_function, but one can use it directly:

```
sage: from sage.functions.spike_function import SpikeFunction
sage: S = SpikeFunction([(0,1),(1,2),(pi,-5)])
sage: S
A spike function with spikes at [0.0, 1.0, 3.141592653589793]
sage: S.support
[0.0, 1.0, 3.141592653589793]
```

plot(*xmin=None*, *xmax=None*, ***kwds*)

Special fast plot method for spike functions.

EXAMPLES:

```
sage: S = spike_function([(-1,1),(1,40)])
sage: P = plot(S)
sage: P[0]
Line defined by 8 points
```

plot_fft_abs (*samples=4096*, *xmin=None*, *xmax=None*, ***kwds*)
Plot of (absolute values of) Fast Fourier Transform of the spike function with given number of samples.

EXAMPLES:

```
sage: S = spike_function([(-3,4),(-1,1),(2,3)]); S
A spike function with spikes at [-3.0, -1.0, 2.0]
sage: P = S.plot_fft_abs(8)
sage: p = P[0]; p.ydata
[5.0, 5.0, 3.367958691924177, 3.367958691924177, 4.123105625617661, 4.123105625617661, 4.759
```

plot_fft_arg (*samples=4096*, *xmin=None*, *xmax=None*, ***kwds*)
Plot of (absolute values of) Fast Fourier Transform of the spike function with given number of samples.

EXAMPLES:

```
sage: S = spike_function([(-3,4),(-1,1),(2,3)]); S
A spike function with spikes at [-3.0, -1.0, 2.0]
sage: P = S.plot_fft_arg(8)
sage: p = P[0]; p.ydata
[0.0, 0.0, -0.211524990023434..., -0.211524990023434..., 0.244978663126864..., 0.24497866312
```

vector (*samples=65536*, *xmin=None*, *xmax=None*)
Creates a sampling vector of the spike function in question.

EXAMPLES:

```
sage: S = spike_function([(-3,4),(-1,1),(2,3)],0.001); S
A spike function with spikes at [-3.0, -1.0, 2.0]
sage: S.vector(16)
(4.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

sage.functions.spike_function.**spike_function**
alias of SpikeFunction

CHAPTER

SEVEN

ORTHOGONAL POLYNOMIALS

- The Chebyshev polynomial of the first kind arises as a solution to the differential equation

$$(1-x^2)\,y'' - x\,y' + n^2\,y = 0$$

and those of the second kind as a solution to

$$(1-x^2)\,y'' - 3x\,y' + n(n+2)\,y = 0.$$

The Chebyshev polynomials of the first kind are defined by the recurrence relation

$$T_0(x) = 1\,T_1(x) = x\,T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

The Chebyshev polynomials of the second kind are defined by the recurrence relation

$$U_0(x) = 1\,U_1(x) = 2x\,U_{n+1}(x) = 2xU_n(x) - U_{n-1}(x).$$

For integers m, n, they satisfy the orthogonality relations

$$\int_{-1}^{1} T_n(x)T_m(x)\,\frac{dx}{\sqrt{1-x^2}} = \begin{cases} 0 & : n \neq m \\ \pi & : n = m = 0 \\ \pi/2 & : n = m \neq 0 \end{cases}$$

and

$$\int_{-1}^{1} U_n(x)U_m(x)\sqrt{1-x^2}\,dx = \frac{\pi}{2}\delta_{m,n}.$$

They are named after Pafnuty Chebyshev (alternative transliterations: Tchebyshef or Tschebyscheff).

- The Hermite polynomials are defined either by

$$H_n(x) = (-1)^n e^{x^2/2}\,\frac{d^n}{dx^n}e^{-x^2/2}$$

(the "probabilists' Hermite polynomials"), or by

$$H_n(x) = (-1)^n e^{x^2}\,\frac{d^n}{dx^n}e^{-x^2}$$

(the "physicists' Hermite polynomials"). Sage (via Maxima) implements the latter flavor. These satisfy the orthogonality relation

$$\int_{-\infty}^{\infty} H_n(x)H_m(x)\,e^{-x^2}\,dx = n!2^n\sqrt{\pi}\delta_{nm}$$

They are named in honor of Charles Hermite.

- Each *Legendre polynomial* $P_n(x)$ is an n-th degree polynomial. It may be expressed using Rodrigues' formula:

$$P_n(x) = (2^n n!)^{-1} \frac{d^n}{dx^n} \left[(x^2-1)^n\right].$$

These are solutions to Legendre's differential equation:

$$\frac{d}{dx}\left[(1-x^2)\frac{d}{dx}P(x)\right] + n(n+1)P(x) = 0.$$

and satisfy the orthogonality relation

$$\int_{-1}^{1} P_m(x)P_n(x)\,dx = \frac{2}{2n+1}\delta_{mn}$$

The *Legendre function of the second kind* $Q_n(x)$ is another (linearly independent) solution to the Legendre differential equation. It is not an "orthogonal polynomial" however.

The associated Legendre functions of the first kind $P_\ell^m(x)$ can be given in terms of the "usual" Legendre polynomials by

$$\begin{aligned} P_\ell^m(x) &= (-1)^m(1-x^2)^{m/2}\tfrac{d^m}{dx^m}P_\ell(x) \\ &= \tfrac{(-1)^m}{2^\ell \ell!}(1-x^2)^{m/2}\tfrac{d^{\ell+m}}{dx^{\ell+m}}(x^2-1)^\ell. \end{aligned}$$

Assuming $0 \le m \le \ell$, they satisfy the orthogonality relation:

$$\int_{-1}^{1} P_k^{(m)} P_\ell^{(m)} dx = \frac{2(\ell+m)!}{(2\ell+1)(\ell-m)!}\,\delta_{k,\ell},$$

where $\delta_{k,\ell}$ is the Kronecker delta.

The associated Legendre functions of the second kind $Q_\ell^m(x)$ can be given in terms of the "usual" Legendre polynomials by

$$Q_\ell^m(x) = (-1)^m(1-x^2)^{m/2}\frac{d^m}{dx^m}Q_\ell(x).$$

They are named after Adrien-Marie Legendre.

- Laguerre polynomials may be defined by the Rodrigues formula

$$L_n(x) = \frac{e^x}{n!}\frac{d^n}{dx^n}\left(e^{-x}x^n\right).$$

They are solutions of Laguerre's equation:

$$x\,y'' + (1-x)\,y' + n\,y = 0$$

and satisfy the orthogonality relation

$$\int_0^\infty L_m(x)L_n(x)e^{-x}\,dx = \delta_{mn}.$$

The generalized Laguerre polynomials may be defined by the Rodrigues formula:

$$L_n^{(\alpha)}(x) = \frac{x^{-\alpha}e^x}{n!}\frac{d^n}{dx^n}\left(e^{-x}x^{n+\alpha}\right).$$

(These are also sometimes called the associated Laguerre polynomials.) The simple Laguerre polynomials are recovered from the generalized polynomials by setting $\alpha = 0$.

They are named after Edmond Laguerre.

- Jacobi polynomials are a class of orthogonal polynomials. They are obtained from hypergeometric series in cases where the series is in fact finite:

$$P_n^{(\alpha,\beta)}(z) = \frac{(\alpha+1)_n}{n!}\,{}_2F_1\left(-n, 1+\alpha+\beta+n; \alpha+1; \frac{1-z}{2}\right),$$

  where $()_n$ is Pochhammer's symbol (for the rising factorial), (Abramowitz and Stegun p561.) and thus have the explicit expression

$$P_n^{(\alpha,\beta)}(z) = \frac{\Gamma(\alpha+n+1)}{n!\Gamma(\alpha+\beta+n+1)} \sum_{m=0}^{n} \binom{n}{m} \frac{\Gamma(\alpha+\beta+n+m+1)}{\Gamma(\alpha+m+1)} \left(\frac{z-1}{2}\right)^m.$$

  They are named after Carl Jacobi.

- Ultraspherical or Gegenbauer polynomials are given in terms of the Jacobi polynomials $P_n^{(\alpha,\beta)}(x)$ with $\alpha = \beta = a - 1/2$ by

$$C_n^{(a)}(x) = \frac{\Gamma(a+1/2)}{\Gamma(2a)} \frac{\Gamma(n+2a)}{\Gamma(n+a+1/2)} P_n^{(a-1/2,a-1/2)}(x).$$

  They satisfy the orthogonality relation

$$\int_{-1}^{1} (1-x^2)^{a-1/2} C_m^{(a)}(x) C_n^{(a)}(x)\,dx = \delta_{mn} 2^{1-2a}\pi \frac{\Gamma(n+2a)}{(n+a)\Gamma^2(a)\Gamma(n+1)},$$

  for $a > -1/2$. They are obtained from hypergeometric series in cases where the series is in fact finite:

$$C_n^{(a)}(z) = \frac{(2a)^{\underline{n}}}{n!}\,{}_2F_1\left(-n, 2a+n; a+\frac{1}{2}; \frac{1-z}{2}\right)$$

  where $\underline{n}$ is the falling factorial. (See Abramowitz and Stegun p561)

  They are named for Leopold Gegenbauer (1849-1903).

For completeness, the Pochhammer symbol, introduced by Leo August Pochhammer, $(x)_n$, is used in the theory of special functions to represent the "rising factorial" or "upper factorial"

$$(x)_n = x(x+1)(x+2)\cdots(x+n-1) = \frac{(x+n-1)!}{(x-1)!}.$$

On the other hand, the "falling factorial" or "lower factorial" is

$$x^{\underline{n}} = \frac{x!}{(x-n)!},$$

in the notation of Ronald L. Graham, Donald E. Knuth and Oren Patashnik in their book Concrete Mathematics.

Todo

Implement Zernike polynomials. Wikipedia article Zernike_polynomials

REFERENCES:

AUTHORS:

- David Joyner (2006-06)
- Stefan Reiterer (2010-)
- Ralf Stephan (2015-)

The original module wrapped some of the orthogonal/special functions in the Maxima package "orthopoly" and was was written by Barton Willis of the University of Nebraska at Kearney.

class sage.functions.orthogonal_polys.**ChebyshevFunction**(*name*, *nargs=2*, *latex_name=None*, *conversions={}*)

Bases: sage.functions.orthogonal_polys.OrthogonalFunction

Abstract base class for Chebyshev polynomials of the first and second kind.

EXAMPLES:

```
sage: chebyshev_T(3,x)
4*x^3 - 3*x
```

class sage.functions.orthogonal_polys.**Func_chebyshev_T**

Bases: sage.functions.orthogonal_polys.ChebyshevFunction

Chebyshev polynomials of the first kind.

REFERENCE:

- [ASHandbook] 22.5.31 page 778 and 6.1.22 page 256.

EXAMPLES:

```
sage: chebyshev_T(5,x)
16*x^5 - 20*x^3 + 5*x
sage: var('k')
k
sage: test = chebyshev_T(k,x)
sage: test
chebyshev_T(k, x)
```

eval_algebraic(*n*, *x*)

Evaluate chebyshev_T as polynomial, using a recursive formula.

INPUT:

- n – an integer
- x – a value to evaluate the polynomial at (this can be any ring element)

EXAMPLES:

```
sage: chebyshev_T.eval_algebraic(5, x)
2*(2*(2*x^2 - 1)*x - x)*(2*x^2 - 1) - x
sage: chebyshev_T(-7, x) - chebyshev_T(7,x)
0
sage: R.<t> = ZZ[]
sage: chebyshev_T.eval_algebraic(-1, t)
t
sage: chebyshev_T.eval_algebraic(0, t)
1
sage: chebyshev_T.eval_algebraic(1, t)
t
sage: chebyshev_T(7^100, 1/2)
1/2
sage: chebyshev_T(7^100, Mod(2,3))
2
sage: n = 97; x = RIF(pi/2/n)
sage: chebyshev_T(n, cos(x)).contains_zero()
True
sage: R.<t> = Zp(2, 8, 'capped-abs')[]
```

```
sage: chebyshev_T(10^6+1, t)
(2^7 + O(2^8))*t^5 + (O(2^8))*t^4 + (2^6 + O(2^8))*t^3 + (O(2^8))*t^2 + (1 + 2^6 + O(2^8))*t
```

eval_formula(*n*, *x*)

Evaluate `chebyshev_T` using an explicit formula. See [ASHandbook] 227 (p. 782) for details for the recurions. See also [EffCheby] for fast evaluation techniques.

INPUT:

- `n` – an integer
- `x` – a value to evaluate the polynomial at (this can be any ring element)

EXAMPLES:

```
sage: chebyshev_T.eval_formula(-1,x)
x
sage: chebyshev_T.eval_formula(0,x)
1
sage: chebyshev_T.eval_formula(1,x)
x
sage: chebyshev_T.eval_formula(2,0.1) == chebyshev_T._evalf_(2,0.1)
True
sage: chebyshev_T.eval_formula(10,x)
512*x^10 - 1280*x^8 + 1120*x^6 - 400*x^4 + 50*x^2 - 1
sage: chebyshev_T.eval_algebraic(10,x).expand()
512*x^10 - 1280*x^8 + 1120*x^6 - 400*x^4 + 50*x^2 - 1
```

class sage.functions.orthogonal_polys.**Func_chebyshev_U**

Bases: sage.functions.orthogonal_polys.ChebyshevFunction

Class for the Chebyshev polynomial of the second kind.

REFERENCE:

- [ASHandbook] 22.8.3 page 783 and 6.1.22 page 256.

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: chebyshev_U(2,t)
4*t^2 - 1
sage: chebyshev_U(3,t)
8*t^3 - 4*t
```

eval_algebraic(*n*, *x*)

Evaluate `chebyshev_U` as polynomial, using a recursive formula.

INPUT:

- `n` – an integer
- `x` – a value to evaluate the polynomial at (this can be any ring element)

EXAMPLES:

```
sage: chebyshev_U.eval_algebraic(5,x)
-2*((2*x + 1)*(2*x - 1)*x - 4*(2*x^2 - 1)*x)*(2*x + 1)*(2*x - 1)
sage: parent(chebyshev_U(3, Mod(8,9)))
Ring of integers modulo 9
sage: parent(chebyshev_U(3, Mod(1,9)))
Ring of integers modulo 9
sage: chebyshev_U(-3,x) + chebyshev_U(1,x)
```

```
0
sage: chebyshev_U(-1,Mod(5,8))
0
sage: parent(chebyshev_U(-1,Mod(5,8)))
Ring of integers modulo 8
sage: R.<t> = ZZ[]
sage: chebyshev_U.eval_algebraic(-2, t)
-1
sage: chebyshev_U.eval_algebraic(-1, t)
0
sage: chebyshev_U.eval_algebraic(0, t)
1
sage: chebyshev_U.eval_algebraic(1, t)
2*t
sage: n = 97; x = RIF(pi/n)
sage: chebyshev_U(n-1, cos(x)).contains_zero()
True
sage: R.<t> = Zp(2, 6, 'capped-abs')[]
sage: chebyshev_U(10^6+1, t)
(2 + O(2^6))*t + (O(2^6))
```

eval_formula(*n*, *x*)

Evaluate chebyshev_U using an explicit formula. See [ASHandbook] 227 (p. 782) for details on the recurions. See also [EffCheby] for the recursion formulas.

INPUT:

- n – an integer
- x – a value to evaluate the polynomial at (this can be any ring element)

EXAMPLES:

```
sage: chebyshev_U.eval_formula(10, x)
1024*x^10 - 2304*x^8 + 1792*x^6 - 560*x^4 + 60*x^2 - 1
sage: chebyshev_U.eval_formula(-2, x)
-1
sage: chebyshev_U.eval_formula(-1, x)
0
sage: chebyshev_U.eval_formula(0, x)
1
sage: chebyshev_U.eval_formula(1, x)
2*x
sage: chebyshev_U.eval_formula(2,0.1) == chebyshev_U._evalf_(2,0.1)
True
```

class sage.functions.orthogonal_polys.**Func_gen_laguerre**

Bases: sage.functions.orthogonal_polys.OrthogonalFunction

REFERENCE:

- [ASHandbook] 22.5.16, page 778 and page 789.

class sage.functions.orthogonal_polys.**Func_laguerre**

Bases: sage.functions.orthogonal_polys.OrthogonalFunction

REFERENCE:

- [ASHandbook] 22.5.16, page 778 and page 789.

class sage.functions.orthogonal_polys.**OrthogonalFunction**(*name*, *nargs=2*, *latex_name=None*, *conversions={}*)

Bases: sage.symbolic.function.BuiltinFunction

Base class for orthogonal polynomials.

This class is an abstract base class for all orthogonal polynomials since they share similar properties. The evaluation as a polynomial is either done via maxima, or with pynac.

Convention: The first argument is always the order of the polynomial, the others are other values or parameters where the polynomial is evaluated.

eval_formula(**args*)

Evaluate this polynomial using an explicit formula.

EXAMPLES:

```
sage: from sage.functions.orthogonal_polys import OrthogonalFunction
sage: P = OrthogonalFunction('testo_P')
sage: P.eval_formula(1,2.0)
Traceback (most recent call last):
...
NotImplementedError: no explicit calculation of values implemented
```

sage.functions.orthogonal_polys.**gegenbauer**(*n*, *a*, *x*)

Returns the ultraspherical (or Gegenbauer) polynomial for integers $n > -1$.

Computed using Maxima.

REFERENCE:

- [ASHandbook] 22.5.27

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: ultraspherical(2,3/2,x)
15/2*x^2 - 3/2
sage: ultraspherical(2,1/2,x)
3/2*x^2 - 1/2
sage: ultraspherical(1,1,x)
2*x
sage: t = PolynomialRing(RationalField(),"t").gen()
sage: gegenbauer(3,2,t)
32*t^3 - 12*t
```

Check that trac ticket #17192 is fixed:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: ultraspherical(0,1,x)
1

sage: ultraspherical(-1,1,x)
Traceback (most recent call last):
...
ValueError: n must be greater than -1, got n = -1

sage: ultraspherical(-7,1,x)
Traceback (most recent call last):
...
ValueError: n must be greater than -1, got n = -7
```

sage.functions.orthogonal_polys.**gen_legendre_P**(*n*, *m*, *x*)
Returns the generalized (or associated) Legendre function of the first kind.

The awkward code for when m is odd and 1 results from the fact that Maxima is happy with, for example, $(1-t^2)^3/2$, but Sage is not. For these cases the function is computed from the (m-1)-case using one of the recursions satisfied by the Legendre functions.

REFERENCE:

•Gradshteyn and Ryzhik 8.706 page 1000.

EXAMPLES:

```
sage: P.<t> = QQ[]
sage: gen_legendre_P(2, 0, t)
3/2*t^2 - 1/2
sage: gen_legendre_P(2, 0, t) == legendre_P(2, t)
True
sage: gen_legendre_P(3, 1, t)
-3/2*(5*t^2 - 1)*sqrt(-t^2 + 1)
sage: gen_legendre_P(4, 3, t)
105*(t^3 - t)*sqrt(-t^2 + 1)
sage: gen_legendre_P(7, 3, I).expand()
-16695*sqrt(2)
sage: gen_legendre_P(4, 1, 2.5)
-583.562373654533*I
```

sage.functions.orthogonal_polys.**gen_legendre_Q**(*n*, *m*, *x*)
Returns the generalized (or associated) Legendre function of the second kind.

Maxima restricts m = n. Hence the cases m n are computed using the same recursion used for gen_legendre_P(n,m,x) when m is odd and 1.

EXAMPLES:

```
sage: P.<t> = QQ[]
sage: gen_legendre_Q(2,0,t)
3/4*t^2*log(-(t + 1)/(t - 1)) - 3/2*t - 1/4*log(-(t + 1)/(t - 1))
sage: gen_legendre_Q(2,0,t) - legendre_Q(2, t)
0
sage: gen_legendre_Q(3,1,0.5)
2.49185259170895
sage: gen_legendre_Q(0, 1, x)
-1/sqrt(-x^2 + 1)
sage: gen_legendre_Q(2, 4, x).factor()
48*x/((x + 1)^2*(x - 1)^2)
```

sage.functions.orthogonal_polys.**hermite**(*n*, *x*)
Returns the Hermite polynomial for integers $n > -1$.

REFERENCE:

•[ASHandbook] 22.5.40 and 22.5.41, page 779.

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: hermite(2,x)
4*x^2 - 2
sage: hermite(3,x)
8*x^3 - 12*x
sage: hermite(3,2)
40
```

```
sage: S.<y> = PolynomialRing(RR)
sage: hermite(3,y)
8.00000000000000*y^3 - 12.0000000000000*y
sage: R.<x,y> = QQ[]
sage: hermite(3,y^2)
8*y^6 - 12*y^2
sage: w = var('w')
sage: hermite(3,2*w)
8*(8*w^2 - 3)*w
```

Check that trac ticket #17192 is fixed:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: hermite(0,x)
1

sage: hermite(-1,x)
Traceback (most recent call last):
...
ValueError: n must be greater than -1, got n = -1

sage: hermite(-7,x)
Traceback (most recent call last):
...
ValueError: n must be greater than -1, got n = -7
```

sage.functions.orthogonal_polys.**jacobi_P**(*n*, *a*, *b*, *x*)

Returns the Jacobi polynomial $P_n^{(a,b)}(x)$ for integers $n > -1$ and a and b symbolic or $a > -1$ and $b > -1$. The Jacobi polynomials are actually defined for all a and b. However, the Jacobi polynomial weight $(1-x)^a(1+x)^b$ isn't integrable for $a \leq -1$ or $b \leq -1$.

REFERENCE:

- Table on page 789 in [ASHandbook].

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: jacobi_P(2,0,0,x)
3/2*x^2 - 1/2
sage: jacobi_P(2,1,2,1.2)        # random output of low order bits
5.009999999999998
```

Check that trac ticket #17192 is fixed:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: jacobi_P(0,0,0,x)
1

sage: jacobi_P(-1,0,0,x)
Traceback (most recent call last):
...
ValueError: n must be greater than -1, got n = -1

sage: jacobi_P(-7,0,0,x)
Traceback (most recent call last):
...
ValueError: n must be greater than -1, got n = -7
```

sage.functions.orthogonal_polys.**legendre_P**(*n*, *x*)

Returns the Legendre polynomial of the first kind.

REFERENCE:

- [ASHandbook] 22.5.35 page 779.

EXAMPLES:

```
sage: P.<t> = QQ[]
sage: legendre_P(2,t)
3/2*t^2 - 1/2
sage: legendre_P(3, 1.1)
1.67750000000000
sage: legendre_P(3, 1 + I)
7/2*I - 13/2
sage: legendre_P(3, MatrixSpace(ZZ, 2)([1, 2, -4, 7]))
[-179  242]
[-484  547]
sage: legendre_P(3, GF(11)(5))
8
```

sage.functions.orthogonal_polys.**legendre_Q**(n, x)

Returns the Legendre function of the second kind.

Computed using Maxima.

EXAMPLES:

```
sage: P.<t> = QQ[]
sage: legendre_Q(2, t)
3/4*t^2*log(-(t + 1)/(t - 1)) - 3/2*t - 1/4*log(-(t + 1)/(t - 1))
sage: legendre_Q(3, 0.5)
-0.198654771479482
sage: legendre_Q(4, 2)
443/16*I*pi + 443/16*log(3) - 365/12
sage: legendre_Q(4, 2.0)
0.00116107583162324 + 86.9828465962674*I
```

sage.functions.orthogonal_polys.**ultraspherical**(n, a, x)

Returns the ultraspherical (or Gegenbauer) polynomial for integers $n > -1$.

Computed using Maxima.

REFERENCE:

- [ASHandbook] 22.5.27

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: ultraspherical(2,3/2,x)
15/2*x^2 - 3/2
sage: ultraspherical(2,1/2,x)
3/2*x^2 - 1/2
sage: ultraspherical(1,1,x)
2*x
sage: t = PolynomialRing(RationalField(),"t").gen()
sage: gegenbauer(3,2,t)
32*t^3 - 12*t
```

Check that trac ticket #17192 is fixed:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: ultraspherical(0,1,x)
```

```
1

sage: ultraspherical(-1,1,x)
Traceback (most recent call last):
...
ValueError: n must be greater than -1, got n = -1

sage: ultraspherical(-7,1,x)
Traceback (most recent call last):
...
ValueError: n must be greater than -1, got n = -7
```

CHAPTER

EIGHT

OTHER FUNCTIONS

class sage.functions.other.**Function_abs**

Bases: sage.symbolic.function.GinacFunction

The absolute value function.

EXAMPLES:

```
sage: var('x y')
(x, y)
sage: abs(x)
abs(x)
sage: abs(x^2 + y^2)
abs(x^2 + y^2)
sage: abs(-2)
2
sage: sqrt(x^2)
sqrt(x^2)
sage: abs(sqrt(x))
sqrt(abs(x))
sage: complex(abs(3*I))
(3+0j)

sage: f = sage.functions.other.Function_abs()
sage: latex(f)
\mathrm{abs}
sage: latex(abs(x))
{\left| x \right|}
sage: abs(x)._sympy_()
Abs(x)
```

Test pickling:

```
sage: loads(dumps(abs(x)))
abs(x)
```

TESTS:

Check that trac ticket #12588 is fixed:

```
sage: abs(pi*I)
pi
sage: abs(pi*I*catalan)
catalan*pi
sage: abs(pi*catalan*x)
catalan*pi*abs(x)
sage: abs(pi*I*catalan*x)
catalan*pi*abs(x)
```

```
sage: abs(1.0j*pi)
1.00000000000000*pi
sage: abs(I*x)
abs(x)
sage: abs(I*pi)
pi
sage: abs(I*log(2))
log(2)
sage: abs(I*e^5)
e^5
sage: abs(log(1/2))
-log(1/2)
sage: abs(log(3/2))
log(3/2)
sage: abs(log(1/2)*log(1/3))
log(1/2)*log(1/3)
sage: abs(log(1/2)*log(1/3)*log(1/4))
-log(1/2)*log(1/3)*log(1/4)
sage: abs(log(1/2)*log(1/3)*log(1/4)*i)
-log(1/2)*log(1/3)*log(1/4)
sage: abs(log(x))
abs(log(x))
sage: abs(zeta(I))
abs(zeta(I))
sage: abs(e^2*x)
abs(x)*e^2
sage: abs((pi+e)*x)
(pi + e)*abs(x)
```

class sage.functions.other.**Function_arg**

Bases: sage.symbolic.function.BuiltinFunction

The argument function for complex numbers.

EXAMPLES:

```
sage: arg(3+i)
arctan(1/3)
sage: arg(-1+i)
3/4*pi
sage: arg(2+2*i)
1/4*pi
sage: arg(2+x)
arg(x + 2)
sage: arg(2.0+i+x)
arg(x + 2.00000000000000 + 1.00000000000000*I)
sage: arg(-3)
pi
sage: arg(3)
0
sage: arg(0)
0
sage: latex(arg(x))
{\rm arg}\left(x\right)
sage: maxima(arg(x))
atan2(0,_SAGE_VAR_x)
sage: maxima(arg(2+i))
atan(1/2)
sage: maxima(arg(sqrt(2)+i))
```

```
atan(1/sqrt(2))
sage: arg(2+i)
arctan(1/2)
sage: arg(sqrt(2)+i)
arg(sqrt(2) + I)
sage: arg(sqrt(2)+i).simplify()
arctan(1/2*sqrt(2))
```

TESTS:

```
sage: arg(0.0)
0.000000000000000
sage: arg(3.0)
0.000000000000000
sage: arg(-2.5)
3.14159265358979
sage: arg(2.0+3*i)
0.982793723247329
```

class sage.functions.other.**Function_beta**

Bases: sage.symbolic.function.GinacFunction

Return the beta function. This is defined by

$$B(p,q) = \int_0^1 t^{p-1}(1-t)^{1-q}dt$$

for complex or symbolic input p and q. Note that the order of inputs does not matter: $B(p,q) = B(q,p)$.

GiNaC is used to compute $B(p,q)$. However, complex inputs are not yet handled in general. When GiNaC raises an error on such inputs, we raise a NotImplementedError.

If either input is 1, GiNaC returns the reciprocal of the other. In other cases, GiNaC uses one of the following formulas:

$$B(p,q) = \Gamma(p)\Gamma(q)/\Gamma(p+q)$$

or

$$B(p,q) = (-1)^q B(1-p-q,q).$$

For numerical inputs, GiNaC uses the formula

$$B(p,q) = \exp[\log\Gamma(p) + \log\Gamma(q) - \log\Gamma(p+q)]$$

INPUT:

- p - number or symbolic expression
- q - number or symbolic expression

OUTPUT: number or symbolic expression (if input is symbolic)

EXAMPLES:

```
sage: beta(3,2)
1/12
sage: beta(3,1)
1/3
sage: beta(1/2,1/2)
beta(1/2, 1/2)
```

```
sage: beta(-1,1)
-1
sage: beta(-1/2,-1/2)
0
sage: beta(x/2,3)
beta(3, 1/2*x)
sage: beta(.5,.5)
3.14159265358979
sage: beta(1,2.0+I)
0.400000000000000 - 0.200000000000000*I
sage: beta(3,x+I)
beta(3, x + I)
```

Note that the order of arguments does not matter:

```
sage: beta(1/2,3*x)
beta(1/2, 3*x)
```

The result is symbolic if exact input is given:

```
sage: beta(2,1+5*I)
beta(2, 5*I + 1)
sage: beta(2, 2.)
0.166666666666667
sage: beta(I, 2.)
-0.500000000000000 - 0.500000000000000*I
sage: beta(2., 2)
0.166666666666667
sage: beta(2., I)
-0.500000000000000 - 0.500000000000000*I
```

Test pickling:

```
sage: loads(dumps(beta))
beta
```

class sage.functions.other.**Function_binomial**

Bases: sage.symbolic.function.GinacFunction

Return the binomial coefficient

$$\binom{x}{m} = x(x-1)\cdots(x-m+1)/m!$$

which is defined for $m \in \mathbf{Z}$ and any x. We extend this definition to include cases when $x - m$ is an integer but m is not by

$$\binom{x}{m} = \binom{x}{x-m}$$

If $m < 0$, return 0.

INPUT:

- x, m - numbers or symbolic expressions. Either m or x-m must be an integer, else the output is symbolic.

OUTPUT: number or symbolic expression (if input is symbolic)

EXAMPLES:

```
sage: binomial(5,2)
10
sage: binomial(2,0)
1
sage: binomial(1/2, 0)
1
sage: binomial(3,-1)
0
sage: binomial(20,10)
184756
sage: binomial(-2, 5)
-6
sage: binomial(RealField()('2.5'), 2)
1.87500000000000
sage: n=var('n'); binomial(n,2)
1/2*(n - 1)*n
sage: n=var('n'); binomial(n,n)
1
sage: n=var('n'); binomial(n,n-1)
n
sage: binomial(2^100, 2^100)
1

sage: k, i = var('k,i')
sage: binomial(k,i)
binomial(k, i)
```

We can use a `hold` parameter to prevent automatic evaluation:

```
sage: SR(5).binomial(3, hold=True)
binomial(5, 3)
sage: SR(5).binomial(3, hold=True).simplify()
10
```

TESTS: We verify that we can convert this function to Maxima and bring it back into Sage.

```
sage: n,k = var('n,k')
sage: maxima(binomial(n,k))
binomial(_SAGE_VAR_n,_SAGE_VAR_k)
sage: _.sage()
binomial(n, k)
sage: binomial._maxima_init_()
'binomial'
```

Test pickling:

```
sage: loads(dumps(binomial(n,k)))
binomial(n, k)
```

class `sage.functions.other.`**`Function_ceil`**

Bases: `sage.symbolic.function.BuiltinFunction`

The ceiling function.

The ceiling of x is computed in the following manner.

1. The `x.ceil()` method is called and returned if it is there. If it is not, then Sage checks if x is one of Python's native numeric data types. If so, then it calls and returns `Integer(int(math.ceil(x)))`.
2. Sage tries to convert x into a `RealIntervalField` with 53 bits of precision. Next, the ceilings of the

endpoints are computed. If they are the same, then that value is returned. Otherwise, the precision of the `RealIntervalField` is increased until they do match up or it reaches `maximum_bits` of precision.

3.If none of the above work, Sage returns a `Expression` object.

EXAMPLES:

```
sage: a = ceil(2/5 + x)
sage: a
ceil(x + 2/5)
sage: a(x=4)
5
sage: a(x=4.0)
5
sage: ZZ(a(x=3))
4
sage: a = ceil(x^3 + x + 5/2); a
ceil(x^3 + x + 5/2)
sage: a.simplify()
ceil(x^3 + x + 1/2) + 2
sage: a(x=2)
13

sage: ceil(sin(8)/sin(2))
2

sage: ceil(5.4)
6
sage: type(ceil(5.4))
<type 'sage.rings.integer.Integer'>

sage: ceil(factorial(50)/exp(1))
11188719610782480504630258070757734324011354208865721592720336801
sage: ceil(SR(10^50 + 10^(-50)))
10000000000000000000000000000000000000000000000001
sage: ceil(SR(10^50 - 10^(-50)))
10000000000000000000000000000000000000000000000000

sage: ceil(sec(e))
-1

sage: latex(ceil(x))
\left \lceil x \right \rceil
sage: ceil(x)._sympy_()
ceiling(x)

sage: import numpy
sage: a = numpy.linspace(0,2,6)
sage: ceil(a)
array([ 0.,  1.,  1.,  2.,  2.,  2.])
```

Test pickling:

```
sage: loads(dumps(ceil))
ceil
```

class `sage.functions.other.`**Function_conjugate**

Bases: `sage.symbolic.function.GinacFunction`

Returns the complex conjugate of the input.

It is possible to prevent automatic evaluation using the `hold` parameter:

```
sage: conjugate(I,hold=True)
conjugate(I)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: conjugate(I,hold=True).simplify()
-I
```

TESTS:

```
sage: x,y = var('x,y')
sage: x.conjugate()
conjugate(x)
sage: latex(conjugate(x))
\overline{x}
sage: f = function('f')
sage: latex(f(x).conjugate())
\overline{f\left(x\right)}
sage: f = function('psi',x,y)
sage: latex(f.conjugate())
\overline{\psi\left(x, y\right)}
sage: x.conjugate().conjugate()
x
sage: x.conjugate().operator()
conjugate
sage: x.conjugate().operator() == conjugate
True
```

Check if trac ticket #8755 is fixed:

```
sage: conjugate(sqrt(-3))
conjugate(sqrt(-3))
sage: conjugate(sqrt(3))
sqrt(3)
sage: conjugate(sqrt(x))
conjugate(sqrt(x))
sage: conjugate(x^2)
conjugate(x)^2
sage: var('y',domain='positive')
y
sage: conjugate(sqrt(y))
sqrt(y)
```

Check if trac ticket #10964 is fixed:

```
sage: z= I*sqrt(-3); z
I*sqrt(-3)
sage: conjugate(z)
-I*conjugate(sqrt(-3))
sage: var('a')
a
sage: conjugate(a*sqrt(-2)*sqrt(-3))
conjugate(sqrt(-2))*conjugate(sqrt(-3))*conjugate(a)
```

Test pickling:

```
sage: loads(dumps(conjugate))
conjugate
```

class sage.functions.other.**Function_erf**

Bases: sage.symbolic.function.BuiltinFunction

The error function, defined for real values as

$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$

This function is also defined for complex values, via analytic continuation.

EXAMPLES:

We can evaluate numerically:

```
sage: erf(2)
erf(2)
sage: erf(2).n()
0.995322265018953
sage: erf(2).n(100)
0.99532226501895273416206925637
sage: erf(ComplexField(100)(2+3j))
-20.829461427614568389103088452 + 8.6873182714701631444280787545*I
```

Basic symbolic properties are handled by Sage and Maxima:

```
sage: x = var("x")
sage: diff(erf(x),x)
2*e^(-x^2)/sqrt(pi)
sage: integrate(erf(x),x)
x*erf(x) + e^(-x^2)/sqrt(pi)
```

ALGORITHM:

Sage implements numerical evaluation of the error function via the `erf()` function from mpmath. Symbolics are handled by Sage and Maxima.

REFERENCES:

- http://en.wikipedia.org/wiki/Error_function
- http://mpmath.googlecode.com/svn/trunk/doc/build/functions/expintegrals.html#error-functions

TESTS:

Check limits:

```
sage: limit(erf(x),x=0)
0
sage: limit(erf(x),x=infinity)
1
```

```
Check that it's odd::

    sage: erf(1.0)
    0.842700792949715
    sage: erf(-1.0)
    -0.842700792949715
```

Check against other implementations and against the definition:

```
sage: erf(3).n()
0.999977909503001
sage: maxima.erf(3).n()
0.999977909503001
sage: (1-pari(3).erfc())
```

```
0.999977909503001
sage: RR(3).erf()
0.999977909503001
sage: (integrate(exp(-x**2),(x,0,3))*2/sqrt(pi)).n()
0.999977909503001
```

trac ticket #9044:

```
sage: N(erf(sqrt(2)),200)
0.95449973610364158559943472566693312505644755259664313203267
```

trac ticket #11626:

```
sage: n(erf(2),100)
0.99532226501895273416206925637
sage: erf(2).n(100)
0.99532226501895273416206925637
```

Test (indirectly) trac ticket #11885:

```
sage: erf(float(0.5))
0.5204998778130465
sage: erf(complex(0.5))
(0.5204998778130465+0j)
```

Ensure conversion from maxima elements works:

```
sage: merf = maxima(erf(x)).sage().operator()
sage: merf == erf
True
```

Make sure we can dump and load it:

```
sage: loads(dumps(erf(2)))
erf(2)
```

Special-case 0 for immediate evaluation:

```
sage: erf(0)
0
sage: solve(erf(x)==0,x)
[x == 0]
```

Make sure that we can hold:

```
sage: erf(0,hold=True)
erf(0)
sage: simplify(erf(0,hold=True))
0
```

Check that high-precision ComplexField inputs work:

```
sage: CC(erf(ComplexField(1000)(2+3j)))
-20.8294614276146 + 8.68731827147016*I
```

class sage.functions.other.**Function_factorial**

Bases: sage.symbolic.function.GinacFunction

Returns the factorial of n.

INPUT:

- n - any complex argument (except negative integers) or any symbolic expression

OUTPUT: an integer or symbolic expression

EXAMPLES:

```
sage: x = var('x')
sage: factorial(0)
1
sage: factorial(4)
24
sage: factorial(10)
3628800
sage: factorial(6) == 6*5*4*3*2
True
sage: f = factorial(x + factorial(x)); f
factorial(x + factorial(x))
sage: f(x=3)
362880
sage: factorial(x)^2
factorial(x)^2
```

To prevent automatic evaluation use the `hold` argument:

```
sage: factorial(5,hold=True)
factorial(5)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: factorial(5,hold=True).simplify()
120
```

We can also give input other than nonnegative integers. For other nonnegative numbers, the `gamma()` function is used:

```
sage: factorial(1/2)
1/2*sqrt(pi)
sage: factorial(3/4)
gamma(7/4)
sage: factorial(2.3)
2.68343738195577
```

But negative input always fails:

```
sage: factorial(-32)
Traceback (most recent call last):
...
ValueError: factorial -- self = (-32) must be nonnegative
```

TESTS:

We verify that we can convert this function to Maxima and bring it back into Sage.:

```
sage: z = var('z')
sage: factorial._maxima_init_()
'factorial'
sage: maxima(factorial(z))
factorial(_SAGE_VAR_z)
sage: _.sage()
factorial(z)
sage: k = var('k')
sage: factorial(k)
```

```
factorial(k)

sage: factorial(3.14)
7.173269190187...
```

Test latex typesetting:

```
sage: latex(factorial(x))
x!
sage: latex(factorial(2*x))
\left(2 \, x\right)!
sage: latex(factorial(sin(x)))
\sin\left(x\right)!
sage: latex(factorial(sqrt(x+1)))
\left(\sqrt{x + 1}\right)!
sage: latex(factorial(sqrt(x)))
\sqrt{x}!
sage: latex(factorial(x^(2/3)))
\left(x^{\frac{2}{3}}\right)!

sage: latex(factorial)
{\rm factorial}
```

Check that #11539 is fixed:

```
sage: (factorial(x) == 0).simplify()
factorial(x) == 0
sage: maxima(factorial(x) == 0).sage()
factorial(x) == 0
sage: y = var('y')
sage: (factorial(x) == y).solve(x)
[factorial(x) == y]
```

Test pickling:

```
sage: loads(dumps(factorial))
factorial
```

class sage.functions.other.**Function_floor**

Bases: sage.symbolic.function.BuiltinFunction

The floor function.

The floor of x is computed in the following manner.

1. The `x.floor()` method is called and returned if it is there. If it is not, then Sage checks if x is one of Python's native numeric data types. If so, then it calls and returns `Integer(int(math.floor(x)))`.
2. Sage tries to convert x into a `RealIntervalField` with 53 bits of precision. Next, the floors of the endpoints are computed. If they are the same, then that value is returned. Otherwise, the precision of the `RealIntervalField` is increased until they do match up or it reaches `maximum_bits` of precision.
3. If none of the above work, Sage returns a symbolic `Expression` object.

EXAMPLES:

```
sage: floor(5.4)
5
sage: type(floor(5.4))
<type 'sage.rings.integer.Integer'>
sage: var('x')
```

```
x
sage: a = floor(5.4 + x); a
floor(x + 5.40000000000000)
sage: a.simplify()
floor(x + 0.4000000000000004) + 5
sage: a(x=2)
7

sage: floor(cos(8)/cos(2))
0

sage: floor(factorial(50)/exp(1))
11188719610782480504630258070757734324011354208865721592720336800
sage: floor(SR(10^50 + 10^(-50)))
10000000000000000000000000000000000000000000000000
sage: floor(SR(10^50 - 10^(-50)))
99999999999999999999999999999999999999999999999999
sage: floor(int(10^50))
10000000000000000000000000000000000000000000000000

sage: import numpy
sage: a = numpy.linspace(0,2,6)
sage: floor(a)
array([ 0.,  0.,  0.,  1.,  1.,  2.])
```

Test pickling:

```
sage: loads(dumps(floor))
floor
```

class sage.functions.other.**Function_gamma**

Bases: sage.symbolic.function.GinacFunction

The Gamma function. This is defined by

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$$

for complex input z with real part greater than zero, and by analytic continuation on the rest of the complex plane (except for negative integers, which are poles).

It is computed by various libraries within Sage, depending on the input type.

EXAMPLES:

```
sage: from sage.functions.other import gamma1
sage: gamma1(CDF(0.5,14))
-4.0537030780372815e-10 - 5.773299834553605e-10*I
sage: gamma1(CDF(I))
-0.15494982830181067 - 0.49801566811835607*I
```

Recall that $\Gamma(n)$ is $n-1$ factorial:

```
sage: gamma1(11) == factorial(10)
True
sage: gamma1(6)
120
sage: gamma1(1/2)
sqrt(pi)
sage: gamma1(-1)
Infinity
```

```
sage: gamma1(I)
gamma(I)
sage: gamma1(x/2)(x=5)
3/4*sqrt(pi)

sage: gamma1(float(6))  # For ARM: rel tol 3e-16
120.0
sage: gamma(6.)
120.000000000000
sage: gamma1(x)
gamma(x)

sage: gamma1(pi)
gamma(pi)
sage: gamma1(i)
gamma(I)
sage: gamma1(i).n()
-0.154949828301811 - 0.498015668118356*I
sage: gamma1(int(5))
24

sage: conjugate(gamma(x))
gamma(conjugate(x))

sage: plot(gamma1(x),(x,1,5))
Graphics object consisting of 1 graphics primitive
```

To prevent automatic evaluation use the `hold` argument:

```
sage: gamma1(1/2,hold=True)
gamma(1/2)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: gamma1(1/2,hold=True).simplify()
sqrt(pi)
```

TESTS:

We verify that we can convert this function to Maxima and convert back to Sage:

```
sage: z = var('z')
sage: maxima(gamma1(z)).sage()
gamma(z)
sage: latex(gamma1(z))
\Gamma\left(z\right)
```

Test that Trac ticket 5556 is fixed:

```
sage: gamma1(3/4)
gamma(3/4)

sage: gamma1(3/4).n(100)
1.2254167024651776451290983034
```

Check that negative integer input works:

```
sage: (-1).gamma()
Infinity
sage: (-1.).gamma()
```

```
NaN
sage: CC(-1).gamma()
Infinity
sage: RDF(-1).gamma()
NaN
sage: CDF(-1).gamma()
Infinity
```

Check if trac ticket #8297 is fixed:

```
sage: latex(gamma(1/4))
\Gamma\left(\frac{1}{4}\right)
```

Test pickling:

```
sage: loads(dumps(gamma(x)))
gamma(x)
```

class sage.functions.other.**Function_gamma_inc**

Bases: sage.symbolic.function.BuiltinFunction

The incomplete gamma function.

EXAMPLES:

```
sage: gamma_inc(CDF(0,1), 3)
0.003208574993369116 + 0.012406185811871568*I
sage: gamma_inc(RDF(1), 3)
0.049787068367863944
sage: gamma_inc(3,2)
gamma(3, 2)
sage: gamma_inc(x,0)
gamma(x)
sage: latex(gamma_inc(3,2))
\Gamma\left(3, 2\right)
sage: loads(dumps((gamma_inc(3,2))))
gamma(3, 2)
sage: i = ComplexField(30).0; gamma_inc(2, 1 + i)
0.70709210 - 0.42035364*I
sage: gamma_inc(2., 5)
0.0404276819945128
```

class sage.functions.other.**Function_imag_part**

Bases: sage.symbolic.function.GinacFunction

Returns the imaginary part of the (possibly complex) input.

It is possible to prevent automatic evaluation using the hold parameter:

```
sage: imag_part(I,hold=True)
imag_part(I)
```

To then evaluate again, we currently must use Maxima via sage.symbolic.expression.Expression.simplify():

```
sage: imag_part(I,hold=True).simplify()
1
```

TESTS:

```
sage: z = 1+2*I
sage: imaginary(z)
2
```

```
sage: imag(z)
2
sage: imag(complex(3, 4))
4.0
sage: loads(dumps(imag_part))
imag_part
sage: imag_part(x)._sympy_()
im(x)
```

Check if trac ticket #6401 is fixed:

```
sage: latex(x.imag())
\Im \left( x \right)

sage: f(x) = function('f',x)
sage: latex( f(x).imag())
\Im \left( f\left(x\right) \right)
```

class sage.functions.other.**Function_log_gamma**

Bases: sage.symbolic.function.GinacFunction

The principal branch of the logarithm of Gamma function. Gamma is defined for complex input z with real part greater than zero, and by analytic continuation on the rest of the complex plane (except for negative integers, which are poles).

It is computed by the log_gamma function for the number type, or by $lgamma$ in Ginac, failing that.

EXAMPLES:

Numerical evaluation happens when appropriate, to the appropriate accuracy (see #10072):

```
sage: log_gamma(6)
log(120)
sage: log_gamma(6.)
4.78749174278205
sage: log_gamma(6).n()
4.78749174278205
sage: log_gamma(RealField(100)(6))
4.7874917427820459942477009345
sage: log_gamma(2.4+i)
-0.0308566579348816 + 0.693427705955790*I
sage: log_gamma(-3.1)
0.400311696703985
```

Symbolic input works (see #10075):

```
sage: log_gamma(3*x)
log_gamma(3*x)
sage: log_gamma(3+i)
log_gamma(I + 3)
sage: log_gamma(3+i+x)
log_gamma(x + I + 3)
```

To get evaluation of input for which gamma is negative and the ceiling is even, we must explicitly make the input complex. This is a known issue, see #12521:

```
sage: log_gamma(-2.1)
NaN
sage: log_gamma(CC(-2.1))
1.53171380819509 + 3.14159265358979*I
```

In order to prevent evaluation, use the *hold* argument; to evaluate a held expression, use the $n()$ numerical evaluation method:

```
sage: log_gamma(SR(5),hold=True)
log_gamma(5)
sage: log_gamma(SR(5),hold=True).n()
3.17805383034795
```

TESTS:

```
sage: log_gamma(-2.1+i)
-1.90373724496982 - 0.901638463592247*I
sage: log_gamma(pari(6))
4.78749174278205
sage: log_gamma(CC(6))
4.78749174278205
sage: log_gamma(CC(-2.5))
-0.0562437164976740 + 3.14159265358979*I
sage: log_gamma(x)._sympy_()
loggamma(x)
```

`conjugate(log_gamma(x))==log_gamma(conjugate(x))` unless on the branch cut, which runs along the negative real axis.:

```
sage: conjugate(log_gamma(x))
conjugate(log_gamma(x))
sage: var('y', domain='positive')
y
sage: conjugate(log_gamma(y))
log_gamma(y)
sage: conjugate(log_gamma(y+I))
conjugate(log_gamma(y + I))
sage: log_gamma(-2)
+Infinity
sage: conjugate(log_gamma(-2))
+Infinity
```

class `sage.functions.other.`**Function_psi1**

Bases: `sage.symbolic.function.GinacFunction`

The digamma function, $\psi(x)$, is the logarithmic derivative of the gamma function.

$$\psi(x) = \frac{d}{dx}\log(\Gamma(x)) = \frac{\Gamma'(x)}{\Gamma(x)}$$

EXAMPLES:

```
sage: from sage.functions.other import psi1
sage: psi1(x)
psi(x)
sage: psi1(x).derivative(x)
psi(1, x)

sage: psi1(3)
-euler_gamma + 3/2

sage: psi(.5)
-1.96351002602142
sage: psi(RealField(100)(.5))
-1.9635100260214234794409763330
```

TESTS:

```
sage: latex(psi1(x))
\psi\left(x\right)
sage: loads(dumps(psi1(x)+1))
psi(x) + 1

sage: t = psi1(x); t
psi(x)
sage: t.subs(x=.2)
-5.28903989659219
sage: psi(x)._sympy_()
polygamma(0, x)
```

class sage.functions.other.**Function_psi2**

Bases: sage.symbolic.function.GinacFunction

Derivatives of the digamma function $\psi(x)$. T

EXAMPLES:

```
sage: from sage.functions.other import psi2
sage: psi2(2, x)
psi(2, x)
sage: psi2(2, x).derivative(x)
psi(3, x)
sage: n = var('n')
sage: psi2(n, x).derivative(x)
psi(n + 1, x)

sage: psi2(0, x)
psi(x)
sage: psi2(-1, x)
log(gamma(x))
sage: psi2(3, 1)
1/15*pi^4

sage: psi2(2, .5).n()
-16.8287966442343
sage: psi2(2, .5).n(100)
-16.828796644234319995596334261
```

TESTS:

```
sage: psi2(n, x).derivative(n)
Traceback (most recent call last):
...
RuntimeError: cannot diff psi(n,x) with respect to n

sage: latex(psi2(2,x))
\psi\left(2, x\right)
sage: loads(dumps(psi2(2,x)+1))
psi(2, x) + 1
sage: psi(2, x)._sympy_()
polygamma(2, x)
```

class sage.functions.other.**Function_real_part**

Bases: sage.symbolic.function.GinacFunction

Returns the real part of the (possibly complex) input.

It is possible to prevent automatic evaluation using the `hold` parameter:

```
sage: real_part(I,hold=True)
real_part(I)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: real_part(I,hold=True).simplify()
0
```

EXAMPLES:

```
sage: z = 1+2*I
sage: real(z)
1
sage: real(5/3)
5/3
sage: a = 2.5
sage: real(a)
2.50000000000000
sage: type(real(a))
<type 'sage.rings.real_mpfr.RealLiteral'>
sage: real(1.0r)
1.0
sage: real(complex(3, 4))
3.0
```

TESTS:

```
sage: loads(dumps(real_part))
real_part
sage: real_part(x)._sympy_()
re(x)
```

Check if trac ticket #6401 is fixed:

```
sage: latex(x.real())
\Re \left( x \right)

sage: f(x) = function('f',x)
sage: latex( f(x).real())
\Re \left( f\left(x\right) \right)
```

class `sage.functions.other.`**Function_sqrt**

Bases: `object`

`sage.functions.other.`**gamma**(*a*, **args*, ***kwds*)

Gamma and incomplete gamma functions. This is defined by the integral

$$\Gamma(a,z) = \int_z^\infty t^{a-1}e^{-t}dt$$

EXAMPLES:

Recall that `\Gamma(n)` is `n-1` factorial::

```
sage: gamma(11) == factorial(10)
True
sage: gamma(6)
120
sage: gamma(1/2)
sqrt(pi)
```

```
    sage: gamma(-4/3)
    gamma(-4/3)
    sage: gamma(-1)
    Infinity
    sage: gamma(0)
    Infinity

::

    sage: gamma_inc(3,2)
    gamma(3, 2)
    sage: gamma_inc(x,0)
    gamma(x)

::

    sage: gamma(5, hold=True)
    gamma(5)
    sage: gamma(x, 0, hold=True)
    gamma(x, 0)

::

    sage: gamma(CDF(0.5,14))
    -4.0537030780372815e-10 - 5.773299834553605e-10*I
    sage: gamma(CDF(I))
    -0.15494982830181067 - 0.49801566811835607*I

The precision for the result is deduced from the precision of the
input. Convert the input to a higher precision explicitly if a result
with higher precision is desired.::

    sage: t = gamma(RealField(100)(2.5)); t
    1.3293403881791370204736256125
    sage: t.prec()
    100

    sage: gamma(6)
    120

    sage: gamma(pi).n(100)
    2.2880377953400324179595889091

    sage: gamma(3/4).n(100)
    1.2254167024651776451290983034

The gamma function only works with input that can be coerced to the
Symbolic Ring::

    sage: Q.<i> = NumberField(x^2+1)
    sage: gamma(i)
    Traceback (most recent call last):
    ...
    TypeError: cannot coerce arguments: no canonical coercion...

We make an exception for elements of AA or QQbar, which cannot be
coerced into symbolic expressions to allow this usage::
```

```
    sage: t = QQbar(sqrt(2)) + sqrt(3); t
    3.146264369941973?
    sage: t.parent()
    Algebraic Field
```

Symbolic functions convert the arguments to symbolic expressions if they are in QQbar or AA::

```
    sage: gamma(QQbar(I))
    -0.154949828301811 - 0.498015668118356*I
```

sage.functions.other.**psi**(*x*, **args*, ***kwds*)

The digamma function, $\psi(x)$, is the logarithmic derivative of the gamma function.

$$\psi(x) = \frac{d}{dx}\log(\Gamma(x)) = \frac{\Gamma'(x)}{\Gamma(x)}$$

We represent the n-th derivative of the digamma function with $\psi(n, x)$ or $psi(n, x)$.

EXAMPLES:

```
sage: psi(x)
psi(x)
sage: psi(.5)
-1.96351002602142
sage: psi(3)
-euler_gamma + 3/2
sage: psi(1, 5)
1/6*pi^2 - 205/144
sage: psi(1, x)
psi(1, x)
sage: psi(1, x).derivative(x)
psi(2, x)

sage: psi(3, hold=True)
psi(3)
sage: psi(1, 5, hold=True)
psi(1, 5)
```

TESTS:

```
sage: psi(2, x, 3)
Traceback (most recent call last):
...
TypeError: Symbolic function psi takes at most 2 arguments (3 given)
```

sage.functions.other.**sqrt**(*x*, **args*, ***kwds*)

INPUT:

- `x` - a number
- `prec` - integer (default: None): if None, returns an exact square root; otherwise returns a numerical square root if necessary, to the given bits of precision.
- `extend` - bool (default: True); this is a place holder, and is always ignored or passed to the sqrt function for x, since in the symbolic ring everything has a square root.
- `all` - bool (default: False); if True, return all square roots of self, instead of just one.

EXAMPLES:

```
sage: sqrt(-1)
I
sage: sqrt(2)
sqrt(2)
sage: sqrt(2)^2
2
sage: sqrt(4)
2
sage: sqrt(4,all=True)
[2, -2]
sage: sqrt(x^2)
sqrt(x^2)
```

For a non-symbolic square root, there are a few options. The best is to numerically approximate afterward:

```
sage: sqrt(2).n()
1.41421356237310
sage: sqrt(2).n(prec=100)
1.4142135623730950488016887242
```

Or one can input a numerical type.

sage: sqrt(2.) 1.41421356237310 sage: sqrt(2.0000000000000000000000) 1.4142135623730950488016887242 sage: sqrt(4.0) 2.00000000000000

To prevent automatic evaluation, one can use the `hold` parameter after coercing to the symbolic ring:

```
sage: sqrt(SR(4),hold=True)
sqrt(4)
sage: sqrt(4,hold=True)
Traceback (most recent call last):
...
TypeError: _do_sqrt() got an unexpected keyword argument 'hold'
```

This illustrates that the bug reported in #6171 has been fixed:

```
sage: a = 1.1
sage: a.sqrt(prec=100)  # this is supposed to fail
Traceback (most recent call last):
...
TypeError: sqrt() got an unexpected keyword argument 'prec'
sage: sqrt(a, prec=100)
1.0488088481701515469914535137
sage: sqrt(4.00, prec=250)
2.0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

One can use numpy input as well:

```
sage: import numpy
sage: a = numpy.arange(2,5)
sage: sqrt(a)
array([ 1.41421356,  1.73205081,  2.        ])
```

CHAPTER

NINE

MISCELLANEOUS SPECIAL FUNCTIONS

AUTHORS:

- David Joyner (2006-13-06): initial version
- David Joyner (2006-30-10): bug fixes to pari wrappers of Bessel functions, hypergeometric_U
- William Stein (2008-02): Impose some sanity checks.
- David Joyner (2008-04-23): addition of elliptic integrals

This module provides easy access to many of Maxima and PARI's special functions.

Maxima's special functions package (which includes spherical harmonic functions, spherical Bessel functions (of the 1st and 2nd kind), and spherical Hankel functions (of the 1st and 2nd kind)) was written by Barton Willis of the University of Nebraska at Kearney. It is released under the terms of the General Public License (GPL).

Support for elliptic functions and integrals was written by Raymond Toy. It is placed under the terms of the General Public License (GPL) that governs the distribution of Maxima.

Next, we summarize some of the properties of the functions implemented here.

- Spherical harmonics: Laplace's equation in spherical coordinates is:

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial f}{\partial r}\right)+\frac{1}{r^2}\sin\theta\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial f}{\partial\theta}\right)+\frac{1}{r^2\sin^2\theta}\frac{\partial^2 f}{\partial\varphi^2}=0.$$

  Note that the spherical coordinates θ and φ are defined here as follows: θ is the colatitude or polar angle, ranging from $0\leq\theta\leq\pi$ and φ the azimuth or longitude, ranging from $0\leq\varphi<2\pi$.

  The general solution which remains finite towards infinity is a linear combination of functions of the form

$$r^{-1-\ell}\cos(m\varphi)P_\ell^m(\cos\theta)$$

  and

$$r^{-1-\ell}\sin(m\varphi)P_\ell^m(\cos\theta)$$

  where P_ℓ^m are the associated Legendre polynomials, and with integer parameters $\ell\geq 0$ and m from 0 to ℓ. Put in another way, the solutions with integer parameters $\ell\geq 0$ and $-\ell\leq m\leq\ell$, can be written as linear combinations of:

$$U_{\ell,m}(r,\theta,\varphi)=r^{-1-\ell}Y_\ell^m(\theta,\varphi)$$

  where the functions Y are the spherical harmonic functions with parameters ℓ, m, which can be written as:

$$Y_\ell^m(\theta,\varphi)=\sqrt{\frac{(2\ell+1)}{4\pi}\frac{(\ell-m)!}{(\ell+m)!}}\cdot e^{im\varphi}\cdot P_\ell^m(\cos\theta).$$

The spherical harmonics obey the normalisation condition

$$\int_{\theta=0}^{\pi}\int_{\varphi=0}^{2\pi} Y_\ell^m Y_{\ell'}^{m'*}\, d\Omega = \delta_{\ell\ell'}\delta_{mm'} \qquad d\Omega = \sin\theta\, d\varphi\, d\theta.$$

- When solving for separable solutions of Laplace's equation in spherical coordinates, the radial equation has the form:

$$x^2\frac{d^2y}{dx^2} + 2x\frac{dy}{dx} + [x^2 - n(n+1)]y = 0.$$

The spherical Bessel functions j_n and y_n, are two linearly independent solutions to this equation. They are related to the ordinary Bessel functions J_n and Y_n by:

$$j_n(x) = \sqrt{\frac{\pi}{2x}}J_{n+1/2}(x),$$

$$y_n(x) = \sqrt{\frac{\pi}{2x}}Y_{n+1/2}(x) = (-1)^{n+1}\sqrt{\frac{\pi}{2x}}J_{-n-1/2}(x).$$

- For $x > 0$, the confluent hypergeometric function $y = U(a,b,x)$ is defined to be the solution to Kummer's differential equation

$$xy'' + (b-x)y' - ay = 0,$$

which satisfies $U(a,b,x) \sim x^{-a}$, as $x \to \infty$. (There is a linearly independent solution, called Kummer's function $M(a,b,x)$, which is not implemented.)

  - The incomplete elliptic integrals (of the first kind, etc.) are:

$$\int_0^\phi \frac{1}{\sqrt{1-m\sin(x)^2}}\,dx,$$
$$\int_0^\phi \sqrt{1-m\sin(x)^2}\,dx,$$
$$\int_0^\phi \frac{\sqrt{1-mt^2}}{\sqrt{(1-t^2)}}\,dx,$$
$$\int_0^\phi \frac{1}{\sqrt{1-m\sin(x)^2}\sqrt{1-n\sin(x)^2}}\,dx,$$

and the complete ones are obtained by taking $\phi = \pi/2$.

REFERENCES:

- Abramowitz and Stegun: Handbook of Mathematical Functions, http://www.math.sfu.ca/~cbm/aands/
- http://en.wikipedia.org/wiki/Spherical_harmonics
- http://en.wikipedia.org/wiki/Helmholtz_equation
- Online Encyclopedia of Special Function http://algo.inria.fr/esf/index.html

TODO: Resolve weird bug in commented out code in hypergeometric_U below.

AUTHORS:

- David Joyner and William Stein

Added 16-02-2008 (wdj): optional calls to scipy and replace all '#random' by '...' (both at the request of William Stein)

Warning: SciPy's versions are poorly documented and seem less accurate than the Maxima and PARI versions; typically they are limited by hardware floats precision.

class sage.functions.special.**EllipticE**

Bases: sage.functions.special.MaximaFunction

This returns the value of the "incomplete elliptic integral of the second kind,"

$$\int_0^\phi \sqrt{1 - m\sin(x)^2}\,dx,$$

i.e., `integrate(sqrt(1 - m*sin(x)^2), x, 0, phi)`. Taking $\phi = \pi/2$ gives `elliptic_ec`.

EXAMPLES:

```
sage: z = var("z")
sage: # this is still wrong: must be abs(sin(z)) + 2*round(z/pi)
sage: elliptic_e(z, 1)
2*round(z/pi) + sin(z)
sage: elliptic_e(z, 0)
z
sage: elliptic_e(0.5, 0.1)  # abs tol 2e-15
0.498011394498832
```

class sage.functions.special.**EllipticEC**

Bases: sage.functions.special.MaximaFunction

This returns the value of the "complete elliptic integral of the second kind,"

$$\int_0^{\pi/2} \sqrt{1 - m\sin(x)^2}\,dx.$$

EXAMPLES:

```
sage: elliptic_ec(0.1)
1.53075763689776
sage: elliptic_ec(x).diff()
1/2*(elliptic_ec(x) - elliptic_kc(x))/x

sage: loads(dumps(elliptic_ec))
elliptic_ec
```

class sage.functions.special.**EllipticEU**

Bases: sage.functions.special.MaximaFunction

Return the value of the "incomplete elliptic integral of the second kind,"

$$\int_0^u \mathrm{dn}(x, m)^2\,dx = \int_0^\tau \frac{\sqrt{1 - mx^2}}{\sqrt{1 - x^2}}\,dx.$$

where $\tau = \mathrm{sn}(u, m)$.

EXAMPLES:

```
sage: elliptic_eu (0.5, 0.1)
0.496054551286597
```

class sage.functions.special.**EllipticF**
Bases: sage.functions.special.MaximaFunction

This returns the value of the "incomplete elliptic integral of the first kind,"

$$\int_0^\phi \frac{dx}{\sqrt{1-m\sin(x)^2}},$$

i.e., `integrate(1/sqrt(1 - m*sin(x)^2), x, 0, phi)`. Taking $\phi = \pi/2$ gives `elliptic_kc`.

EXAMPLES:

```
sage: z = var("z")
sage: elliptic_f (z, 0)
z
sage: elliptic_f (z, 1)
log(tan(1/4*pi + 1/2*z))
sage: elliptic_f (0.2, 0.1)
0.200132506747543
```

class sage.functions.special.**EllipticKC**
Bases: sage.functions.special.MaximaFunction

This returns the value of the "complete elliptic integral of the first kind,"

$$\int_0^{\pi/2} \frac{dx}{\sqrt{1-m\sin(x)^2}}.$$

EXAMPLES:

```
sage: elliptic_kc(0.5)
1.85407467730137
sage: elliptic_f(RR(pi/2), 0.5)
1.85407467730137
```

class sage.functions.special.**EllipticPi**
Bases: sage.functions.special.MaximaFunction

This returns the value of the "incomplete elliptic integral of the third kind,"

$$\text{elliptic\_pi}(n,t,m) = \int_0^t \frac{dx}{(1-n\sin(x)^2)\sqrt{1-m\sin(x)^2}}.$$

INPUT:

- `n` – a real number, called the "characteristic"
- `t` – a real number, called the "amplitude"
- `m` – a real number, called the "parameter"

EXAMPLES:

```
sage: N(elliptic_pi(1, pi/4, 1))
1.14779357469632
```

Compare the value computed by Maxima to the definition as a definite integral (using GSL):

```
sage: elliptic_pi(0.1, 0.2, 0.3)
0.200665068220979
sage: numerical_integral(1/(1-0.1*sin(x)^2)/sqrt(1-0.3*sin(x)^2), 0.0, 0.2)
(0.2006650682209791, 2.227829789769088e-15)
```

ALGORITHM:

Numerical evaluation and symbolic manipulation are provided by Maxima.

REFERENCES:

- Abramowitz and Stegun: Handbook of Mathematical Functions, section 17.7 http://www.math.sfu.ca/~cbm/aands/
- Elliptic Functions in Maxima

class sage.functions.special.**MaximaFunction**(*name*, *nargs=2*, *conversions={}*)
Bases: sage.symbolic.function.BuiltinFunction

EXAMPLES:

```
sage: from sage.functions.special import MaximaFunction
sage: f = MaximaFunction("jacobi_sn")
sage: f(1,1)
tanh(1)
sage: f(1/2,1/2).n()
0.470750473655657
```

class sage.functions.special.**SphericalHarmonic**
Bases: sage.symbolic.function.BuiltinFunction

Returns the spherical harmonic function $Y_n^m(\theta, \varphi)$.

For integers $n > -1$, $|m| \leq n$, simplification is done automatically. Numeric evaluation is supported for complex n and m.

EXAMPLES:

```
sage: x, y = var('x, y')
sage: spherical_harmonic(3, 2, x, y)
15/4*sqrt(7/30)*cos(x)*e^(2*I*y)*sin(x)^2/sqrt(pi)
sage: spherical_harmonic(3, 2, 1, 2)
15/4*sqrt(7/30)*cos(1)*e^(4*I)*sin(1)^2/sqrt(pi)
sage: spherical_harmonic(3 + I, 2., 1, 2)
-0.351154337307488 - 0.415562233975369*I
sage: latex(spherical_harmonic(3, 2, x, y, hold=True))
Y_{3}^{2}\left(x, y\right)
sage: spherical_harmonic(1, 2, x, y)
0
```

sage.functions.special.**elliptic_j**(*z*)
Returns the elliptic modular j-function evaluated at z.

INPUT:

- z (complex) – a complex number with positive imaginary part.

OUTPUT:

(complex) The value of $j(z)$.

ALGORITHM:

Calls the pari function ellj().

AUTHOR:

John Cremona

EXAMPLES:

```
sage: elliptic_j(CC(i))
1728.00000000000
sage: elliptic_j(sqrt(-2.0))
8000.00000000000
sage: z = ComplexField(100)(1,sqrt(11))/2
sage: elliptic_j(z)
-32768.000...
sage: elliptic_j(z).real().round()
-32768
```

::

```
sage: tau = (1 + sqrt(-163))/2
sage: (-elliptic_j(tau.n(100)).real().round())^(1/3)
640320
```

sage.functions.special.**error_fcn**(*t*)

The complementary error function $\frac{2}{\sqrt{\pi}} \int_t^\infty e^{-x^2} dx$ (t belongs to RR). This function is currently always evaluated immediately.

EXAMPLES:

```
sage: error_fcn(6)
2.15197367124989e-17
sage: error_fcn(RealField(100)(1/2))
0.47950012218695346231725334611
```

Note this is literally equal to $1 - erf(t)$:

```
sage: 1 - error_fcn(0.5)
0.520499877813047
sage: erf(0.5)
0.520499877813047
```

sage.functions.special.**hypergeometric_U**(*alpha*, *beta*, *x*, *algorithm='pari'*, *prec=53*)

Default is a wrap of PARI's hyperu(alpha,beta,x) function. Optionally, algorithm = "scipy" can be used.

The confluent hypergeometric function $y = U(a, b, x)$ is defined to be the solution to Kummer's differential equation

$$xy'' + (b - x)y' - ay = 0.$$

This satisfies $U(a, b, x) \sim x^{-a}$, as $x \to \infty$, and is sometimes denoted `x^{-a}2_F_0(a,1+a-b,-1/x)`. This is not the same as Kummer's M-hypergeometric function, denoted sometimes as `_1F_1(alpha,beta,x)`, though it satisfies the same DE that U does.

Warning: In the literature, both are called "Kummer confluent hypergeometric" functions.

EXAMPLES:

```
sage: hypergeometric_U(1,1,1,"scipy")
0.596347362323...
sage: hypergeometric_U(1,1,1)
0.59634736232319...
sage: hypergeometric_U(1,1,1,"pari",70)
0.59634736232319407434...
```

sage.functions.special.**maxima_function**(*name*)

Returns a function which is evaluated both symbolically and numerically via Maxima. In particular, it returns

an instance of MaximaFunction.

Note: This function is cached so that duplicate copies of the same function are not created.

EXAMPLES:

```
sage: spherical_hankel2(2,i)
-e
```

sage.functions.special.**meval**(*x*)

Return x evaluated in Maxima, then returned to Sage.

This is used to evaluate several of these special functions.

TEST:

```
sage: from sage.functions.special import spherical_bessel_J
sage: spherical_bessel_J(2.,3.)      # rel tol 1e-10
0.2986374970757335
```

sage.functions.special.**spherical_bessel_J**(*n*, *var*, *algorithm='maxima'*)

Returns the spherical Bessel function of the first kind for integers n >= 1.

Reference: AS 10.1.8 page 437 and AS 10.1.15 page 439.

EXAMPLES:

```
sage: spherical_bessel_J(2,x)
((3/x^2 - 1)*sin(x) - 3*cos(x)/x)/x
sage: spherical_bessel_J(1, 5.2, algorithm='scipy')
-0.12277149950007...
sage: spherical_bessel_J(1, 3, algorithm='scipy')
0.345677499762355...
```

sage.functions.special.**spherical_bessel_Y**(*n*, *var*, *algorithm='maxima'*)

Returns the spherical Bessel function of the second kind for integers n -1.

Reference: AS 10.1.9 page 437 and AS 10.1.15 page 439.

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: spherical_bessel_Y(2,x)
-((3/x^2 - 1)*cos(x) + 3*sin(x)/x)/x
```

sage.functions.special.**spherical_hankel1**(*n*, *var*)

Returns the spherical Hankel function of the first kind for integers $n > -1$, written as a string. Reference: AS 10.1.36 page 439.

EXAMPLES:

```
sage: spherical_hankel1(2, x)
(I*x^2 - 3*x - 3*I)*e^(I*x)/x^3
```

sage.functions.special.**spherical_hankel2**(*n*, *x*)

Returns the spherical Hankel function of the second kind for integers $n > -1$, written as a string. Reference: AS 10.1.17 page 439.

EXAMPLES:

```
sage: spherical_hankel2(2, x)
(-I*x^2 - 3*x + 3*I)*e^(-I*x)/x^3
```

Here I = sqrt(-1).

CHAPTER

TEN

HYPERGEOMETRIC FUNCTIONS

This module implements manipulation of infinite hypergeometric series represented in standard parametric form (as ${}_pF_q$ functions).

AUTHORS:

- Fredrik Johansson (2010): initial version
- Eviatar Bach (2013): major changes

EXAMPLES:

Examples from trac ticket #9908:

```
sage: maxima('integrate(bessel_j(2, x), x)').sage()
1/24*x^3*hypergeometric((3/2,), (5/2, 3), -1/4*x^2)
sage: sum(((2*I)^x/(x^3 + 1)*(1/4)^x), x, 0, oo)
hypergeometric((1, 1, -1/2*I*sqrt(3) - 1/2, 1/2*I*sqrt(3) - 1/2),...
(2, -1/2*I*sqrt(3) + 1/2, 1/2*I*sqrt(3) + 1/2), 1/2*I)
sage: sum((-1)^x/((2*x + 1)*factorial(2*x + 1)), x, 0, oo)
hypergeometric((1/2,), (3/2, 3/2), -1/4)
```

Simplification (note that `simplify_full` does not yet call `simplify_hypergeometric`):

```
sage: hypergeometric([-2], [], x).simplify_hypergeometric()
x^2 - 2*x + 1
sage: hypergeometric([], [], x).simplify_hypergeometric()
e^x
sage: a = hypergeometric((hypergeometric((), (), x),), (),
....:                    hypergeometric((), (), x))
sage: a.simplify_hypergeometric()
1/((-e^x + 1)^e^x)
sage: a.simplify_hypergeometric(algorithm='sage')
(-e^x + 1)^(-e^x)
```

Equality testing:

```
sage: bool(hypergeometric([], [], x).derivative(x) ==
....:      hypergeometric([], [], x))  # diff(e^x, x) == e^x
True
sage: bool(hypergeometric([], [], x) == hypergeometric([], [1], x))
False
```

Computing terms and series:

```
sage: z = var('z')
sage: hypergeometric([], [], z).series(z, 0)
```

```
Order(1)
sage: hypergeometric([], [], z).series(z, 1)
1 + Order(z)
sage: hypergeometric([], [], z).series(z, 2)
1 + 1*z + Order(z^2)
sage: hypergeometric([], [], z).series(z, 3)
1 + 1*z + 1/2*z^2 + Order(z^3)

sage: hypergeometric([-2], [], z).series(z, 3)
1 + (-2)*z + 1*z^2
sage: hypergeometric([-2], [], z).series(z, 6)
1 + (-2)*z + 1*z^2
sage: hypergeometric([-2], [], z).series(z, 6).is_terminating_series()
True
sage: hypergeometric([-2], [], z).series(z, 2)
1 + (-2)*z + Order(z^2)
sage: hypergeometric([-2], [], z).series(z, 2).is_terminating_series()
False

sage: hypergeometric([1], [], z).series(z, 6)
1 + 1*z + 1*z^2 + 1*z^3 + 1*z^4 + 1*z^5 + Order(z^6)
sage: hypergeometric([], [1/2], -z^2/4).series(z, 11)
1 + (-1/2)*z^2 + 1/24*z^4 + (-1/720)*z^6 + 1/40320*z^8 +...
(-1/3628800)*z^10 + Order(z^11)

sage: hypergeometric([1], [5], x).series(x, 5)
1 + 1/5*x + 1/30*x^2 + 1/210*x^3 + 1/1680*x^4 + Order(x^5)

sage: sum(hypergeometric([1, 2], [3], 1/3).terms(6)).n()
1.29788359788360
sage: hypergeometric([1, 2], [3], 1/3).n()
1.29837194594696
sage: hypergeometric([], [], x).series(x, 20)(x=1).n() == e.n()
True
```

Plotting:

```
sage: plot(hypergeometric([1, 1], [3, 3, 3], x), x, -30, 30)
Graphics object consisting of 1 graphics primitive
sage: complex_plot(hypergeometric([x], [], 2), (-1, 1), (-1, 1))
Graphics object consisting of 1 graphics primitive
```

Numeric evaluation:

```
sage: hypergeometric([1], [], 1/10).n()  # geometric series
1.11111111111111
sage: hypergeometric([], [], 1).n()  # e
2.71828182845905
sage: hypergeometric([], [], 3., hold=True)
hypergeometric((), (), 3.00000000000000)
sage: hypergeometric([1, 2, 3], [4, 5, 6], 1/2).n()
1.02573619590134
sage: hypergeometric([1, 2, 3], [4, 5, 6], 1/2).n(digits=30)
1.02573619590133865036584139535
sage: hypergeometric([5 - 3*I], [3/2, 2 + I, sqrt(2)], 4 + I).n()
5.52605111678805 - 7.86331357527544*I
sage: hypergeometric((10, 10), (50,), 2.)
-1705.75733163554 - 356.749986056024*I
```

Conversions:

```
sage: maxima(hypergeometric([1, 1, 1], [3, 3, 3], x))
hypergeometric([1,1,1],[3,3,3],_SAGE_VAR_x)
sage: hypergeometric((5, 4), (4, 4), 3)._sympy_()
hyper((5, 4), (4, 4), 3)
sage: hypergeometric((5, 4), (4, 4), 3)._mathematica_init_()
'HypergeometricPFQ[{5,4},{4,4},3]'
```

Arbitrary level of nesting for conversions:

```
sage: maxima(nest(lambda y: hypergeometric([y], [], x), 3, 1))
1/(1-_SAGE_VAR_x)^(1/(1-_SAGE_VAR_x)^(1/(1-_SAGE_VAR_x)))
sage: maxima(nest(lambda y: hypergeometric([y], [3], x), 3, 1))._sage_()
hypergeometric((hypergeometric((hypergeometric((1,), (3,), x),), (3,),...
x),), (3,), x)
sage: nest(lambda y: hypergeometric([y], [], x), 3, 1)._mathematica_init_()
'HypergeometricPFQ[{HypergeometricPFQ[{HypergeometricPFQ[{1},{},x]},...
```

class sage.functions.hypergeometric.**Hypergeometric**

Bases: sage.symbolic.function.BuiltinFunction

Represents a (formal) generalized infinite hypergeometric series. It is defined as

$$ {}_pF_q(a_1,\ldots,a_p;b_1,\ldots,b_q;z) = \sum_{n=0}^{\infty} \frac{(a_1)_n \cdots (a_p)_n}{(b_1)_n \cdots (b_q)_n} \frac{z^n}{n!}, $$

where $(x)_n$ is the rising factorial.

class EvaluationMethods

deflated(*self*, *a*, *b*, *z*)

Rewrite as a linear combination of functions of strictly lower degree by eliminating all parameters `a[i]` and `b[j]` such that `a[i]` = `b[i]` + `m` for nonnegative integer `m`.

EXAMPLES:

```
sage: x = hypergeometric([6, 1], [3, 4, 5], 10)
sage: y = x.deflated()
sage: y
1/252*hypergeometric((4,), (7, 8), 10)
 + 1/12*hypergeometric((3,), (6, 7), 10)
 + 1/2*hypergeometric((2,), (5, 6), 10)
 + hypergeometric((1,), (4, 5), 10)
sage: x.n(); y.n()
2.87893612686782
2.87893612686782

sage: x = hypergeometric([6, 7], [3, 4, 5], 10)
sage: y = x.deflated()
sage: y
25/27216*hypergeometric((), (11,), 10)
 + 25/648*hypergeometric((), (10,), 10)
 + 265/504*hypergeometric((), (9,), 10)
 + 181/63*hypergeometric((), (8,), 10)
 + 19/3*hypergeometric((), (7,), 10)
 + 5*hypergeometric((), (6,), 10)
 + hypergeometric((), (5,), 10)
sage: x.n(); y.n()
```

```
63.0734110716969
63.0734110716969
```

eliminate_parameters (*self*, *a*, *b*, *z*)

Eliminate repeated parameters by pairwise cancellation of identical terms in `a` and `b`.

EXAMPLES:

```
sage: hypergeometric([1, 1, 2, 5], [5, 1, 4],
....:                1/2).eliminate_parameters()
hypergeometric((1, 2), (4,), 1/2)
sage: hypergeometric([x], [x], x).eliminate_parameters()
hypergeometric((), (), x)
sage: hypergeometric((5, 4), (4, 4), 3).eliminate_parameters()
hypergeometric((5,), (4,), 3)
```

is_absolutely_convergent (*self*, *a*, *b*, *z*)

Determine whether `self` converges absolutely as an infinite series. `False` is returned if not all terms are finite.

EXAMPLES:

Degree giving infinite radius of convergence:

```
sage: hypergeometric([2, 3], [4, 5],
....:                6).is_absolutely_convergent()
True
sage: hypergeometric([2, 3], [-4, 5],
....:                6).is_absolutely_convergent()  # undefined
False
sage: (hypergeometric([2, 3], [-4, 5], Infinity)
....:  .is_absolutely_convergent())  # undefined
False
```

Ordinary geometric series (unit radius of convergence):

```
sage: hypergeometric([1], [], 1/2).is_absolutely_convergent()
True
sage: hypergeometric([1], [], 2).is_absolutely_convergent()
False
sage: hypergeometric([1], [], 1).is_absolutely_convergent()
False
sage: hypergeometric([1], [], -1).is_absolutely_convergent()
False
sage: hypergeometric([1], [], -1).n()  # Sum still exists
0.500000000000000
```

Degree $p = q + 1$ (unit radius of convergence):

```
sage: hypergeometric([2, 3], [4], 6).is_absolutely_convergent()
False
sage: hypergeometric([2, 3], [4], 1).is_absolutely_convergent()
False
sage: hypergeometric([2, 3], [5], 1).is_absolutely_convergent()
False
sage: hypergeometric([2, 3], [6], 1).is_absolutely_convergent()
True
sage: hypergeometric([-2, 3], [4],
....:                5).is_absolutely_convergent()
True
sage: hypergeometric([2, -3], [4],
....:                5).is_absolutely_convergent()
True
```

```
sage: hypergeometric([2, -3], [-4],
....:                5).is_absolutely_convergent()
True
sage: hypergeometric([2, -3], [-1],
....:                5).is_absolutely_convergent()
False
```

Degree giving zero radius of convergence:

```
sage: hypergeometric([1, 2, 3], [4],
....:                2).is_absolutely_convergent()
False
sage: hypergeometric([1, 2, 3], [4],
....:                1/2).is_absolutely_convergent()
False
sage: (hypergeometric([1, 2, -3], [4], 1/2)
....:  .is_absolutely_convergent())  # polynomial
True
```

is_terminating (*self*, *a*, *b*, *z*)

Determine whether the series represented by self terminates after a finite number of terms, i.e. whether any of the numerator parameters are nonnegative integers (with no preceding nonnegative denominator parameters), or $z = 0$.

If terminating, the series represents a polynomial of z.

EXAMPLES:

```
sage: hypergeometric([1, 2], [3, 4], x).is_terminating()
False
sage: hypergeometric([1, -2], [3, 4], x).is_terminating()
True
sage: hypergeometric([1, -2], [], x).is_terminating()
True
```

is_termwise_finite (*self*, *a*, *b*, *z*)

Determine whether all terms of `self` are finite. Any infinite terms or ambiguous terms beyond the first zero, if one exists, are ignored.

Ambiguous cases (where a term is the product of both zero and an infinity) are not considered finite.

EXAMPLES:

```
sage: hypergeometric([2], [3, 4], 5).is_termwise_finite()
True
sage: hypergeometric([2], [-3, 4], 5).is_termwise_finite()
False
sage: hypergeometric([-2], [-3, 4], 5).is_termwise_finite()
True
sage: hypergeometric([-3], [-3, 4],
....:                5).is_termwise_finite()  # ambiguous
False

sage: hypergeometric([0], [-1], 5).is_termwise_finite()
True
sage: hypergeometric([0], [0],
....:                5).is_termwise_finite()  # ambiguous
False
sage: hypergeometric([1], [2], Infinity).is_termwise_finite()
False
sage: (hypergeometric([0], [0], Infinity)
....:  .is_termwise_finite())  # ambiguous
```

```
False
sage: (hypergeometric([0], [], Infinity)
....:  .is_termwise_finite())  # ambiguous
False
```

sorted_parameters(*self*, *a*, *b*, *z*)

Return with parameters sorted in a canonical order.

EXAMPLES:

```
sage: hypergeometric([2, 1, 3], [5, 4],
....:                1/2).sorted_parameters()
hypergeometric((1, 2, 3), (4, 5), 1/2)
```

terms(*self*, *a*, *b*, *z*, *n=None*)

Generate the terms of `self` (optionally only `n` terms).

EXAMPLES:

```
sage: list(hypergeometric([-2, 1], [3, 4], x).terms())
[1, -1/6*x, 1/120*x^2]
sage: list(hypergeometric([-2, 1], [3, 4], x).terms(2))
[1, -1/6*x]
sage: list(hypergeometric([-2, 1], [3, 4], x).terms(0))
[]
```

sage.functions.hypergeometric.**closed_form**(*hyp*)

Try to evaluate `hyp` in closed form using elementary (and other simple) functions.

It may be necessary to call `Hypergeometric.deflated()` first to find some closed forms.

EXAMPLES:

```
sage: from sage.functions.hypergeometric import closed_form
sage: var('a b c z')
(a, b, c, z)
sage: closed_form(hypergeometric([1], [], 1 + z))
-1/z
sage: closed_form(hypergeometric([], [], 1 + z))
e^(z + 1)
sage: closed_form(hypergeometric([], [1/2], 4))
cosh(4)
sage: closed_form(hypergeometric([], [3/2], 4))
1/4*sinh(4)
sage: closed_form(hypergeometric([], [5/2], 4))
3/16*cosh(4) - 3/64*sinh(4)
sage: closed_form(hypergeometric([], [-3/2], 4))
19/3*cosh(4) - 4*sinh(4)
sage: closed_form(hypergeometric([-3, 1], [var('a')], z))
-3*z/a + 6*z^2/((a + 1)*a) - 6*z^3/((a + 2)*(a + 1)*a) + 1
sage: closed_form(hypergeometric([-3, 1/3], [-4], z))
7/162*z^3 + 1/9*z^2 + 1/4*z + 1
sage: closed_form(hypergeometric([], [], z))
e^z
sage: closed_form(hypergeometric([a], [], z))
(-z + 1)^(-a)
sage: closed_form(hypergeometric([1, 1, 2], [1, 1], z))
(z - 1)^(-2)
sage: closed_form(hypergeometric([2, 3], [1], x))
-1/(x - 1)^3 + 3*x/(x - 1)^4
sage: closed_form(hypergeometric([1/2], [3/2], -5))
1/10*sqrt(5)*sqrt(pi)*erf(sqrt(5))
```

```
sage: closed_form(hypergeometric([2], [5], 3))
4
sage: closed_form(hypergeometric([2], [5], 5))
48/625*e^5 + 612/625
sage: closed_form(hypergeometric([1/2, 7/2], [3/2], z))
1/5*z^2/(-z + 1)^(5/2) + 2/3*z/(-z + 1)^(3/2) + 1/sqrt(-z + 1)
sage: closed_form(hypergeometric([1/2, 1], [2], z))
-2*(sqrt(-z + 1) - 1)/z
sage: closed_form(hypergeometric([1, 1], [2], z))
-log(-z + 1)/z
sage: closed_form(hypergeometric([1, 1], [3], z))
-2*((z - 1)*log(-z + 1)/z - 1)/z
sage: closed_form(hypergeometric([1, 1, 1], [2, 2], x))
hypergeometric((1, 1, 1), (2, 2), x)
```

sage.functions.hypergeometric.**rational_param_as_tuple**(x)

Utility function for converting rational ${}_pF_q$ parameters to tuples (which mpmath handles more efficiently).

EXAMPLES:

```
sage: from sage.functions.hypergeometric import rational_param_as_tuple
sage: rational_param_as_tuple(1/2)
(1, 2)
sage: rational_param_as_tuple(3)
3
sage: rational_param_as_tuple(pi)
pi
```

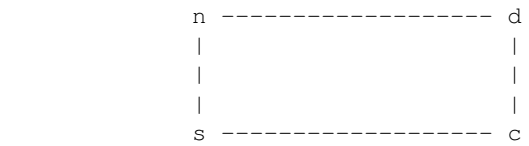
CHAPTER

ELEVEN

JACOBI ELLIPTIC FUNCTIONS

This module implements the 12 Jacobi elliptic functions, along with their inverses and the Jacobi amplitude function.

Jacobi elliptic functions can be thought of as generalizations of both ordinary and hyperbolic trig functions. There are twelve Jacobian elliptic functions. Each of the twelve corresponds to an arrow drawn from one corner of a rectangle to another.



Each of the corners of the rectangle are labeled, by convention, `s`, `c`, `d`, and `n`. The rectangle is understood to be lying on the complex plane, so that `s` is at the origin, `c` is on the real axis, and `n` is on the imaginary axis. The twelve Jacobian elliptic functions are then $\mathrm{pq}(x)$, where `p` and `q` are one of the letters `s`, `c`, `d`, `n`.

The Jacobian elliptic functions are then the unique doubly-periodic, meromorphic functions satisfying the following three properties:

1. There is a simple zero at the corner `p`, and a simple pole at the corner `q`.
2. The step from `p` to `q` is equal to half the period of the function $\mathrm{pq}(x)$; that is, the function $\mathrm{pq}(x)$ is periodic in the direction `pq`, with the period being twice the distance from `p` to `q`. $\mathrm{pq}(x)$ is periodic in the other two directions as well, with a period such that the distance from `p` to one of the other corners is a quarter period.
3. If the function $\mathrm{pq}(x)$ is expanded in terms of x at one of the corners, the leading term in the expansion has a coefficient of 1. In other words, the leading term of the expansion of $\mathrm{pq}(x)$ at the corner `p` is x; the leading term of the expansion at the corner `q` is $1/x$, and the leading term of an expansion at the other two corners is 1.

We can write

$$\mathrm{pq}(x) = \frac{\mathrm{pr}(x)}{\mathrm{qr}(x)}$$

where `p`, `q`, and `r` are any of the letters `s`, `c`, `d`, `n`, with the understanding that $\mathrm{ss} = \mathrm{cc} = \mathrm{dd} = \mathrm{nn} = 1$.

Let

$$u = \int_0^\phi \frac{d\theta}{\sqrt{1 - m\sin^2\theta}},$$

then the *Jacobi elliptic function* $\mathrm{sn}(u)$ is given by

$$\mathrm{sn}\, u = \sin\phi$$

and $\mathrm{cn}(u)$ is given by

$$\mathrm{cn}\, u = \cos\phi$$

and

$$\mathrm{dn}\, u = \sqrt{1 - m \sin^2 \phi}.$$

To emphasize the dependence on m, one can write $\mathrm{sn}(u|m)$ for example (and similarly for cn and dn). This is the notation used below.

For a given k with $0 < k < 1$ they therefore are solutions to the following nonlinear ordinary differential equations:

- $\mathrm{sn}\,(x; k)$ solves the differential equations

$$\frac{d^2y}{dx^2} + (1 + k^2)y - 2k^2y^3 = 0 \quad \text{and} \quad \left(\frac{dy}{dx}\right)^2 = (1 - y^2)(1 - k^2y^2).$$

- $\mathrm{cn}(x; k)$ solves the differential equations

$$\frac{d^2y}{dx^2} + (1 - 2k^2)y + 2k^2y^3 = 0 \quad \text{and} \quad \left(\frac{dy}{dx}\right)^2 = (1 - y^2)(1 - k^2 + k^2y^2).$$

- $\mathrm{dn}(x; k)$ solves the differential equations

$$\frac{d^2y}{dx^2} - (2 - k^2)y + 2y^3 = 0 \quad \text{and} \quad \left(\frac{dy}{dx}\right)^2 = y^2(1 - k^2 - y^2).$$

  If $K(m)$ denotes the complete elliptic integral of the first kind (named `elliptic_kc` in Sage), the elliptic functions $\mathrm{sn}(x|m)$ and $\mathrm{cn}(x|m)$ have real periods $4K(m)$, whereas $\mathrm{dn}(x|m)$ has a period $2K(m)$. The limit $m \to 0$ gives $K(0) = \pi/2$ and trigonometric functions: $\mathrm{sn}(x|0) = \sin x$, $\mathrm{cn}(x|0) = \cos x$, $\mathrm{dn}(x|0) = 1$. The limit $m \to 1$ gives $K(1) \to \infty$ and hyperbolic functions: $\mathrm{sn}(x|1) = \tanh x$, $\mathrm{cn}(x|1) = \mathrm{sech}\, x$, $\mathrm{dn}(x|1) = \mathrm{sech}\, x$.

REFERENCES:

- Wikipedia article Jacobi's_elliptic_functions

AUTHORS:

- David Joyner (2006): initial version
- Eviatar Bach (2013): complete rewrite, new numerical evaluation, and addition of the Jacobi amplitude function

class `sage.functions.jacobi.`**`InverseJacobi`**(*kind*)
Bases: `sage.symbolic.function.BuiltinFunction`

Base class for the inverse Jacobi elliptic functions.

class `sage.functions.jacobi.`**`Jacobi`**(*kind*)
Bases: `sage.symbolic.function.BuiltinFunction`

Base class for the Jacobi elliptic functions.

class `sage.functions.jacobi.`**`JacobiAmplitude`**
Bases: `sage.symbolic.function.BuiltinFunction`

The Jacobi amplitude function $\mathrm{am}(x|m) = \int_0^x \mathrm{dn}(t|m)dt$ for $-K(m) \le x \le K(m)$, $F(\mathrm{am}(x|m)|m) = x$.

`sage.functions.jacobi.`**`inverse_jacobi`**(*kind*, *x*, *m*, ***kwargs*)
The inverses of the 12 Jacobi elliptic functions. They have the property that

$$\mathrm{pq}(\mathrm{arcpq}(x|m)|m) = \mathrm{pq}(\mathrm{pq}^{-1}(x|m)|m) = x.$$

INPUT:

- `kind` – a string of the form `'pq'`, where `p`, `q` are in `c`, `d`, `n`, `s`
- `x` – a real number
- `m` – a real number; note that $m = k^2$, where k is the elliptic modulus

EXAMPLES:

```
sage: jacobi('dn', inverse_jacobi('dn', 3, 0.4), 0.4)
3.00000000000000
sage: inverse_jacobi('dn', 10, 1/10).n(digits=50)
2.4777736267904273296523691232988240759001423661683*I
sage: inverse_jacobi_dn(x, 1)
arcsech(x)
sage: inverse_jacobi_dn(1, 3)
0
sage: m = var('m')
sage: z = inverse_jacobi_dn(x, m).series(x, 4).subs(x=0.1, m=0.7)
sage: jacobi_dn(z, 0.7)
0.0999892750039819...
sage: inverse_jacobi_nd(x, 1)
arccosh(x)
sage: inverse_jacobi_nd(1, 2)
0
sage: inverse_jacobi_ns(10^-5, 3).n()
5.77350269202456e-6 + 1.17142008414677*I
sage: jacobi('sn', 1/2, 1/2)
jacobi_sn(1/2, 1/2)
sage: jacobi('sn', 1/2, 1/2).n()
0.470750473655657
sage: inverse_jacobi('sn', 0.47, 1/2)
0.499098231322220
sage: inverse_jacobi('sn', 0.4707504, 0.5)
0.499999911466555
sage: P = plot(inverse_jacobi('sn', x, 0.5), 0, 1)
```

sage.functions.jacobi.**inverse_jacobi_f**(*kind*, *x*, *m*)

Internal function for numerical evaluation of a continous complex branch of each inverse Jacobi function, as described in [Tee97]. Only accepts real arguments.

REFERENCES:

TESTS:

```
sage: from mpmath import ellipfun, chop
sage: from sage.functions.jacobi import inverse_jacobi_f

sage: chop(ellipfun('sn', inverse_jacobi_f('sn', 0.6, 0), 0))
mpf('0.59999999999999998')
sage: chop(ellipfun('sn', inverse_jacobi_f('sn', 0.6, 1), 1))
mpf('0.59999999999999998')
sage: chop(ellipfun('sn', inverse_jacobi_f('sn', 0, -3), -3))
mpf('0.0')
sage: chop(ellipfun('sn', inverse_jacobi_f('sn', -1, 4), 4))
mpf('-1.0')
sage: chop(ellipfun('sn', inverse_jacobi_f('sn', 0.3, 4), 4))
mpf('0.29999999999999999')
sage: chop(ellipfun('sn', inverse_jacobi_f('sn', 0.8, 4), 4))
mpf('0.80000000000000004')

sage: chop(ellipfun('ns', inverse_jacobi_f('ns', 0.8, 0), 0))
```

```
mpf('0.80000000000000004')
sage: chop(ellipfun('ns', inverse_jacobi_f('ns', -0.7, 1), 1))
mpf('-0.69999999999999996')
sage: chop(ellipfun('ns', inverse_jacobi_f('ns', 0.01, 2), 2))
mpf('0.01')
sage: chop(ellipfun('ns', inverse_jacobi_f('ns', 0, 2), 2))
mpf('0.0')
sage: chop(ellipfun('ns', inverse_jacobi_f('ns', -10, 6), 6))
mpf('-10.0')

sage: chop(ellipfun('cn', inverse_jacobi_f('cn', -10, 0), 0))
mpf('-9.9999999999999982')
sage: chop(ellipfun('cn', inverse_jacobi_f('cn', 50, 1), 1))
mpf('50.000000000000071')
sage: chop(ellipfun('cn', inverse_jacobi_f('cn', 1, 5), 5))
mpf('1.0')
sage: chop(ellipfun('cn', inverse_jacobi_f('cn', 0.5, -5), -5))
mpf('0.5')
sage: chop(ellipfun('cn', inverse_jacobi_f('cn', -0.75, -15), -15))
mpf('-0.75000000000000022')
sage: chop(ellipfun('cn', inverse_jacobi_f('cn', 10, 0.8), 0.8))
mpf('9.9999999999999982')
sage: chop(ellipfun('cn', inverse_jacobi_f('cn', -2, 0.9), 0.9))
mpf('-2.0')

sage: chop(ellipfun('nc', inverse_jacobi_f('nc', -4, 0), 0))
mpf('-3.9999999999999987')
sage: chop(ellipfun('nc', inverse_jacobi_f('nc', 7, 1), 1))
mpf('7.0000000000000009')
sage: chop(ellipfun('nc', inverse_jacobi_f('nc', 7, 3), 3))
mpf('7.0')
sage: chop(ellipfun('nc', inverse_jacobi_f('nc', 0, 2), 2))
mpf('0.0')
sage: chop(ellipfun('nc', inverse_jacobi_f('nc', -18, -4), -4))
mpf('-17.999999999999925')

sage: chop(ellipfun('dn', inverse_jacobi_f('dn', -0.3, 1), 1))
mpf('-0.29999999999999999')
sage: chop(ellipfun('dn', inverse_jacobi_f('dn', 1, -1), -1))
mpf('1.0')
sage: chop(ellipfun('dn', inverse_jacobi_f('dn', 0.8, 0.5), 0.5))
mpf('0.80000000000000004')
sage: chop(ellipfun('dn', inverse_jacobi_f('dn', 5, -4), -4))
mpf('5.0')
sage: chop(ellipfun('dn', inverse_jacobi_f('dn', 0.4, 0.5), 0.5))
mpf('0.40000000000000002')
sage: chop(ellipfun('dn', inverse_jacobi_f('dn', -0.4, 0.5), 0.5))
mpf('-0.40000000000000002')
sage: chop(ellipfun('dn', inverse_jacobi_f('dn', -0.9, 0.5), 0.5))
mpf('-0.90000000000000002')
sage: chop(ellipfun('dn', inverse_jacobi_f('dn', -1.9, 0.2), 0.2))
mpf('-1.8999999999999999')

sage: chop(ellipfun('nd', inverse_jacobi_f('nd', -1.9, 1), 1))
mpf('-1.8999999999999999')
sage: chop(ellipfun('nd', inverse_jacobi_f('nd', 1, -1), -1))
mpf('1.0')
sage: chop(ellipfun('nd', inverse_jacobi_f('nd', 11, -6), -6))
```

```
    mpf('11.0')
    sage: chop(ellipfun('nd', inverse_jacobi_f('nd', 0, 8), 8))
    mpf('0.0')
    sage: chop(ellipfun('nd', inverse_jacobi_f('nd', -3, 0.8), 0.8))
    mpf('-2.9999999999999996')

    sage: chop(ellipfun('sc', inverse_jacobi_f('sc', -3, 0), 0))
    mpf('-3.0')
    sage: chop(ellipfun('sc', inverse_jacobi_f('sc', 2, 1), 1))
    mpf('2.0')
    sage: chop(ellipfun('sc', inverse_jacobi_f('sc', 0, 9), 9))
    mpf('0.0')
    sage: chop(ellipfun('sc', inverse_jacobi_f('sc', -7, 3), 3))
    mpf('-7.0')

    sage: chop(ellipfun('cs', inverse_jacobi_f('cs', -7, 0), 0))
    mpf('-6.9999999999999991')
    sage: chop(ellipfun('cs', inverse_jacobi_f('cs', 8, 1), 1))
    mpf('8.0')
    sage: chop(ellipfun('cs', inverse_jacobi_f('cs', 2, 6), 6))
    mpf('2.0')
    sage: chop(ellipfun('cs', inverse_jacobi_f('cs', 0, 4), 4))
    mpf('0.0')
    sage: chop(ellipfun('cs', inverse_jacobi_f('cs', -6, 8), 8))
    mpf('-6.0000000000000018')

    sage: chop(ellipfun('cd', inverse_jacobi_f('cd', -6, 0), 0))
    mpf('-6.0000000000000009')
    sage: chop(ellipfun('cd', inverse_jacobi_f('cd', 1, 3), 3))
    mpf('1.0')
    sage: chop(ellipfun('cd', inverse_jacobi_f('cd', 6, 8), 8))
    mpf('6.0000000000000027')

    sage: chop(ellipfun('dc', inverse_jacobi_f('dc', 5, 0), 0))
    mpf('5.0000000000000018')
    sage: chop(ellipfun('dc', inverse_jacobi_f('dc', -4, 2), 2))
    mpf('-4.0000000000000018')

    sage: chop(ellipfun('sd', inverse_jacobi_f('sd', -4, 0), 0))
    mpf('-3.9999999999999991')
    sage: chop(ellipfun('sd', inverse_jacobi_f('sd', 7, 1), 1))
    mpf('7.0')
    sage: chop(ellipfun('sd', inverse_jacobi_f('sd', 0, 9), 9))
    mpf('0.0')
    sage: chop(ellipfun('sd', inverse_jacobi_f('sd', 8, 0.8), 0.8))
    mpf('7.9999999999999991')

    sage: chop(ellipfun('ds', inverse_jacobi_f('ds', 4, 0.25), 0.25))
    mpf('4.0')
```

sage.functions.jacobi.**jacobi**(*kind*, *z*, *m*, ***kwargs*)

The 12 Jacobi elliptic functions.

INPUT:

- `kind` – a string of the form `'pq'`, where `p`, `q` are in `c`, `d`, `n`, `s`
- `z` – a complex number

- m – a complex number; note that $m = k^2$, where k is the elliptic modulus

EXAMPLES:

```
sage: jacobi('sn', 1, 1)
tanh(1)
sage: jacobi('cd', 1, 1/2)
jacobi_cd(1, 1/2)
sage: RDF(jacobi('cd', 1, 1/2))
0.7240097216593705
sage: (RDF(jacobi('cn', 1, 1/2)), RDF(jacobi('dn', 1, 1/2)),
....:  RDF(jacobi('cn', 1, 1/2) / jacobi('dn', 1, 1/2)))
(0.5959765676721407, 0.8231610016315962, 0.7240097216593705)
sage: jsn = jacobi('sn', x, 1)
sage: P = plot(jsn, 0, 1)
```

sage.functions.jacobi.**jacobi_am_f**(*x*, *m*)

Internal function for numeric evaluation of the Jacobi amplitude function for real arguments. Procedure described in [Ehrhardt13].

REFERENCES:

TESTS:

```
sage: from mpmath import ellipf
sage: from sage.functions.jacobi import jacobi_am_f
sage: ellipf(jacobi_am_f(0.5, 1), 1)
mpf('0.5')
sage: ellipf(jacobi_am(3, 0.3), 0.3)
mpf('3.0')
sage: ellipf(jacobi_am_f(2, -0.5), -0.5)
mpf('2.0')
sage: jacobi_am_f(2, -0.5)
mpf('2.2680930777934176')
sage: jacobi_am_f(-2, -0.5)
mpf('-2.2680930777934176')
sage: jacobi_am_f(-3, 2)
mpf('0.36067407399586108')
```

CHAPTER

TWELVE

AIRY FUNCTIONS

This module implements Airy functions and their generalized derivatives. It supports symbolic functionality through Maxima and numeric evaluation through mpmath and scipy.

Airy functions are solutions to the differential equation $f''(x) - xf(x) = 0$.

Four global function symbols are immediately available, please see

- `airy_ai()`: for the Airy Ai function
- `airy_ai_prime()`: for the first differential of the Airy Ai function
- `airy_bi()`: for the Airy Bi function
- **`airy_bi_prime()`: for the first differential** of the Airy Bi function

AUTHORS:

- Oscar Gerardo Lazo Arjona (2010): initial version
- Douglas McNeil (2012): rewrite

EXAMPLES:

Verify that the Airy functions are solutions to the differential equation:

```
sage: diff(airy_ai(x), x, 2) - x * airy_ai(x)
0
sage: diff(airy_bi(x), x, 2) - x * airy_bi(x)
0
```

class `sage.functions.airy.`**`FunctionAiryAiGeneral`**

Bases: `sage.symbolic.function.BuiltinFunction`

The generalized derivative of the Airy Ai function

INPUT:

- `alpha` – Return the α-th order fractional derivative with respect to z. For $\alpha = n = 1, 2, 3, \ldots$ this gives the derivative $\mathrm{Ai}^{(n)}(z)$, and for $\alpha = -n = -1, -2, -3, \ldots$ this gives the n-fold iterated integral.

$$f_0(z) = \mathrm{Ai}(z)$$
$$f_n(z) = \int_0^z f_{n-1}(t)dt$$

- `x` – The argument of the function

EXAMPLES:

```
sage: from sage.functions.airy import airy_ai_general
sage: x, n = var('x n')
sage: airy_ai_general(-2, x)
airy_ai(-2, x)
sage: derivative(airy_ai_general(-2, x), x)
airy_ai(-1, x)
sage: airy_ai_general(n, x)
airy_ai(n, x)
sage: derivative(airy_ai_general(n, x), x)
airy_ai(n + 1, x)
```

class sage.functions.airy.**FunctionAiryAiPrime**
Bases: sage.symbolic.function.BuiltinFunction

The derivative of the Airy Ai function; see airy_ai() for the full documentation.

EXAMPLES:

```
sage: x, n = var('x n')
sage: airy_ai_prime(x)
airy_ai_prime(x)
sage: airy_ai_prime(0)
-1/3*3^(2/3)/gamma(1/3)
```

class sage.functions.airy.**FunctionAiryAiSimple**
Bases: sage.symbolic.function.BuiltinFunction

The class for the Airy Ai function.

EXAMPLES:

```
sage: from sage.functions.airy import airy_ai_simple
sage: f = airy_ai_simple(x); f
airy_ai(x)
```

class sage.functions.airy.**FunctionAiryBiGeneral**
Bases: sage.symbolic.function.BuiltinFunction

The generalized derivative of the Airy Bi function.

INPUT:

- alpha – Return the α-th order fractional derivative with respect to z. For $\alpha = n = 1, 2, 3, \ldots$ this gives the derivative $\mathrm{Bi}^{(n)}(z)$, and for $\alpha = -n = -1, -2, -3, \ldots$ this gives the n-fold iterated integral.

$$f_0(z) = \mathrm{Bi}(z)$$
$$f_n(z) = \int_0^z f_{n-1}(t)dt$$

- x – The argument of the function

EXAMPLES:

```
sage: from sage.functions.airy import airy_bi_general
sage: x, n = var('x n')
sage: airy_bi_general(-2, x)
airy_bi(-2, x)
sage: derivative(airy_bi_general(-2, x), x)
airy_bi(-1, x)
sage: airy_bi_general(n, x)
airy_bi(n, x)
```

```
sage: derivative(airy_bi_general(n, x), x)
airy_bi(n + 1, x)
```

class sage.functions.airy.**FunctionAiryBiPrime**
Bases: sage.symbolic.function.BuiltinFunction

The derivative of the Airy Bi function; see airy_bi() for the full documentation.

EXAMPLES:

```
sage: x, n = var('x n')
sage: airy_bi_prime(x)
airy_bi_prime(x)
sage: airy_bi_prime(0)
3^(1/6)/gamma(1/3)
```

class sage.functions.airy.**FunctionAiryBiSimple**
Bases: sage.symbolic.function.BuiltinFunction

The class for the Airy Bi function.

EXAMPLES:

```
sage: from sage.functions.airy import airy_bi_simple
sage: f = airy_bi_simple(x); f
airy_bi(x)
```

sage.functions.airy.**airy_ai**(*alpha*, *x=None*, *hold_derivative=True*, ***kwds*)
The Airy Ai function

The Airy Ai function $\mathrm{Ai}(x)$ is (along with $\mathrm{Bi}(x)$) one of the two linearly independent standard solutions to the Airy differential equation $f''(x) - xf(x) = 0$. It is defined by the initial conditions:

$$\mathrm{Ai}(0) = \frac{1}{2^{2/3}\Gamma\left(\frac{2}{3}\right)},$$
$$\mathrm{Ai}'(0) = -\frac{1}{2^{1/3}\Gamma\left(\frac{1}{3}\right)}.$$

Another way to define the Airy Ai function is:

$$\mathrm{Ai}(x) = \frac{1}{\pi}\int_0^\infty \cos\left(\frac{1}{3}t^3 + xt\right) dt.$$

INPUT:

- alpha – Return the α-th order fractional derivative with respect to z. For $\alpha = n = 1, 2, 3, \ldots$ this gives the derivative $\mathrm{Ai}^{(n)}(z)$, and for $\alpha = -n = -1, -2, -3, \ldots$ this gives the n-fold iterated integral.

$$f_0(z) = \mathrm{Ai}(z)$$
$$f_n(z) = \int_0^z f_{n-1}(t)dt$$

- x – The argument of the function
- hold_derivative – Whether or not to stop from returning higher derivatives in terms of $\mathrm{Ai}(x)$ and $\mathrm{Ai}'(x)$

See also:

airy_bi()

EXAMPLES:

```
sage: n, x = var('n x')
sage: airy_ai(x)
airy_ai(x)
```

It can return derivatives or integrals:

```
sage: airy_ai(2, x)
airy_ai(2, x)
sage: airy_ai(1, x, hold_derivative=False)
airy_ai_prime(x)
sage: airy_ai(2, x, hold_derivative=False)
x*airy_ai(x)
sage: airy_ai(-2, x, hold_derivative=False)
airy_ai(-2, x)
sage: airy_ai(n, x)
airy_ai(n, x)
```

It can be evaluated symbolically or numerically for real or complex values:

```
sage: airy_ai(0)
1/3*3^(1/3)/gamma(2/3)
sage: airy_ai(0.0)
0.355028053887817
sage: airy_ai(I)
airy_ai(I)
sage: airy_ai(1.0*I)
0.331493305432141 - 0.317449858968444*I
```

The functions can be evaluated numerically either using mpmath. which can compute the values to arbitrary precision, and scipy:

```
sage: airy_ai(2).n(prec=100)
0.034924130423274379135322080792
sage: airy_ai(2).n(algorithm='mpmath', prec=100)
0.034924130423274379135322080792
sage: airy_ai(2).n(algorithm='scipy')  # rel tol 1e-10
0.03492413042327323
```

And the derivatives can be evaluated:

```
sage: airy_ai(1, 0)
-1/3*3^(2/3)/gamma(1/3)
sage: airy_ai(1, 0.0)
-0.258819403792807
```

Plots:

```
sage: plot(airy_ai(x), (x, -10, 5)) + plot(airy_ai_prime(x),
....:  (x, -10, 5), color='red')
Graphics object consisting of 2 graphics primitives
```

References

- Abramowitz, Milton; Stegun, Irene A., eds. (1965), "Chapter 10"
- Wikipedia article Airy_function

`sage.functions.airy.`**`airy_bi`**(*alpha*, *x=None*, *hold_derivative=True*, ***kwds*)

The Airy Bi function

The Airy Bi function $\mathrm{Bi}(x)$ is (along with $\mathrm{Ai}(x)$) one of the two linearly independent standard solutions to the

Airy differential equation $f''(x) - xf(x) = 0$. It is defined by the initial conditions:

$$\mathrm{Bi}(0) = \frac{1}{3^{1/6}\Gamma\left(\frac{2}{3}\right)},$$
$$\mathrm{Bi}'(0) = \frac{3^{1/6}}{\Gamma\left(\frac{1}{3}\right)}.$$

Another way to define the Airy Bi function is:

$$\mathrm{Bi}(x) = \frac{1}{\pi}\int_0^\infty \left[\exp\left(xt - \frac{t^3}{3}\right) + \sin\left(xt + \frac{1}{3}t^3\right)\right] dt.$$

INPUT:

- `alpha` – Return the α-th order fractional derivative with respect to z. For $\alpha = n = 1, 2, 3, \ldots$ this gives the derivative $\mathrm{Bi}^{(n)}(z)$, and for $\alpha = -n = -1, -2, -3, \ldots$ this gives the n-fold iterated integral.

$$f_0(z) = \mathrm{Bi}(z)$$
$$f_n(z) = \int_0^z f_{n-1}(t)dt$$

- `x` – The argument of the function
- `hold_derivative` – Whether or not to stop from returning higher derivatives in terms of $\mathrm{Bi}(x)$ and $\mathrm{Bi}'(x)$

See also:

`airy_ai()`

EXAMPLES:

```
sage: n, x = var('n x')
sage: airy_bi(x)
airy_bi(x)
```

It can return derivatives or integrals:

```
sage: airy_bi(2, x)
airy_bi(2, x)
sage: airy_bi(1, x, hold_derivative=False)
airy_bi_prime(x)
sage: airy_bi(2, x, hold_derivative=False)
x*airy_bi(x)
sage: airy_bi(-2, x, hold_derivative=False)
airy_bi(-2, x)
sage: airy_bi(n, x)
airy_bi(n, x)
```

It can be evaluated symbolically or numerically for real or complex values:

```
sage: airy_bi(0)
1/3*3^(5/6)/gamma(2/3)
sage: airy_bi(0.0)
0.614926627446001
sage: airy_bi(I)
airy_bi(I)
sage: airy_bi(1.0*I)
0.648858208330395 + 0.344958634768048*I
```

The functions can be evaluated numerically using mpmath, which can compute the values to arbitrary precision, and scipy:

```
sage: airy_bi(2).n(prec=100)
3.2980949999782147102806044252
sage: airy_bi(2).n(algorithm='mpmath', prec=100)
3.2980949999782147102806044252
sage: airy_bi(2).n(algorithm='scipy')  # rel tol 1e-10
3.2980949999782134
```

And the derivatives can be evaluated:

```
sage: airy_bi(1, 0)
3^(1/6)/gamma(1/3)
sage: airy_bi(1, 0.0)
0.448288357353826
```

Plots:

```
sage: plot(airy_bi(x), (x, -10, 5)) + plot(airy_bi_prime(x),
....:  (x, -10, 5), color='red')
Graphics object consisting of 2 graphics primitives
```

References

- Abramowitz, Milton; Stegun, Irene A., eds. (1965), "Chapter 10"
- Wikipedia article Airy_function

CHAPTER

THIRTEEN

BESSEL FUNCTIONS

This module provides symbolic Bessel Functions. These functions use the mpmath library for numerical evaluation and Maxima, GiNaC, Pynac for symbolics.

The main objects which are exported from this module are:

- `bessel_J` – The Bessel J function
- `bessel_Y` – The Bessel Y function
- `bessel_I` – The Bessel I function
- `bessel_K` – The Bessel K function
- `Bessel` – A factory function for producing Bessel functions of various kinds and orders
- Bessel functions, first defined by the Swiss mathematician Daniel Bernoulli and named after Friedrich Bessel, are canonical solutions y(x) of Bessel's differential equation:

$$x^2 \frac{d^2y}{dx^2} + x\frac{dy}{dx} + \left(x^2 - \nu^2\right) y = 0,$$

  for an arbitrary complex number ν (the order).

- In this module, J_ν denotes the unique solution of Bessel's equation which is non-singular at $x = 0$. This function is known as the Bessel Function of the First Kind. This function also arises as a special case of the hypergeometric function ${}_0F_1$:

$$J_\nu(x) = \frac{x^n}{2^\nu \Gamma(\nu+1)} {}_0F_1(\nu+1, -\frac{x^2}{4}).$$

- The second linearly independent solution to Bessel's equation (which is singular at $x = 0$) is denoted by Y_ν and is called the Bessel Function of the Second Kind:

$$Y_\nu(x) = \frac{J_\nu(x)\cos(\pi\nu) - J_{-\nu}(x)}{\sin(\pi\nu)}.$$

- There are also two commonly used combinations of the Bessel J and Y Functions. The Bessel I Function, or the Modified Bessel Function of the First Kind, is defined by:

$$I_\nu(x) = i^{-\nu} J_\nu(ix).$$

  The Bessel K Function, or the Modified Bessel Function of the Second Kind, is defined by:

$$K_\nu(x) = \frac{\pi}{2} \cdot \frac{I_{-\nu}(x) - I_n(x)}{\sin(\pi\nu)}.$$

  We should note here that the above formulas for Bessel Y and K functions should be understood as limits when ν is an integer.

- It follows from Bessel's differential equation that the derivative of $J_n(x)$ with respect to x is:

$$\frac{d}{dx} J_n(x) = \frac{1}{x^n} \left(x^n J_{n-1}(x) - n x^{n-1} J_n(z) \right)$$

- Another important formulation of the two linearly independent solutions to Bessel's equation are the Hankel functions $H_\nu^{(1)}(x)$ and $H_\nu^{(2)}(x)$, defined by:

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

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

where i is the imaginary unit (and J_* and Y_* are the usual J- and Y-Bessel functions). These linear combinations are also known as Bessel functions of the third kind; they are also two linearly independent solutions of Bessel's differential equation. They are named for Hermann Hankel.

EXAMPLES:

Evaluate the Bessel J function symbolically and numerically:

```
sage: bessel_J(0, x)
bessel_J(0, x)
sage: bessel_J(0, 0)
bessel_J(0, 0)
sage: bessel_J(0, x).diff(x)
-1/2*bessel_J(1, x) + 1/2*bessel_J(-1, x)

sage: N(bessel_J(0, 0), digits = 20)
1.0000000000000000000
sage: find_root(bessel_J(0,x), 0, 5)
2.404825557695773
```

Plot the Bessel J function:

```
sage: f(x) = Bessel(0)(x); f
x |--> bessel_J(0, x)
sage: plot(f, (x, 1, 10))
Graphics object consisting of 1 graphics primitive
```

Visualize the Bessel Y function on the complex plane (set plot_points to a higher value to get more detail):

```
sage: complex_plot(bessel_Y(0, x), (-5, 5), (-5, 5), plot_points=20)
Graphics object consisting of 1 graphics primitive
```

Evaluate a combination of Bessel functions:

```
sage: f(x) = bessel_J(1, x) - bessel_Y(0, x)
sage: f(pi)
bessel_J(1, pi) - bessel_Y(0, pi)
sage: f(pi).n()
-0.0437509653365599
sage: f(pi).n(digits=50)
-0.043750965336559909054985168023342675387737118378169
```

Symbolically solve a second order differential equation with initial conditions $y(1) = a$ and $y'(1) = b$ in terms of Bessel functions:

```
sage: y = function('y', x)
sage: a, b = var('a, b')
```

```
sage: diffeq = x^2*diff(y,x,x) + x*diff(y,x) + x^2*y == 0
sage: f = desolve(diffeq, y, [1, a, b]); f
(a*bessel_Y(1, 1) + b*bessel_Y(0, 1))*bessel_J(0, x)/(bessel_J(0,
1)*bessel_Y(1, 1) - bessel_J(1, 1)*bessel_Y(0, 1)) -
(a*bessel_J(1, 1) + b*bessel_J(0, 1))*bessel_Y(0, x)/(bessel_J(0,
1)*bessel_Y(1, 1) - bessel_J(1, 1)*bessel_Y(0, 1))
```

For more examples, see the docstring for `Bessel()`.

AUTHORS:

- Benjamin Jones (2012-12-27): initial version
- Some of the documentation here has been adapted from David Joyner's original documentation of Sage's special functions module (2006).

REFERENCES:

- Abramowitz and Stegun: Handbook of Mathematical Functions, http://www.math.sfu.ca/~cbm/aands/
- http://en.wikipedia.org/wiki/Bessel_function
- mpmath Library Bessel Functions

`sage.functions.bessel.`**`Bessel`**`(*args, **kwds)`

A function factory that produces symbolic I, J, K, and Y Bessel functions. There are several ways to call this function:

- `Bessel(order, type)`
- `Bessel(order)` – type defaults to 'J'
- `Bessel(order, typ=T)`
- `Bessel(typ=T)` – order is unspecified, this is a 2-parameter function
- `Bessel()` – order is unspecified, type is 'J'

where `order` can be any integer and T must be one of the strings 'I', 'J', 'K', or 'Y'.

See the EXAMPLES below.

EXAMPLES:

Construction of Bessel functions with various orders and types:

```
sage: Bessel()
bessel_J
sage: Bessel(1)(x)
bessel_J(1, x)
sage: Bessel(1, 'Y')(x)
bessel_Y(1, x)
sage: Bessel(-2, 'Y')(x)
bessel_Y(-2, x)
sage: Bessel(typ='K')
bessel_K
sage: Bessel(0, typ='I')(x)
bessel_I(0, x)
```

Evaluation:

```
sage: f = Bessel(1)
sage: f(3.0)
0.339058958525936
sage: f(3)
```

```
bessel_J(1, 3)
sage: f(3).n(digits=50)
0.33905895852593645892551459720647889697308041819801

sage: g = Bessel(typ='J')
sage: g(1,3)
bessel_J(1, 3)
sage: g(2, 3+I).n()
0.634160370148554 + 0.0253384000032695*I
sage: abs(numerical_integral(1/pi*cos(3*sin(x)), 0.0, pi)[0] - Bessel(0, 'J')(3.0)) < 1e-15
True
```

Symbolic calculus:

```
sage: f(x) = Bessel(0, 'J')(x)
sage: derivative(f, x)
x |--> -1/2*bessel_J(1, x) + 1/2*bessel_J(-1, x)
sage: derivative(f, x, x)
x |--> 1/4*bessel_J(2, x) - 1/2*bessel_J(0, x) + 1/4*bessel_J(-2, x)
```

Verify that J_0 satisfies Bessel's differential equation numerically using the `test_relation()` method:

```
sage: y = bessel_J(0, x)
sage: diffeq = x^2*derivative(y,x,x) + x*derivative(y,x) + x^2*y == 0
sage: diffeq.test_relation(proof=False)
True
```

Conversion to other systems:

```
sage: x,y = var('x,y')
sage: f = maxima(Bessel(typ='K')(x,y))
sage: f.derivative('_SAGE_VAR_x')
%pi*csc(%pi*_SAGE_VAR_x)*('diff(bessel_i(-_SAGE_VAR_x,_SAGE_VAR_y),_SAGE_VAR_x,1)-'diff(bessel_i
sage: f.derivative('_SAGE_VAR_y')
-(bessel_k(_SAGE_VAR_x+1,_SAGE_VAR_y)+bessel_k(_SAGE_VAR_x-1,_SAGE_VAR_y))/2
```

Compute the particular solution to Bessel's Differential Equation that satisfies $y(1) = 1$ and $y'(1) = 1$, then verify the initial conditions and plot it:

```
sage: y = function('y', x)
sage: diffeq = x^2*diff(y,x,x) + x*diff(y,x) + x^2*y == 0
sage: f = desolve(diffeq, y, [1, 1, 1]); f
(bessel_Y(1, 1) + bessel_Y(0, 1))*bessel_J(0, x)/(bessel_J(0,
1)*bessel_Y(1, 1) - bessel_J(1, 1)*bessel_Y(0, 1)) - (bessel_J(1,
1) + bessel_J(0, 1))*bessel_Y(0, x)/(bessel_J(0, 1)*bessel_Y(1, 1)
- bessel_J(1, 1)*bessel_Y(0, 1))
sage: f.subs(x=1).n() # numerical verification
1.00000000000000
sage: fp = f.diff(x)
sage: fp.subs(x=1).n()
1.00000000000000

sage: f.subs(x=1).simplify_full() # symbolic verification
1
sage: fp = f.diff(x)
sage: fp.subs(x=1).simplify_full()
1

sage: plot(f, (x,0,5))
Graphics object consisting of 1 graphics primitive
```

Plotting:

```
sage: f(x) = Bessel(0)(x); f
x |--> bessel_J(0, x)
sage: plot(f, (x, 1, 10))
Graphics object consisting of 1 graphics primitive

sage: plot([ Bessel(i, 'J') for i in range(5) ], 2, 10)
Graphics object consisting of 5 graphics primitives

sage: G = Graphics()
sage: G += sum([ plot(Bessel(i), 0, 4*pi, rgbcolor=hue(sin(pi*i/10))) for i in range(5) ])
sage: show(G)
```

A recreation of Abramowitz and Stegun Figure 9.1:

```
sage: G  = plot(Bessel(0, 'J'), 0, 15, color='black')
sage: G += plot(Bessel(0, 'Y'), 0, 15, color='black')
sage: G += plot(Bessel(1, 'J'), 0, 15, color='black', linestyle='dotted')
sage: G += plot(Bessel(1, 'Y'), 0, 15, color='black', linestyle='dotted')
sage: show(G, ymin=-1, ymax=1)
```

class sage.functions.bessel.**Function_Bessel_I**

Bases: sage.symbolic.function.BuiltinFunction

The Bessel I function, or the Modified Bessel Function of the First Kind.

DEFINITION:

$$I_\nu(x) = i^{-\nu} J_\nu(ix)$$

EXAMPLES:

```
sage: bessel_I(1, x)
bessel_I(1, x)
sage: bessel_I(1.0, 1.0)
0.565159103992485
sage: n = var('n')
sage: bessel_I(n, x)
bessel_I(n, x)
sage: bessel_I(2, I).n()
-0.114903484931900
```

Examples of symbolic manipulation:

```
sage: a = bessel_I(pi, bessel_I(1, I))
sage: N(a, digits=20)
0.00026073272117205890528 - 0.0011528954889080572266*I

sage: f = bessel_I(2, x)
sage: f.diff(x)
1/2*bessel_I(3, x) + 1/2*bessel_I(1, x)
```

Special identities that bessel_I satisfies:

```
sage: bessel_I(1/2, x)
sqrt(2)*sqrt(1/(pi*x))*sinh(x)
sage: eq = bessel_I(1/2, x) == bessel_I(0.5, x)
sage: eq.test_relation()
True
```

```
sage: bessel_I(-1/2, x)
sqrt(2)*sqrt(1/(pi*x))*cosh(x)
sage: eq = bessel_I(-1/2, x) == bessel_I(-0.5, x)
sage: eq.test_relation()
True
```

Examples of asymptotic behavior:

```
sage: limit(bessel_I(0, x), x=oo)
+Infinity
sage: limit(bessel_I(0, x), x=0)
1
```

High precision and complex valued inputs:

```
sage: bessel_I(0, 1).n(128)
1.2660658777520083355982446252147175376
sage: bessel_I(0, RealField(200)(1))
1.2660658777520083355982446252147175376076703113549622068081
sage: bessel_I(0, ComplexField(200)(0.5+I))
0.80644357583493619472428518415019222845373366024179916785502 + 0.22686958987911161141397453401
```

Visualization (set plot_points to a higher value to get more detail):

```
sage: plot(bessel_I(1,x), (x,0,5), color='blue')
Graphics object consisting of 1 graphics primitive
sage: complex_plot(bessel_I(1, x), (-5, 5), (-5, 5), plot_points=20)
Graphics object consisting of 1 graphics primitive
```

ALGORITHM:

Numerical evaluation is handled by the mpmath library. Symbolics are handled by a combination of Maxima and Sage (Ginac/Pynac).

TESTS:

```
sage: N(bessel_I(1,1),500)
0.56515910399248502720769602760986330732889962162109200948029448947925564096437113409266499776
```

Check whether the return value is real whenever the argument is real (trac ticket #10251):

```
sage: bessel_I(5, 1.5) in RR
True
```

class sage.functions.bessel.**Function_Bessel_J**

Bases: sage.symbolic.function.BuiltinFunction

The Bessel J Function, denoted by bessel_J(ν, x) or $J_\nu(x)$. As a Taylor series about $x = 0$ it is equal to:

$$J_\nu(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{k!\Gamma(k+\nu+1)} \left(\frac{x}{2}\right)^{2k+\nu}$$

The parameter ν is called the order and may be any real or complex number; however, integer and half-integer values are most common. It is defined for all complex numbers x when ν is an integer or greater than zero and it diverges as $x \to 0$ for negative non-integer values of ν.

For integer orders $\nu = n$ there is an integral representation:

$$J_n(x) = \frac{1}{\pi} \int_0^\pi \cos(nt - x\sin(t))\, dt$$

This function also arises as a special case of the hypergeometric function ${}_0F_1$:

$$J_\nu(x) = \frac{x^n}{2^\nu \Gamma(\nu+1)} {}_0F_1\left(\nu+1, -\frac{x^2}{4}\right).$$

EXAMPLES:

```
sage: bessel_J(1.0, 1.0)
0.440050585744933
sage: bessel_J(2, I).n(digits=30)
-0.135747669767038281182852569995

sage: bessel_J(1, x)
bessel_J(1, x)
sage: n = var('n')
sage: bessel_J(n, x)
bessel_J(n, x)
```

Examples of symbolic manipulation:

```
sage: a = bessel_J(pi, bessel_J(1, I)); a
bessel_J(pi, bessel_J(1, I))
sage: N(a, digits=20)
0.00059023706363796717363 - 0.0026098820470081958110*I

sage: f = bessel_J(2, x)
sage: f.diff(x)
-1/2*bessel_J(3, x) + 1/2*bessel_J(1, x)
```

Comparison to a well-known integral representation of $J_1(1)$:

```
sage: A = numerical_integral(1/pi*cos(x - sin(x)), 0, pi)
sage: A[0]  # abs tol 1e-14
0.44005058574493355
sage: bessel_J(1.0, 1.0) - A[0] < 1e-15
True
```

Integration is supported directly and through Maxima:

```
sage: f = bessel_J(2, x)
sage: f.integrate(x)
1/24*x^3*hypergeometric((3/2,), (5/2, 3), -1/4*x^2)
sage: m = maxima(bessel_J(2, x))
sage: m.integrate(x)
hypergeometric([3/2],[5/2,3],-_SAGE_VAR_x^2/4)*_SAGE_VAR_x^3/24
```

Visualization (set plot_points to a higher value to get more detail):

```
sage: plot(bessel_J(1,x), (x,0,5), color='blue')
Graphics object consisting of 1 graphics primitive
sage: complex_plot(bessel_J(1, x), (-5, 5), (-5, 5), plot_points=20)
Graphics object consisting of 1 graphics primitive
```

ALGORITHM:

Numerical evaluation is handled by the mpmath library. Symbolics are handled by a combination of Maxima and Sage (Ginac/Pynac).

Check whether the return value is real whenever the argument is real (trac ticket #10251):

```
sage: bessel_J(5, 1.5) in RR
True
```

class sage.functions.bessel.**Function_Bessel_K**

Bases: sage.symbolic.function.BuiltinFunction

The Bessel K function, or the modified Bessel function of the second kind.

DEFINITION:

$$K_\nu(x) = \frac{\pi}{2}\frac{I_{-\nu}(x) - I_\nu(x)}{\sin(\nu\pi)}$$

EXAMPLES:

```
sage: bessel_K(1, x)
bessel_K(1, x)
sage: bessel_K(1.0, 1.0)
0.601907230197235
sage: n = var('n')
sage: bessel_K(n, x)
bessel_K(n, x)
sage: bessel_K(2, I).n()
-2.59288617549120 + 0.180489972066962*I
```

Examples of symbolic manipulation:

```
sage: a = bessel_K(pi, bessel_K(1, I)); a
bessel_K(pi, bessel_K(1, I))
sage: N(a, digits=20)
3.8507583115005220157 + 0.068528298579883425792*I

sage: f = bessel_K(2, x)
sage: f.diff(x)
-1/2*bessel_K(3, x) - 1/2*bessel_K(1, x)

sage: bessel_K(1/2, x)
bessel_K(1/2, x)
sage: bessel_K(1/2, -1)
bessel_K(1/2, -1)
sage: bessel_K(1/2, 1)
sqrt(1/2)*sqrt(pi)*e^(-1)
```

Examples of asymptotic behavior:

```
sage: bessel_K(0, 0.0)
+infinity
sage: limit(bessel_K(0, x), x=0)
+Infinity
sage: limit(bessel_K(0, x), x=oo)
0
```

High precision and complex valued inputs:

```
sage: bessel_K(0, 1).n(128)
0.42102443824070833333562737921260903614
sage: bessel_K(0, RealField(200)(1))
0.42102443824070833333562737921260903613621974822666047229897
sage: bessel_K(0, ComplexField(200)(0.5+I))
0.058365979093103864080375311643360048144715516692187818271179 - 0.67645499731334483535184142196
```

Visualization (set plot_points to a higher value to get more detail):

```
sage: plot(bessel_K(1,x), (x,0,5), color='blue')
Graphics object consisting of 1 graphics primitive
sage: complex_plot(bessel_K(1, x), (-5, 5), (-5, 5), plot_points=20)
Graphics object consisting of 1 graphics primitive
```

ALGORITHM:

Numerical evaluation is handled by the mpmath library. Symbolics are handled by a combination of Maxima and Sage (Ginac/Pynac).

TESTS:

Verify that trac ticket #3426 is fixed:

The Bessel K function can be evaluated numerically at complex orders:

```
sage: bessel_K(10 * I, 10).n()
9.82415743819925e-8
```

For a fixed imaginary order and increasing, real, second component the value of Bessel K is exponentially decaying:

```
sage: for x in [10, 20, 50, 100, 200]: print bessel_K(5*I, x).n()
5.27812176514912e-6
3.11005908421801e-10
2.66182488515423e-23 - 8.59622057747552e-58*I
4.11189776828337e-45 - 1.01494840019482e-80*I
1.15159692553603e-88 - 6.75787862113718e-125*I
```

Check whether the return value is real whenever the argument is real (trac ticket #10251):

```
sage: bessel_K(5, 1.5) in RR
True
```

class sage.functions.bessel.**Function_Bessel_Y**

Bases: sage.symbolic.function.BuiltinFunction

The Bessel Y functions, also known as the Bessel functions of the second kind, Weber functions, or Neumann functions.

$Y_\nu(z)$ is a holomorphic function of z on the complex plane, cut along the negative real axis. It is singular at $z = 0$. When z is fixed, $Y_\nu(z)$ is an entire function of the order ν.

DEFINITION:

$$Y_n(z) = \frac{J_\nu(z)\cos(\nu z) - J_{-\nu}(z)}{\sin(\nu z)}$$

Its derivative with respect to z is:

$$\frac{d}{dz}Y_n(z) = \frac{1}{z^n}\left(z^n Y_{n-1}(z) - n z^{n-1} Y_n(z)\right)$$

EXAMPLES:

```
sage: bessel_Y(1, x)
bessel_Y(1, x)
sage: bessel_Y(1.0, 1.0)
-0.781212821300289
sage: n = var('n')
sage: bessel_Y(n, x)
bessel_Y(n, x)
```

```
sage: bessel_Y(2, I).n()
1.03440456978312 - 0.135747669767038*I
sage: bessel_Y(0, 0).n()
-infinity
sage: bessel_Y(0, 1).n(128)
0.088256964215676957982926766023515162828
```

Examples of symbolic manipulation:

```
sage: a = bessel_Y(pi, bessel_Y(1, I)); a
bessel_Y(pi, bessel_Y(1, I))
sage: N(a, digits=20)
4.2059146571791095708 + 21.307914215321993526*I

sage: f = bessel_Y(2, x)
sage: f.diff(x)
-1/2*bessel_Y(3, x) + 1/2*bessel_Y(1, x)
```

High precision and complex valued inputs (see trac ticket #4230):

```
sage: bessel_Y(0, 1).n(128)
0.088256964215676957982926766023515162828
sage: bessel_Y(0, RealField(200)(1))
0.088256964215676957982926766023515162827817523090675546711044
sage: bessel_Y(0, ComplexField(200)(0.5+I))
0.077763160184438051408593468823822434235010300228009867784073 + 1.014233604991606915264467768
```

Visualization (set plot_points to a higher value to get more detail):

```
sage: plot(bessel_Y(1,x), (x,0,5), color='blue')
Graphics object consisting of 1 graphics primitive
sage: complex_plot(bessel_Y(1, x), (-5, 5), (-5, 5), plot_points=20)
Graphics object consisting of 1 graphics primitive
```

ALGORITHM:

Numerical evaluation is handled by the mpmath library. Symbolics are handled by a combination of Maxima and Sage (Ginac/Pynac).

TESTS:

Check whether the return value is real whenever the argument is real (trac ticket #10251):

```
sage: bessel_Y(5, 1.5) in RR
True
```

Coercion works correctly (see trac ticket #17130):

```
sage: r = bessel_Y(RealField(200)(1), 1.0); r
-0.781212821300289
sage: parent(r)
Real Field with 53 bits of precision
sage: r = bessel_Y(RealField(200)(1), 1); r
-0.78121282130028871654715000004796482054990639071644460784383
sage: parent(r)
Real Field with 200 bits of precision
```

CHAPTER

FOURTEEN

EXPONENTIAL INTEGRALS

AUTHORS:

- Benjamin Jones (2011-06-12)

This module provides easy access to many exponential integral special functions. It utilizes Maxima's special functions package and the mpmath library.

REFERENCES:

- [AS] Abramowitz and Stegun: *Handbook of Mathematical Functions*
- Wikipedia Entry: http://en.wikipedia.org/wiki/Exponential_integral
- Online Encyclopedia of Special Function: http://algo.inria.fr/esf/index.html
- NIST Digital Library of Mathematical Functions: http://dlmf.nist.gov/
- Maxima special functions package
- mpmath library

AUTHORS:

- Benjamin Jones

  Implementations of the classes `Function_exp_integral_*`.

- David Joyner and William Stein

  Authors of the code which was moved from special.py and trans.py. Implementation of `exp_int()` (from sage/functions/special.py). Implementation of `exponential_integral_1()` (from sage/functions/transcendental.py).

class `sage.functions.exp_integral.`**`Function_cos_integral`**

Bases: `sage.symbolic.function.BuiltinFunction`

The trigonometric integral $\mathrm{Ci}(z)$ defined by

$$\mathrm{Ci}(z) = \gamma + \log(z) + \int_0^z \frac{\cos(t) - 1}{t}\, dt,$$

where γ is the Euler gamma constant (`euler_gamma` in Sage), see [AS] 5.2.1.

EXAMPLES:

```
sage: z = var('z')
sage: cos_integral(z)
cos_integral(z)
sage: cos_integral(3.0)
0.119629786008000
sage: cos_integral(0)
```

```
cos_integral(0)
sage: N(cos_integral(0))
-infinity
```

Numerical evaluation for real and complex arguments is handled using mpmath:

```
sage: cos_integral(3.0)
0.119629786008000
```

The alias Ci can be used instead of $cos_integral$:

```
sage: Ci(3.0)
0.119629786008000
```

Compare `cos_integral(3.0)` to the definition of the value using numerical integration:

```
sage: N(euler_gamma + log(3.0) + integrate((cos(x)-1)/x, x, 0, 3.0) - cos_integral(3.0)) < 1e-14
True
```

Arbitrary precision and complex arguments are handled:

```
sage: N(cos_integral(3), digits=30)
0.119629786008000327626472281177
sage: cos_integral(ComplexField(100)(3+I))
0.078134230477495714401983633057 - 0.37814733904787920181190368789*I
```

The limit $\mathrm{Ci}(z)$ as $z \to \infty$ is zero:

```
sage: N(cos_integral(1e23))
-3.24053937643003e-24
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: x = var('x')
sage: f = cos_integral(x)
sage: f.diff(x)
cos(x)/x

sage: f.integrate(x)
x*cos_integral(x) - sin(x)
```

The Nielsen spiral is the parametric plot of (Si(t), Ci(t)):

```
sage: t=var('t')
sage: f(t) = sin_integral(t)
sage: g(t) = cos_integral(t)
sage: P = parametric_plot([f, g], (t, 0.5 ,20))
sage: show(P, frame=True, axes=False)
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

REFERENCES:

- http://en.wikipedia.org/wiki/Trigonometric_integral
- mpmath documentation: ci

class sage.functions.exp_integral.**Function_cosh_integral**

Bases: sage.symbolic.function.BuiltinFunction

The trigonometric integral $\mathrm{Chi}(z)$ defined by

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

see [AS] 5.2.4.

EXAMPLES:

```
sage: z = var('z')
sage: cosh_integral(z)
cosh_integral(z)
sage: cosh_integral(3.0)
4.96039209476561
```

Numerical evaluation for real and complex arguments is handled using mpmath:

```
sage: cosh_integral(1.0)
0.837866940980208
```

The alias Chi can be used instead of $cosh_integral$:

```
sage: Chi(1.0)
0.837866940980208
```

Here is an example from the mpmath documentation:

```
sage: f(x) = cosh_integral(x)
sage: find_root(f, 0.1, 1.0)
0.523822571389482...
```

Compare `cosh_integral(3.0)` to the definition of the value using numerical integration:

```
sage: N(euler_gamma + log(3.0) + integrate((cosh(x)-1)/x, x, 0, 3.0) -
...      cosh_integral(3.0)) < 1e-14
True
```

Arbitrary precision and complex arguments are handled:

```
sage: N(cosh_integral(3), digits=30)
4.96039209476560976029791763669
sage: cosh_integral(ComplexField(100)(3+I))
3.9096723099686417127843516794 + 3.0547519627014217273323873274*I
```

The limit of $\mathrm{Chi}(z)$ as $z \to \infty$ is ∞:

```
sage: N(cosh_integral(Infinity))
+infinity
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: x = var('x')
sage: f = cosh_integral(x)
sage: f.diff(x)
cosh(x)/x

sage: f.integrate(x)
x*cosh_integral(x) - sinh(x)
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

REFERENCES:

- http://en.wikipedia.org/wiki/Trigonometric_integral
- mpmath documentation: chi

class sage.functions.exp_integral.**Function_exp_integral**
Bases: sage.symbolic.function.BuiltinFunction

The generalized complex exponential integral Ei(z) defined by

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

for x > 0 and for complex arguments by analytic continuation, see [AS] 5.1.2.

EXAMPLES:

```
sage: Ei(10)
Ei(10)
sage: Ei(I)
Ei(I)
sage: Ei(3+I)
Ei(I + 3)
sage: Ei(1.3)
2.72139888023202
sage: Ei(10r)
Ei(10)
sage: Ei(1.3r)
2.7213988802320235
```

The branch cut for this function is along the negative real axis:

```
sage: Ei(-3 + 0.1*I)
-0.0129379427181693 + 3.13993830250942*I
sage: Ei(-3 - 0.1*I)
-0.0129379427181693 - 3.13993830250942*I
```

The precision for the result is deduced from the precision of the input. Convert the input to a higher precision explicitly if a result with higher precision is desired:

```
sage: Ei(RealField(300)(1.1))
2.16737827956340282358378734233807621497112737591639704719499002090327541763352339357795426
```

ALGORITHM: Uses mpmath.

TESTS:

Show that the evaluation and limit issue in trac ticket #13271 is fixed:

```
sage: var('Z')
Z
sage: (Ei(-Z)).limit(Z=oo)
0
sage: (Ei(-Z)).limit(Z=1000)
Ei(-1000)
sage: (Ei(-Z)).limit(Z=1000).n()
-5.07089306023517e-438
```

class sage.functions.exp_integral.**Function_exp_integral_e**
Bases: sage.symbolic.function.BuiltinFunction

The generalized complex exponential integral $E_n(z)$ defined by

$$\mathrm{E_n}(z) = \int_{1}^{\infty} \frac{e^{-zt}}{t^n}\, dt$$

for complex numbers n and z, see [AS] 5.1.4.

The special case where $n = 1$ is denoted in Sage by `exp_integral_e1`.

EXAMPLES:

Numerical evaluation is handled using mpmath:

```
sage: N(exp_integral_e(1,1))
0.219383934395520
sage: exp_integral_e(1, RealField(100)(1))
0.21938393439552027367716377546
```

We can compare this to PARI's evaluation of `exponential_integral_1()`:

```
sage: N(exponential_integral_1(1))
0.219383934395520
```

We can verify one case of [AS] 5.1.45, i.e. $E_n(z) = z^{n-1}\Gamma(1-n, z)$:

```
sage: N(exp_integral_e(2, 3+I))
0.00354575823814662 - 0.00973200528288687*I
sage: N((3+I)*gamma(-1, 3+I))
0.00354575823814662 - 0.00973200528288687*I
```

Maxima returns the following improper integral as a multiple of `exp_integral_e(1,1)`:

```
sage: uu = integral(e^(-x)*log(x+1),x,0,oo)
sage: uu
e*exp_integral_e(1, 1)
sage: uu.n(digits=30)
0.596347362323194074341078499369
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: x = var('x')
sage: f = exp_integral_e(2,x)
sage: f.diff(x)
-exp_integral_e(1, x)

sage: f.integrate(x)
-exp_integral_e(3, x)

sage: f = exp_integral_e(-1,x)
sage: f.integrate(x)
Ei(-x) - gamma(-1, x)
```

Some special values of `exp_integral_e` can be simplified. [AS] 5.1.23:

```
sage: exp_integral_e(0,x)
e^(-x)/x
```

[AS] 5.1.24:

```
sage: exp_integral_e(6,0)
1/5
sage: nn = var('nn')
sage: assume(nn > 1)
sage: f = exp_integral_e(nn,0)
sage: f.simplify()
1/(nn - 1)
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

class sage.functions.exp_integral.**Function_exp_integral_e1**
Bases: sage.symbolic.function.BuiltinFunction

The generalized complex exponential integral $E_1(z)$ defined by

$$\mathrm{E}_1(z) = \int_z^\infty \frac{e^{-t}}{t}\,dt$$

see [AS] 5.1.4.

EXAMPLES:

```
sage: exp_integral_e1(x)
exp_integral_e1(x)
sage: exp_integral_e1(1.0)
0.219383934395520
```

Numerical evaluation is handled using mpmath:

```
sage: N(exp_integral_e1(1))
0.219383934395520
sage: exp_integral_e1(RealField(100)(1))
0.21938393439552027367716377546
```

We can compare this to PARI's evaluation of `exponential_integral_1()`:

```
sage: N(exp_integral_e1(2.0))
0.0489005107080611
sage: N(exponential_integral_1(2.0))
0.0489005107080611
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: x = var('x')
sage: f = exp_integral_e1(x)
sage: f.diff(x)
-e^(-x)/x

sage: f.integrate(x)
-exp_integral_e(2, x)
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

class sage.functions.exp_integral.**Function_log_integral**
Bases: sage.symbolic.function.BuiltinFunction

The logarithmic integral $\mathrm{li}(z)$ defined by

$$\mathrm{li}(x) = \int_0^z \frac{dt}{\ln(t)} = \mathrm{Ei}(\ln(x))$$

for x > 1 and by analytic continuation for complex arguments z (see [AS] 5.1.3).

EXAMPLES:

Numerical evaluation for real and complex arguments is handled using mpmath:

```
sage: N(log_integral(3))
2.16358859466719
sage: N(log_integral(3), digits=30)
```

```
2.1635885946671919728769223673 5
sage: log_integral(ComplexField(100)(3+I))
2.2879892769816826157078450911 + 0.87232935488528370139883806779*I
sage: log_integral(0)
0
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: x = var('x')
sage: f = log_integral(x)
sage: f.diff(x)
1/log(x)

sage: f.integrate(x)
x*log_integral(x) - Ei(2*log(x))
```

Here is a test from the mpmath documentation. There are 1,925,320,391,606,803,968,923 many prime numbers less than 1e23. The value of `log_integral(1e23)` is very close to this:

```
sage: log_integral(1e23)
1.92532039161405e21
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

REFERENCES:

- http://en.wikipedia.org/wiki/Logarithmic_integral_function
- mpmath documentation: logarithmic-integral

class `sage.functions.exp_integral.`**`Function_log_integral_offset`**

Bases: `sage.symbolic.function.BuiltinFunction`

The offset logarithmic integral, or Eulerian logarithmic integral, $\mathrm{Li}(x)$ is defined by

$$\mathrm{Li}(x) = \int_2^x \frac{dt}{ln(t)} = \mathrm{li}(x) - \mathrm{li}(2)$$

for $x \geq 2$.

The offset logarithmic integral should also not be confused with the polylogarithm (also denoted by $\mathrm{Li}(x)$), which is implemented as `sage.functions.log.Function_polylog`.

$\mathrm{Li}(x)$ is identical to $\mathrm{li}(x)$ except that the lower limit of integration is 2 rather than 0 to avoid the singularity at $x = 1$ of

$$\frac{1}{ln(t)}$$

See `Function_log_integral` for details of $\mathrm{li}(x)$. Thus $\mathrm{Li}(x)$ can also be represented by

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

So we have:

```
sage: li(4.5)-li(2.0)-Li(4.5)
0.000000000000000
```

$\mathrm{Li}(x)$ is extended to complex arguments z by analytic continuation (see [AS] 5.1.3):

```
sage: Li(6.6+5.4*I)
3.97032201503632 + 2.62311237593572*I
```

The function Li is an approximation for the number of primes up to x. In fact, the famous Riemann Hypothesis is

$$|\pi(x) - \mathrm{Li}(x)| \leq \sqrt{x}\log(x).$$

For "small" x, $\mathrm{Li}(x)$ is always slightly bigger than $\pi(x)$. However it is a theorem that there are very large values of x (e.g., around 10^{316}), such that $\exists x : \pi(x) > \mathrm{Li}(x)$. See "A new bound for the smallest x with $\pi(x) > \mathrm{li}(x)$", Bays and Hudson, Mathematics of Computation, 69 (2000) 1285-1296.

Note: Definite integration returns a part symbolic and part numerical result. This is because when Li(x) is evaluated it is passed as li(x)-li(2).

EXAMPLES:

Numerical evaluation for real and complex arguments is handled using mpmath:

```
sage: N(log_integral_offset(3))
1.11842481454970
sage: N(log_integral_offset(3), digits=30)
1.11842481454969918803233347815
sage: log_integral_offset(ComplexField(100)(3+I))
1.2428254968641898308632562019 + 0.87232935488528370139883806779*I
sage: log_integral_offset(2)
0
sage: for n in range(1,7):
....:  print '%-10s%-10s%-20s'%(10^n, prime_pi(10^n), N(Li(10^n)))
10        4         5.12043572466980
100       25        29.0809778039621
1000      168       176.564494210035
10000     1229      1245.09205211927
100000    9592      9628.76383727068
1000000   78498     78626.5039956821
```

Here is a test from the mpmath documentation. There are 1,925,320,391,606,803,968,923 prime numbers less than 1e23. The value of `log_integral_offset(1e23)` is very close to this:

```
sage: log_integral_offset(1e23)
1.92532039161405e21
```

Symbolic derivatives are handled by Sage and integration by Maxima:

```
sage: x = var('x')
sage: f = log_integral_offset(x)
sage: f.diff(x)
1/log(x)
sage: f.integrate(x)
-x*log_integral(2) + x*log_integral(x) - Ei(2*log(x))
sage: Li(x).integrate(x,2.0,4.5)
-2.5*log_integral(2) + 5.799321147411334
sage: N(f.integrate(x,2.0,3.0))   # abs tol 1e-15
0.601621785860587
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

REFERENCES:

- http://en.wikipedia.org/wiki/Logarithmic_integral_function
- mpmath documentation: logarithmic-integral

class sage.functions.exp_integral.**Function_sin_integral**
Bases: sage.symbolic.function.BuiltinFunction

The trigonometric integral $\mathrm{Si}(z)$ defined by

$$\mathrm{Si}(z) = \int_0^z \frac{\sin(t)}{t}\, dt,$$

see [AS] 5.2.1.

EXAMPLES:

Numerical evaluation for real and complex arguments is handled using mpmath:

```
sage: sin_integral(0)
0
sage: sin_integral(0.0)
0.000000000000000
sage: sin_integral(3.0)
1.84865252799947
sage: N(sin_integral(3), digits=30)
1.84865252799946825639773025111
sage: sin_integral(ComplexField(100)(3+I))
2.0277151656451253616038525998 + 0.015210926166954211913653130271*I
```

The alias Si can be used instead of $sin_integral$:

```
sage: Si(3.0)
1.84865252799947
```

The limit of $\mathrm{Si}(z)$ as $z \to \infty$ is $\pi/2$:

```
sage: N(sin_integral(1e23))
1.57079632679490
sage: N(pi/2)
1.57079632679490
```

At 200 bits of precision $\mathrm{Si}(10^{23})$ agrees with $\pi/2$ up to 10^{-24}:

```
sage: sin_integral(RealField(200)(1e23))
1.5707963267948966192313288218697837425815368604836679189519
sage: N(pi/2, prec=200)
1.5707963267948966192313216916397514420985846996875529104875
```

The exponential sine integral is analytic everywhere:

```
sage: sin_integral(-1.0)
-0.946083070367183
sage: sin_integral(-2.0)
-1.60541297680269
sage: sin_integral(-1e23)
-1.57079632679490
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: x = var('x')
sage: f = sin_integral(x)
sage: f.diff(x)
sin(x)/x
```

```
sage: f.integrate(x)
x*sin_integral(x) + cos(x)

sage: integrate(sin(x)/x, x)
-1/2*I*Ei(I*x) + 1/2*I*Ei(-I*x)
```

Compare values of the functions $\mathrm{Si}(x)$ and $f(x) = (1/2)i \cdot \mathrm{Ei}(-ix) - (1/2)i \cdot \mathrm{Ei}(ix) - \pi/2$, which are both anti-derivatives of $\sin(x)/x$, at some random positive real numbers:

```
sage: f(x) = 1/2*I*Ei(-I*x) - 1/2*I*Ei(I*x) - pi/2
sage: g(x) = sin_integral(x)
sage: R = [ abs(RDF.random_element()) for i in range(100) ]
sage: all(abs(f(x) - g(x)) < 1e-10 for x in R)
True
```

The Nielsen spiral is the parametric plot of (Si(t), Ci(t)):

```
sage: x=var('x')
sage: f(x) = sin_integral(x)
sage: g(x) = cos_integral(x)
sage: P = parametric_plot([f, g], (x, 0.5 ,20))
sage: show(P, frame=True, axes=False)
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

REFERENCES:

- http://en.wikipedia.org/wiki/Trigonometric_integral
- mpmath documentation: si

class sage.functions.exp_integral.**Function_sinh_integral**

Bases: sage.symbolic.function.BuiltinFunction

The trigonometric integral $\mathrm{Shi}(z)$ defined by

$$\mathrm{Shi}(z) = \int_0^z \frac{\sinh(t)}{t}\, dt,$$

see [AS] 5.2.3.

EXAMPLES:

Numerical evaluation for real and complex arguments is handled using mpmath:

```
sage: sinh_integral(3.0)
4.97344047585981
sage: sinh_integral(1.0)
1.05725087537573
sage: sinh_integral(-1.0)
-1.05725087537573
```

The alias Shi can be used instead of $sinh_integral$:

```
sage: Shi(3.0)
4.97344047585981
```

Compare `sinh_integral(3.0)` to the definition of the value using numerical integration:

```
sage: N(integrate((sinh(x))/x, x, 0, 3.0) - sinh_integral(3.0)) < 1e-14
True
```

Arbitrary precision and complex arguments are handled:

```
sage: N(sinh_integral(3), digits=30)
4.97344047585980679771041838252
sage: sinh_integral(ComplexField(100)(3+I))
3.9134623660329374406788354078 + 3.0427678212908839256360163759*I
```

The limit $\mathrm{Shi}(z)$ as $z \to \infty$ is ∞:

```
sage: N(sinh_integral(Infinity))
+infinity
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: x = var('x')
sage: f = sinh_integral(x)
sage: f.diff(x)
sinh(x)/x

sage: f.integrate(x)
x*sinh_integral(x) - cosh(x)
```

Note that due to some problems with the way Maxima handles these expressions, definite integrals can sometimes give unexpected results (typically when using inexact endpoints) due to inconsistent branching:

```
sage: integrate(sinh_integral(x), x, 0, 1/2)
-cosh(1/2) + 1/2*sinh_integral(1/2) + 1
sage: integrate(sinh_integral(x), x, 0, 1/2).n() # correct
0.125872409703453
sage: integrate(sinh_integral(x), x, 0, 0.5).n() # fixed in maxima 5.29.1
0.125872409703453
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

REFERENCES:

- http://en.wikipedia.org/wiki/Trigonometric_integral
- mpmath documentation: shi

sage.functions.exp_integral.**exponential_integral_1**(*x*, *n=0*)

Returns the exponential integral $E_1(x)$. If the optional argument n is given, computes list of the first n values of the exponential integral $E_1(xm)$.

The exponential integral $E_1(x)$ is

$$E_1(x) = \int_x^\infty e^{-t}/t dt$$

INPUT:

- `x` – a positive real number
- `n` – (default: 0) a nonnegative integer; if nonzero, then return a list of values `E_1(x*m)` for m = 1,2,3,...,n. This is useful, e.g., when computing derivatives of L-functions.

OUTPUT:

A real number if n is 0 (the default) or a list of reals if n > 0. The precision is the same as the input, with a default of 53 bits in case the input is exact.

EXAMPLES:

```
sage: exponential_integral_1(2)
0.0489005107080611
sage: exponential_integral_1(2, 4)  # abs tol 1e-18
[0.0489005107080611, 0.00377935240984891, 0.000360082452162659, 0.0000376656228439245]
sage: exponential_integral_1(40, 5)
[0.000000000000000, 2.22854325868847e-37, 6.33732515501151e-55, 2.02336191509997e-72, 6.88522610
sage: exponential_integral_1(0)
+Infinity
sage: r = exponential_integral_1(RealField(150)(1))
sage: r
0.21938393439552027367716377546012164903104729
sage: parent(r)
Real Field with 150 bits of precision
sage: exponential_integral_1(RealField(150)(100))
3.68359776168203218023519262050811898765522 01e-46
```

TESTS:

The relative error for a single value should be less than 1 ulp:

```
sage: for prec in [20..1000]:  # long time (22s on sage.math, 2013)
....:     R = RealField(prec)
....:     S = RealField(prec+64)
....:     for t in range(8):  # Try 8 values for each precision
....:         a = R.random_element(-15,10).exp()
....:         x = exponential_integral_1(a)
....:         y = exponential_integral_1(S(a))
....:         e = float(abs(S(x) - y)/x.ulp())
....:         if e >= 1.0:
....:             print "exponential_integral_1(%s) with precision %s has error of %s ulp"%(a, p
```

The absolute error for a vector should be less than $c2^{-p}$, where p is the precision in bits of x and $c = 2max(1, exponential_integral_1(x))$:

```
sage: for prec in [20..128]:  # long time (15s on sage.math, 2013)
....:     R = RealField(prec)
....:     S = RealField(prec+64)
....:     a = R.random_element(-15,10).exp()
....:     n = 2^ZZ.random_element(14)
....:     x = exponential_integral_1(a, n)
....:     y = exponential_integral_1(S(a), n)
....:     c = RDF(2 * max(1.0, y[0]))
....:     for i in range(n):
....:         e = float(abs(S(x[i]) - y[i]) << prec)
....:         if e >= c:
....:             print "exponential_integral_1(%s, %s)[%s] with precision %s has error of %s >=
```

ALGORITHM: use the PARI C-library function `eint1`.

REFERENCE:

- See Proposition 5.6.12 of Cohen's book "A Course in Computational Algebraic Number Theory".

CHAPTER

FIFTEEN

WIGNER, CLEBSCH-GORDAN, RACAH, AND GAUNT COEFFICIENTS

Collection of functions for calculating Wigner 3j, 6j, 9j, Clebsch-Gordan, Racah as well as Gaunt coefficients exactly, all evaluating to a rational number times the square root of a rational number [Rasch03].

Please see the description of the individual functions for further details and examples.

REFERENCES:

AUTHORS:

- Jens Rasch (2009-03-24): initial version for Sage
- Jens Rasch (2009-05-31): updated to sage-4.0

sage.functions.wigner.**clebsch_gordan**(*j_1*, *j_2*, *j_3*, *m_1*, *m_2*, *m_3*, *prec=None*)

Calculates the Clebsch-Gordan coefficient $\langle j_1 m_1\ j_2 m_2 | j_3 m_3 \rangle$.

The reference for this function is [Edmonds74].

INPUT:

- `j_1`, `j_2`, `j_3`, `m_1`, `m_2`, `m_3` - integer or half integer
- `prec` - precision, default: `None`. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

EXAMPLES:

```
sage: simplify(clebsch_gordan(3/2,1/2,2, 3/2,1/2,2))
1
sage: clebsch_gordan(1.5,0.5,1, 1.5,-0.5,1)
1/2*sqrt(3)
sage: clebsch_gordan(3/2,1/2,1, -1/2,1/2,0)
-sqrt(3)*sqrt(1/6)
```

NOTES:

The Clebsch-Gordan coefficient will be evaluated via its relation to Wigner 3j symbols:

$$\langle j_1 m_1\ j_2 m_2 | j_3 m_3 \rangle = (-1)^{j_1 - j_2 + m_3} \sqrt{2j_3 + 1}\ Wigner3j(j_1, j_2, j_3, m_1, m_2, -m_3)$$

See also the documentation on Wigner 3j symbols which exhibit much higher symmetry relations than the Clebsch-Gordan coefficient.

AUTHORS:

- Jens Rasch (2009-03-24): initial version

sage.functions.wigner.**gaunt**(*l_1*, *l_2*, *l_3*, *m_1*, *m_2*, *m_3*, *prec=None*)

Calculate the Gaunt coefficient.

The Gaunt coefficient is defined as the integral over three spherical harmonics:

$$Y(j_1,j_2,j_3,m_1,m_2,m_3) = \int Y_{l_1,m_1}(\Omega)Y_{l_2,m_2}(\Omega)Y_{l_3,m_3}(\Omega)d\Omega = \sqrt{(2l_1+1)(2l_2+1)(2l_3+1)/(4\pi)}\, Y(j_1,j_2,j_3,0,0,0)\ Y$$

INPUT:

- `l_1`, `l_2`, `l_3`, `m_1`, `m_2`, `m_3` - integer
- `prec` - precision, default: `None`. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

EXAMPLES:

```
sage: gaunt(1,0,1,1,0,-1)
-1/2/sqrt(pi)
sage: gaunt(1,0,1,1,0,0)
0
sage: gaunt(29,29,34,10,-5,-5)
1821867940156/215552371055153321*sqrt(22134)/sqrt(pi)
sage: gaunt(20,20,40,1,-1,0)
28384503878959800/74029560764440771/sqrt(pi)
sage: gaunt(12,15,5,2,3,-5)
91/124062*sqrt(36890)/sqrt(pi)
sage: gaunt(10,10,12,9,3,-12)
-98/62031*sqrt(6279)/sqrt(pi)
sage: gaunt(1000,1000,1200,9,3,-12).n(64)
0.00689500421922113448
```

If the sum of the l_i is odd, the answer is zero, even for Python ints (see trac ticket #14766):

```
sage: gaunt(1,2,2,1,0,-1)
0
sage: gaunt(int(1),int(2),int(2),1,0,-1)
0
```

It is an error to use non-integer values for l or m:

```
sage: gaunt(1.2,0,1.2,0,0,0)
Traceback (most recent call last):
...
TypeError: Attempt to coerce non-integral RealNumber to Integer
sage: gaunt(1,0,1,1.1,0,-1.1)
Traceback (most recent call last):
...
TypeError: Attempt to coerce non-integral RealNumber to Integer
```

NOTES:

The Gaunt coefficient obeys the following symmetry rules:

- invariant under any permutation of the columns

$$Y(j_1,j_2,j_3,m_1,m_2,m_3) = Y(j_3,j_1,j_2,m_3,m_1,m_2) = Y(j_2,j_3,j_1,m_2,m_3,m_1) = Y(j_3,j_2,j_1,m_3,m_2,m_1) = Y(j_1,j$$

•invariant under space inflection, i.e.

$$Y(j_1, j_2, j_3, m_1, m_2, m_3) = Y(j_1, j_2, j_3, -m_1, -m_2, -m_3)$$

•symmetric with respect to the 72 Regge symmetries as inherited for the $3j$ symbols [Regge58]

•zero for l_1, l_2, l_3 not fulfilling triangle relation

•zero for violating any one of the conditions: $l_1 \geq |m_1|$, $l_2 \geq |m_2|$, $l_3 \geq |m_3|$

•non-zero only for an even sum of the l_i, i.e. $J = l_1 + l_2 + l_3 = 2n$ for n in $\mathbf{N}$

ALGORITHM:

This function uses the algorithm of [Liberatodebrito82] to calculate the value of the Gaunt coefficient exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

REFERENCES:

AUTHORS:

•Jens Rasch (2009-03-24): initial version for Sage

sage.functions.wigner.**racah**(*aa*, *bb*, *cc*, *dd*, *ee*, *ff*, *prec=None*)

Calculate the Racah symbol $W(a, b, c, d; e, f)$.

INPUT:

•a, ..., f - integer or half integer

•prec - precision, default: None. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if prec=None), or real number if a precision is given.

EXAMPLES:

```
sage: racah(3,3,3,3,3,3)
-1/14
```

NOTES:

The Racah symbol is related to the Wigner 6j symbol:

$$Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6) = (-1)^{j_1+j_2+j_4+j_5} W(j_1, j_2, j_5, j_4, j_3, j_6)$$

Please see the 6j symbol for its much richer symmetries and for additional properties.

ALGORITHM:

This function uses the algorithm of [Edmonds74] to calculate the value of the 6j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

AUTHORS:

•Jens Rasch (2009-03-24): initial version

sage.functions.wigner.**wigner_3j**(*j_1*, *j_2*, *j_3*, *m_1*, *m_2*, *m_3*, *prec=None*)

Calculate the Wigner 3j symbol $Wigner3j(j_1, j_2, j_3, m_1, m_2, m_3)$.

INPUT:

•j_1, j_2, j_3, m_1, m_2, m_3 - integer or half integer

- prec - precision, default: None. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if prec=None), or real number if a precision is given.

EXAMPLES:

```
sage: wigner_3j(2, 6, 4, 0, 0, 0)
sqrt(5/143)
sage: wigner_3j(2, 6, 4, 0, 0, 1)
0
sage: wigner_3j(0.5, 0.5, 1, 0.5, -0.5, 0)
sqrt(1/6)
sage: wigner_3j(40, 100, 60, -10, 60, -50)
95608/18702538494885*sqrt(21082735836735314343364163310/220491455010479533763)
sage: wigner_3j(2500, 2500, 5000, 2488, 2400, -4888, prec=64)
7.604244568834485 89e-12
```

It is an error to have arguments that are not integer or half integer values:

```
sage: wigner_3j(2.1, 6, 4, 0, 0, 0)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer
sage: wigner_3j(2, 6, 4, 1, 0, -1.1)
Traceback (most recent call last):
...
ValueError: m values must be integer or half integer
```

NOTES:

The Wigner 3j symbol obeys the following symmetry rules:

- invariant under any permutation of the columns (with the exception of a sign change where $J := j_1 + j_2 + j_3$):

$$Wigner3j(j_1, j_2, j_3, m_1, m_2, m_3) = Wigner3j(j_3, j_1, j_2, m_3, m_1, m_2) = Wigner3j(j_2, j_3, j_1, m_2, m_3, m_1) = (-1)^J W$$

- invariant under space inflection, i.e.

$$Wigner3j(j_1, j_2, j_3, m_1, m_2, m_3) = (-1)^J Wigner3j(j_1, j_2, j_3, -m_1, -m_2, -m_3)$$

- symmetric with respect to the 72 additional symmetries based on the work by [Regge58]
- zero for j_1, j_2, j_3 not fulfilling triangle relation
- zero for $m_1 + m_2 + m_3 \neq 0$
- zero for violating any one of the conditions $j_1 \geq |m_1|$, $j_2 \geq |m_2|$, $j_3 \geq |m_3|$

ALGORITHM:

This function uses the algorithm of [Edmonds74] to calculate the value of the 3j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

REFERENCES:

AUTHORS:

- Jens Rasch (2009-03-24): initial version

sage.functions.wigner.**wigner_6j**(*j_1*, *j_2*, *j_3*, *j_4*, *j_5*, *j_6*, *prec=None*)

Calculate the Wigner 6j symbol $Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6)$.

INPUT:

- `j_1`, ..., `j_6` - integer or half integer
- `prec` - precision, default: `None`. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

EXAMPLES:

```
sage: wigner_6j(3,3,3,3,3,3)
-1/14
sage: wigner_6j(5,5,5,5,5,5)
1/52
sage: wigner_6j(6,6,6,6,6,6)
309/10868
sage: wigner_6j(8,8,8,8,8,8)
-12219/965770
sage: wigner_6j(30,30,30,30,30,30)
360821868690334795814816948289817141 24
sage: wigner_6j(0.5,0.5,1,0.5,0.5,1)
1/6
sage: wigner_6j(200,200,200,200,200,200, prec=1000)*1.0
0.000155903212413242
```

It is an error to have arguments that are not integer or half integer values or do not fulfill the triangle relation:

```
sage: wigner_6j(2.5,2.5,2.5,2.5,2.5,2.5)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the triangle relation
sage: wigner_6j(0.5,0.5,1.1,0.5,0.5,1.1)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the triangle relation
```

NOTES:

The Wigner 6j symbol is related to the Racah symbol but exhibits more symmetries as detailed below.

$$Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6) = (-1)^{j_1+j_2+j_4+j_5} W(j_1, j_2, j_5, j_4, j_3, j_6)$$

The Wigner 6j symbol obeys the following symmetry rules:

- Wigner 6j symbols are left invariant under any permutation of the columns:

$$Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6) = Wigner6j(j_3, j_1, j_2, j_6, j_4, j_5) = Wigner6j(j_2, j_3, j_1, j_5, j_6, j_4) = Wigner6j(j_3, j_2, j$$

- They are invariant under the exchange of the upper and lower arguments in each of any two columns, i.e.

$$Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6) = Wigner6j(j_1, j_5, j_6, j_4, j_2, j_3) = Wigner6j(j_4, j_2, j_6, j_1, j_5, j_3) = Wigner6j(j_4, j_5, j$$

- additional 6 symmetries [Regge59] giving rise to 144 symmetries in total
- only non-zero if any triple of j's fulfill a triangle relation

ALGORITHM:

This function uses the algorithm of [Edmonds74] to calculate the value of the 6j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

REFERENCES:

sage.functions.wigner.**wigner_9j**(*j_1*, *j_2*, *j_3*, *j_4*, *j_5*, *j_6*, *j_7*, *j_8*, *j_9*, *prec=None*)
Calculate the Wigner 9j symbol $Wigner9j(j_1,j_2,j_3,j_4,j_5,j_6,j_7,j_8,j_9)$.

INPUT:

- `j_1`, ..., `j_9` - integer or half integer
- `prec` - precision, default: `None`. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

EXAMPLES:

A couple of examples and test cases, note that for speed reasons a precision is given:

```
sage: wigner_9j(1,1,1, 1,1,1, 1,1,0 ,prec=64) # ==1/18
0.055555555555555555
sage: wigner_9j(1,1,1, 1,1,1, 1,1,1)
0
sage: wigner_9j(1,1,1, 1,1,1, 1,1,2 ,prec=64) # ==1/18
0.055555555555555556
sage: wigner_9j(1,2,1, 2,2,2, 1,2,1 ,prec=64) # ==-1/150
-0.0066666666666666667
sage: wigner_9j(3,3,2, 2,2,2, 3,3,2 ,prec=64) # ==157/14700
0.010680272108843537
sage: wigner_9j(3,3,2, 3,3,2, 3,3,2 ,prec=64) # ==3221*sqrt(70)/(246960*sqrt(105)) - 365/(3528*s
0.009442477466511117 39
sage: wigner_9j(3,3,1, 3.5,3.5,2, 3.5,3.5,1 ,prec=64) # ==3221*sqrt(70)/(246960*sqrt(105)) - 365
0.011021667854435136 4
sage: wigner_9j(100,80,50, 50,100,70, 60,50,100 ,prec=1000)*1.0
1.05597798065761e-7
sage: wigner_9j(30,30,10, 30.5,30.5,20, 30.5,30.5,10 ,prec=1000)*1.0 # ==(8094468018635996899 0/9
0.0000325841699408828
sage: wigner_9j(64,62.5,114.5, 61.5,61,112.5, 113.5,110.5,60, prec=1000)*1.0
-3.41407910055520e-39
sage: wigner_9j(15,15,15, 15,3,15, 15,18,10, prec=1000)*1.0
-0.00007783246153095 39
sage: wigner_9j(1.5,1,1.5, 1,1,1, 1.5,1,1.5)
0
```

It is an error to have arguments that are not integer or half integer values or do not fulfill the triangle relation:

```
sage: wigner_9j(0.5,0.5,0.5, 0.5,0.5,0.5, 0.5,0.5,0.5,prec=64)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the triangle relation
sage: wigner_9j(1,1,1, 0.5,1,1.5, 0.5,1,2.5,prec=64)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the triangle relation
```

ALGORITHM:

This function uses the algorithm of [Edmonds74] to calculate the value of the 3j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

CHAPTER

SIXTEEN

GENERALIZED FUNCTIONS

Sage implements several generalized functions (also known as distributions) such as Dirac delta, Heaviside step functions. These generalized functions can be manipulated within Sage like any other symbolic functions.

AUTHORS:

- Golam Mortuza Hossain (2009-06-26): initial version

EXAMPLES:

Dirac delta function:

```
sage: dirac_delta(x)
dirac_delta(x)
```

Heaviside step function:

```
sage: heaviside(x)
heaviside(x)
```

Unit step function:

```
sage: unit_step(x)
unit_step(x)
```

Signum (sgn) function:

```
sage: sgn(x)
sgn(x)
```

Kronecker delta function:

```
sage: m,n=var('m,n')
sage: kronecker_delta(m,n)
kronecker_delta(m, n)
```

class sage.functions.generalized.**FunctionDiracDelta**

Bases: sage.symbolic.function.BuiltinFunction

The Dirac delta (generalized) function, $\delta(x)$ (dirac_delta(x)).

INPUT:

- x - a real number or a symbolic expression

DEFINITION:

Dirac delta function $\delta(x)$, is defined in Sage as:

$\delta(x) = 0$ for real $x \neq 0$ and $\int_{-\infty}^{\infty} \delta(x)dx = 1$

Its alternate definition with respect to an arbitrary test function $f(x)$ is

$\int_{-\infty}^{\infty} f(x)\delta(x-a)dx = f(a)$

EXAMPLES:

```
sage: dirac_delta(1)
0
sage: dirac_delta(0)
dirac_delta(0)
sage: dirac_delta(x)
dirac_delta(x)
sage: integrate(dirac_delta(x), x, -1, 1, algorithm='sympy')
1
```

REFERENCES:

- http://en.wikipedia.org/wiki/Dirac_delta_function

class sage.functions.generalized.**FunctionHeaviside**

Bases: sage.symbolic.function.BuiltinFunction

The Heaviside step function, $H(x)$ (`heaviside(x)`).

INPUT:

- `x` - a real number or a symbolic expression

DEFINITION:

The Heaviside step function, $H(x)$ is defined in Sage as:

$H(x) = 0$ for $x < 0$ and $H(x) = 1$ for $x > 0$

EXAMPLES:

```
sage: heaviside(-1)
0
sage: heaviside(1)
1
sage: heaviside(0)
heaviside(0)
sage: heaviside(x)
heaviside(x)
```

TESTS:

```
sage: heaviside(x)._sympy_()
Heaviside(x)
```

REFERENCES:

- Wikipedia article Heaviside_function

class sage.functions.generalized.**FunctionKroneckerDelta**

Bases: sage.symbolic.function.BuiltinFunction

The Kronecker delta function $\delta_{m,n}$ (`kronecker_delta(m, n)`).

INPUT:

- `m` - a number or a symbolic expression
- `n` - a number or a symbolic expression

DEFINITION:

Kronecker delta function $\delta_{m,n}$ is defined as:

$\delta_{m,n} = 0$ for $m \neq n$ and $\delta_{m,n} = 1$ for $m = n$

EXAMPLES:

```
sage: kronecker_delta(1,2)
0
sage: kronecker_delta(1,1)
1
sage: m,n=var('m,n')
sage: kronecker_delta(m,n)
kronecker_delta(m, n)
```

REFERENCES:

- http://en.wikipedia.org/wiki/Kronecker_delta

class sage.functions.generalized.**FunctionSignum**

Bases: sage.symbolic.function.BuiltinFunction

The signum or sgn function $\mathrm{sgn}(x)$ (`sgn(x)`).

INPUT:

- `x` - a real number or a symbolic expression

DEFINITION:

The sgn function, $\mathrm{sgn}(x)$ is defined as:

$\mathrm{sgn}(x) = 1$ for $x > 0$, $\mathrm{sgn}(x) = 0$ for $x = 0$ and $\mathrm{sgn}(x) = -1$ for $x < 0$

EXAMPLES:

```
sage: sgn(-1)
-1
sage: sgn(1)
1
sage: sgn(0)
0
sage: sgn(x)
sgn(x)
```

We can also use `sign`:

```
sage: sign(1)
1
sage: sign(0)
0
sage: a = AA(-5).nth_root(7)
sage: sign(a)
-1
```

TESTS:

Check if conversion to sympy works trac ticket #11921:

```
sage: sgn(x)._sympy_()
sign(x)
```

REFERENCES:

- http://en.wikipedia.org/wiki/Sign_function

class sage.functions.generalized.**FunctionUnitStep**

Bases: sage.symbolic.function.BuiltinFunction

The unit step function, $\mathrm{u}(x)$ (unit_step(x)).

INPUT:

- x - a real number or a symbolic expression

DEFINITION:

The unit step function, $\mathrm{u}(x)$ is defined in Sage as:

$\mathrm{u}(x) = 0$ for $x < 0$ and $\mathrm{u}(x) = 1$ for $x \geq 0$

EXAMPLES:

```
sage: unit_step(-1)
0
sage: unit_step(1)
1
sage: unit_step(0)
1
sage: unit_step(x)
unit_step(x)
```

CHAPTER

SEVENTEEN

COUNTING PRIMES

AUTHORS:

- R. Andrew Ohana (2009): initial version of efficient prime_pi
- William Stein (2009): fix plot method
- R. Andrew Ohana (2011): complete rewrite, ~5x speedup

EXAMPLES:

```
sage: z = sage.functions.prime_pi.PrimePi()
sage: loads(dumps(z))
prime_pi
sage: loads(dumps(z)) == z
True
```

class sage.functions.prime_pi.**PrimePi**

Bases: sage.symbolic.function.BuiltinFunction

The prime counting function, which counts the number of primes less than or equal to a given value.

INPUT:

- x - a real number
- prime_bound - (default 0) a real number < 2^32, prime_pi will make sure to use all the primes up to prime_bound (although, possibly more) in computing prime_pi, this can potentially speedup the time of computation, at a cost to memory usage.

OUTPUT:

integer – the number of primes $\leq$ x

EXAMPLES:

These examples test common inputs:

```
sage: prime_pi(7)
4
sage: prime_pi(100)
25
sage: prime_pi(1000)
168
sage: prime_pi(100000)
9592
sage: prime_pi(500509)
41581
```

These examples test a variety of odd inputs:

```
sage: prime_pi(3.5)
2
sage: prime_pi(sqrt(2357))
15
sage: prime_pi(mod(30957, 9750979))
3337
```

We test non-trivial `prime_bound` values:

```
sage: prime_pi(100000, 10000)
9592
sage: prime_pi(500509, 50051)
41581
```

The following test is to verify that ticket #4670 has been essentially resolved:

```
sage: prime_pi(10^10)
455052511
```

The `prime_pi` function also has a special plotting method, so it plots quickly and perfectly as a step function:

```
sage: P = plot(prime_pi, 50, 100)
```

NOTES:

Uses a recursive implementation, using the optimizations described in [RAO2011].

REFERENCES:

AUTHOR:

- R. Andrew Ohana (2011)

plot(*xmin=0*, *xmax=100*, *vertical_lines=True*, ***kwds*)

Draw a plot of the prime counting function from `xmin` to `xmax`. All additional arguments are passed on to the line command.

WARNING: we draw the plot of `prime_pi` as a stairstep function with explicitly drawn vertical lines where the function jumps. Technically there should not be any vertical lines, but they make the graph look much better, so we include them. Use the option `vertical_lines=False` to turn these off.

EXAMPLES:

```
sage: plot(prime_pi, 1, 100)
Graphics object consisting of 1 graphics primitive
sage: prime_pi.plot(-2, sqrt(2501), thickness=2, vertical_lines=False)
Graphics object consisting of 16 graphics primitives
```

sage.functions.prime_pi.**legendre_phi**(*x*, *a*)

Legendre's formula, also known as the partial sieve function, is a useful combinatorial function for computing the prime counting function (the `prime_pi` method in Sage). It counts the number of positive integers $\leq$ `x` that are not divisible by the first `a` primes.

INPUT:

- `x` – a real number
- `a` – a non-negative integer

OUTPUT:

integer – the number of positive integers $\leq$ `x` that are not divisible by the first `a` primes

EXAMPLES:

```
sage: legendre_phi(100, 0)
100
sage: legendre_phi(29375, 1)
14688
sage: legendre_phi(91753, 5973)
2893
sage: legendre_phi(7.5, 2)
3
sage: legendre_phi(str(-2^100), 92372)
0
sage: legendre_phi(4215701455, 6450023226)
1
```

NOTES:

Uses a recursive implementation, using the optimizations described in [RAO2011].

AUTHOR:

- R. Andrew Ohana (2011)

sage.functions.prime_pi.**partial_sieve_function**(*x*, *a*)

Legendre's formula, also known as the partial sieve function, is a useful combinatorial function for computing the prime counting function (the `prime_pi` method in Sage). It counts the number of positive integers $\leq$ `x` that are not divisible by the first `a` primes.

INPUT:

- `x` – a real number
- `a` – a non-negative integer

OUTPUT:

integer – the number of positive integers $\leq$ `x` that are not divisible by the first `a` primes

EXAMPLES:

```
sage: legendre_phi(100, 0)
100
sage: legendre_phi(29375, 1)
14688
sage: legendre_phi(91753, 5973)
2893
sage: legendre_phi(7.5, 2)
3
sage: legendre_phi(str(-2^100), 92372)
0
sage: legendre_phi(4215701455, 6450023226)
1
```

NOTES:

Uses a recursive implementation, using the optimizations described in [RAO2011].

AUTHOR:

- R. Andrew Ohana (2011)

CHAPTER

EIGHTEEN

SYMBOLIC MINIMUM AND MAXIMUM

Sage provides a symbolic maximum and minimum due to the fact that the Python builtin max and min are not able to deal with variables as users might expect. These functions wait to evaluate if there are variables.

Here you can see some differences:

```
sage: max(x,x^2)
x
sage: max_symbolic(x,x^2)
max(x, x^2)
sage: f(x) = max_symbolic(x,x^2); f(1/2)
1/2
```

This works as expected for more than two entries:

```
sage: max(3,5,x)
5
sage: min(3,5,x)
3
sage: max_symbolic(3,5,x)
max(x, 5)
sage: min_symbolic(3,5,x)
min(x, 3)
```

class sage.functions.min_max.**MaxSymbolic**

Bases: sage.functions.min_max.MinMax_base

Symbolic max function.

The Python builtin max function doesn't work as expected when symbolic expressions are given as arguments. This function delays evaluation until all symbolic arguments are substituted with values.

EXAMPLES:

```
sage: max_symbolic(3, x)
max(3, x)
sage: max_symbolic(3, x).subs(x=5)
5
sage: max_symbolic(3, 5, x)
max(x, 5)
sage: max_symbolic([3,5,x])
max(x, 5)
```

TESTS:

```
sage: loads(dumps(max_symbolic(x,5)))
max(x, 5)
sage: latex(max_symbolic(x,5))
```

```
\max\left(x, 5\right)
sage: max_symbolic(x, 5)._sympy_()
Max(5, x)
```

class sage.functions.min_max.**MinMax_base**
Bases: sage.symbolic.function.BuiltinFunction

eval_helper(*this_f*, *builtin_f*, *initial_val*, *args*)
EXAMPLES:

```
sage: max_symbolic(3,5,x) # indirect doctest
max(x, 5)
sage: min_symbolic(3,5,x)
min(x, 3)
```

class sage.functions.min_max.**MinSymbolic**
Bases: sage.functions.min_max.MinMax_base

Symbolic min function.

The Python builtin min function doesn't work as expected when symbolic expressions are given as arguments. This function delays evaluation until all symbolic arguments are substituted with values.

EXAMPLES:

```
sage: min_symbolic(3, x)
min(3, x)
sage: min_symbolic(3, x).subs(x=5)
3
sage: min_symbolic(3, 5, x)
min(x, 3)
sage: min_symbolic([3,5,x])
min(x, 3)
```

TESTS:

```
sage: loads(dumps(min_symbolic(x,5)))
min(x, 5)
sage: latex(min_symbolic(x,5))
\min\left(x, 5\right)
sage: min_symbolic(x, 5)._sympy_()
Min(5, x)
```

CHAPTER

NINETEEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

[ASHandbook] Abramowitz and Stegun: Handbook of Mathematical Functions, http://www.math.sfu.ca/ cbm/aands/

[EffCheby] Wolfram Koepf: Effcient Computation of Chebyshev Polynomials in Computer Algebra Computer Algebra Systems: A Practical Guide. John Wiley, Chichester (1999): 79-99.

[KhaSuk04] A. Khare and U. Sukhatme. "Cyclic Identities Involving Jacobi Elliptic Functions". Arxiv math-ph/0201004

[Tee97] Tee, Garry J. "Continuous branches of inverses of the 12 Jacobi elliptic functions for real argument". 1997. https://researchspace.auckland.ac.nz/bitstream/handle/2292/5042/390.pdf.

[Ehrhardt13] Ehrhardt, Wolfgang. "The AMath and DAMath Special Functions: Reference Manual and Implementation Notes, Version 1.3". 2013. http://www.wolfgang-ehrhardt.de/specialfunctions.pdf.

[AS] 'Handbook of Mathematical Functions', Milton Abramowitz and Irene A. Stegun, National Bureau of Standards Applied Mathematics Series, 55. See also http://www.math.sfu.ca/~cbm/aands/.

[Rasch03] J. Rasch and A. C. H. Yu, 'Efficient Storage Scheme for Pre-calculated Wigner 3j, 6j and Gaunt Coefficients', SIAM J. Sci. Comput. Volume 25, Issue 4, pp. 1416-1428 (2003)

[Liberatodebrito82] 'FORTRAN program for the integral of three spherical harmonics', A. Liberato de Brito, Comput. Phys. Commun., Volume 25, pp. 81-85 (1982)

[Regge58] 'Symmetry Properties of Clebsch-Gordan Coefficients', T. Regge, Nuovo Cimento, Volume 10, pp. 544 (1958)

[Edmonds74] 'Angular Momentum in Quantum Mechanics', A. R. Edmonds, Princeton University Press (1974)

[Regge59] 'Symmetry Properties of Racah Coefficients', T. Regge, Nuovo Cimento, Volume 11, pp. 116 (1959)

[RAO2011] R.A. Ohana. On Prime Counting in Abelian Number Fields. http://wstein.org/home/ohanar/papers/abelian_prime_counting/main.pdf.

f

A

B

C

D

E

F

G

H

I

J

L

M

O

P

R

S

T

U

V

W

Z